COMPUTAÇÃO EVOLUCIONÁRIA

RELATÓRIO 3
Comparação do Algoritmo de Evolução Diferencial com Algoritmo de Colônia de Formigas.

Michelle Hanne Soares de Andrade

Belo Horizonte
Julho, 2025.

SUMÁRIO

1- INTRODUÇÃO

O **Algoritmo de Evolução Diferencial** (DE), conforme descrito por Chakraborty (2008), é um método de otimização inteligente e biótico, proposto inicialmente por **Storn & Price** (1995). Ele simula a lei natural da "***sobrevivência do mais apto***" para resolver problemas de otimização, especialmente em espaços contínuos multidimensionais.

O DE mantém uma população de soluções candidatas e gera novas soluções por meio da combinação de indivíduos existentes, utilizando operações básicas de mutação, *crossover,* diferenciação e seleção. A mutação é realizada pela adição da diferença ponderada entre dois vetores de parâmetros a um terceiro vetor, criando um vetor mutado. Em seguida, ocorre o *crossover*, que combina componentes do vetor mutado com um vetor alvo para formar um vetor de teste (vetor ruído). Por fim, a seleção adota uma estratégia gulosa, onde o vetor de teste substitui o vetor alvo apenas se apresentar melhor desempenho segundo a função objetivo, garantindo que a população evolua rumo a soluções ótimas.

O **Algoritmo de Colônia de Formigas (*Ant Colony Optimization* - ACO)** é uma meta-heurística bioinspirada desenvolvida inicialmente por Marco Dorigo (Dorigo et al., 1991), com base no comportamento de formigas reais durante a busca por alimento. Essas formigas depositam uma substância chamada **feromônio** ao longo dos caminhos que percorrem, permitindo que outras formigas sigam as rotas mais promissoras com maior probabilidade.

A versão de otimização contínua utilizada neste trabalho foi o ***Continuous* ACO (ACOR - *Ant Colony Optimization for Continuous Domains*)** (SOCHA & DORIGO, 2008). O algoritmo substitui a ideia de trilhas discretas por uma distribuição probabilística de amostragem de soluções em torno das melhores já encontradas. O conceito de feromônio é representado por um arquivo de com as melhores soluções anteriores, a partir do qual novas soluções são geradas por meio de amostras Gaussianas. Os parâmetros utilizados foram: ***α (alpha)***, ***β (beta)*** e ***ρ (rho).*** Sendo ***α (alpha)*** a influência das soluções boas (peso do feromônio), ***β (beta)*** a exploração ao redor das soluções e ***ρ (rho)*** a taxa de esquecimento das antigas.

Este relatório apresenta a comparação entre os Algoritmos Genéticos, Evolução Diferencial e Colônia de Formigas para resolver o problema com "***n***" variáveis. Considerando funções quadrática de *max* e *min* e a função *Rastrigin* para *n = 2.*

2- IMPLEMENTAÇÃO DOS ALGORITMOS

O problema consiste na implementação de uma função quadrática e a função *Rastrigin*. Tendo como escopo os testes abaixo:

F1 – Quadrática tendo a restrição das variáveis no intervalo de x = [−10 10].
Teste 1 – $Min\ f_1\ (x) = \ x^{\ 2}$
Teste 2 – $Max\ f_1\ (x) = \ x^{\ 2}$
F2 – Rastrigin tendo a restrição das variáveis no intervalo de x = [−5 5].
Teste 3 – $Min\ f_{\ 2}\ (x) = An + \sum_i^n=0[x_i^2 - A\cos(2x_{\ i})]$. Considere A = 10.

Neste escopo, a função objetivo é composta de *n=2* (duas dimensões), ou seja, x_1 e x_2.

Função Objetivo:

$f(x_1,x_2) = x_1^2+x_2^{\ 2}$

Os parâmetros adotados nas implementações foram:
População: 50
Gerações: 100
Mutação: 0,1
Cruzamento: 0,8

Para cada uma das heurísticas foram **30 execuções** para cada instância de teste. A seguir o Algoritmo Genético e suas classes

Classe Individual: Representa um único indivíduo (solução candidata) na população do algoritmo genético ou de evolução diferencial. Cada indivíduo possui um cromossomo (um conjunto de variáveis) e um valor de *fitness* associado.

- `chromosome`: vetor de variáveis reais que define a solução.
- `fitness`: valor da função objetivo para esse indivíduo.

- `_generate_chromosome()`: cria o vetor de variáveis com valores aleatórios dentro dos limites definidos.
- `calculate_fitness()`: avalia o indivíduo usando uma função objetivo e normaliza o valor caso seja um problema de minimização.

```
class Individual:
def __init__(self, n_variables, lower_bound, upper_bound):
self.n_variables = n_variables
self.lower_bound = lower_bound
self.upper_bound = upper_bound
self.chromosome = self._generate_chromosome()
self.fitness = 0.0

def _generate_chromosome(self):
return [random.uniform(self.lower_bound, self.upper_bound) for _ in
range(self.n_variables)]

def calculate_fitness(self, objective_function, problem_type):
self.fitness = objective_function(self.chromosome)
if problem_type == "min":
if self.fitness >= 0:
self.fitness = 1 / (1 + self.fitness)

def __repr__(self):
return f"Chromosome: {self.chromosome}, Fitness: {self.fitness:.4f}"
```

Classe GeneticAlorithm: Implementa o Algoritmo Genético (AG) para otimização. O AG é um algoritmo de busca meta-heurística inspirado no processo de seleção natural, que utiliza operadores como seleção, cruzamento (*crossover*) e mutação para evoluir uma população de soluções.

```
class GeneticAlgorithm:
def __init__(self, population_size, generations, crossover_rate,
mutation_rate,
n_variables, lower_bound, upper_bound, objective_function,
problem_type):

# Inicializa o algoritmo genético com seus parâmetros.
self.population_size = population_size
self.generations = generations
self.crossover_rate = crossover_rate
```

```
self.mutation_rate = mutation_rate
self.n_variables = n_variables
self.lower_bound = lower_bound
self.upper_bound = upper_bound
self.objective_function = objective_function
self.problem_type = problem_type
self.population = []
self.best_individual_overall = None
self.best_fitness_overall = float('-inf') if problem_type == "max" else
float('inf')
```

Cria a população inicial de indivíduos com cromossomos gerados aleatoriamente e avalia o *fitness*.

```
def _initialize_population(self):
self.population = [Individual(self.n_variables, self.lower_bound,
self.upper_bound) for _ in range(self.population_size)]
self._evaluate_population()
```

Avalia o *fitness* de cada indivíduo na população usando a função objetivo.

```
def _evaluate_population(self):
for individual in self.population:
individual.fitness = self.objective_function(individual.chromosome)
```

Seleciona dois pais da população usando o método de seleção por roleta, onde a probabilidade de um indivíduo ser selecionado é proporcional ao seu *fitness*. Para problemas de **minimização**, o *fitness* é ajustado para que valores menores do que a função objetivo tenham maior probabilidade de serem selecionados.

```
def _select_parents(self):
fitness_values = [ind.fitness for ind in self.population]
if self.problem_type == "min":

min_val = min(fitness_values)
if min_val < 0: # # Se houver fitness negativos, shift para positivo antes de inverter a
escala
adjusted_fitness = [f - min_val + 1e-6 for f in fitness_values]
else: # e todos os fitness são não-negativos, usa a técnica de maximização de 1/(1+f) ou
(max_f - f)
max_val = max(fitness_values)
adjusted_fitness = [max_val - f + 1e-6 for f in fitness_values] #
Adiciona um pequeno valor para evitar divisão por zero
else: # Para maximização, usa o fitness diretamente (garantindo que seja positivo)
```

```
adjusted_fitness = [f + 1e-6 for f in fitness_values]
total_fitness = sum(adjusted_fitness)
if total_fitness == 0: # Caso todos os fitness sejam zero, seleciona aleatoriamente
para evitar erro.
return random.sample(self.population, 2)
probabilities = [f / total_fitness for f in adjusted_fitness]
parents = random.choices(self.population, weights=probabilities, k=2) #
Seleciona 2 pais com base nas probabilidades.
return parents[0], parents[1]
```

Realiza o cruzamento (*crossover*) entre dois pais para gerar dois filhos. Utiliza o *crossover* aritmético de ponto único, onde uma **porção do cromossomo** é combinada **linearmente entre os pais.** Os valores resultantes são truncados dentro dos limites.

```
def _crossover(self, parent1, parent2):
child1_chromosome = list(parent1.chromosome)
child2_chromosome = list(parent2.chromosome)
if random.random() < self.crossover_rate:
crossover_point = random.randint(1, self.n_variables – 1) # Ponto onde o
cruzamento ocorre.
alpha = random.random()
for i in range(crossover_point, self.n_variables):
child1_chromosome[i] = alpha * parent1.chromosome[i] + (1 - alpha) *
parent2.chromosome[i]
child2_chromosome[i] = alpha * parent2.chromosome[i] + (1 - alpha) *
parent1.chromosome[i]
child1_chromosome = [max(self.lower_bound, min(self.upper_bound, val))
for val in child1_chromosome]
child2_chromosome = [max(self.lower_bound, min(self.upper_bound, val))
for val in child2_chromosome]
child1 = Individual(self.n_variables, self.lower_bound,
self.upper_bound)
child1.chromosome = child1_chromosome
child2 = Individual(self.n_variables, self.lower_bound,
self.upper_bound)
child2.chromosome = child2_chromosome
return child1, child2
```

Aplica mutação a um indivíduo com uma certa probabilidade. A mutação envolve a substituição **aleatória** de um gene no **cromossomo** por um **novo valor dentro** dos **limites do domínio.**

```
def _mutate(self, individual):
if random.random() < self.mutation_rate:
```

```
mutation_point = random.randint(0, self.n_variables - 1)
individual.chromosome[mutation_point] =
random.uniform(self.lower_bound, self.upper_bound)
```

Executa o Algoritmo Genético por um número **especificado** de **gerações**. Em cada geração, uma **nova população é criada através de seleção, cruzamento e mutação, e o melhor indivíduo é selecionado.**

```
def run(self):
self._initialize_population()
history = [] # Armazena o melhor fitness de cada geração
for generation in range(self.generations):
new_population = []
# Elitismo: o melhor indivíduo da geração atual é transferido
diretamente para a próxima população.
current_best = max(self.population, key=lambda ind: ind.fitness) if
self.problem_type == "max" else min(self.population, key=lambda ind:
ind.fitness)

# Atualiza o melhor indivíduo geral encontrado.
if self.best_individual_overall is None or \
(self.problem_type == "max" and current_best.fitness >
self.best_fitness_overall) or \
(self.problem_type == "min" and current_best.fitness <
self.best_fitness_overall):
# Cria uma cópia para evitar que alterações futuras no 'current_best'
afetem 'best_individual_overall'.
self.best_individual_overall = Individual(self.n_variables,
self.lower_bound, self.upper_bound)
self.best_individual_overall.chromosome = list(current_best.chromosome)
self.best_individual_overall.fitness = current_best.fitness
self.best_fitness_overall = current_best.fitness

new_population.append(current_best) # Adiciona o elite à nova população.

# Preenche o restante da nova população através de seleção, cruzamento
e mutação.
while len(new_population) < self.population_size:
parent1, parent2 = self._select_parents()
child1, child2 = self._crossover(parent1, parent2)
self._mutate(child1)
self._mutate(child2)
new_population.append(child1)
if len(new_population) < self.population_size:
new_population.append(child2)
self.population = new_population
self._evaluate_population() # Reavalia a nova população.
```

```
# Registra o melhor fitness da geração atual para o histórico.
best_fitness_in_gen = max(self.population, key=lambda ind:
ind.fitness).fitness if self.problem_type == "max" else
min(self.population, key=lambda ind: ind.fitness).fitness
history.append(best_fitness_in_gen)
return self.best_individual_overall, history
```

Classe DifferentialEvolution: Implementa o algoritmo de Evolução Diferencial (DE), eficaz para otimização contínua. O DE utiliza operadores de mutação baseados em diferenças vetoriais e um operador de cruzamento para gerar novas soluções.

Parâmetros principais:

- `D`: número de variáveis.
- `NP`: tamanho da população.
- `CR`, `F`: taxa de *crossover* e fator de diferenciação.
- Demais parâmetros similares ao AG.

```
class DifferentialEvolution:

def __init__(self, D, NP, CR, F, generations, lower_bound, upper_bound,
objective_function, problem_type):
self.D = D # Number of variables
self.NP = NP # Population size
self.CR = CR # Crossover rate
self.F = F # Differential weight
self.generations = generations
self.lower_bound = lower_bound
self.upper_bound = upper_bound
self.objective_function = objective_function
self.problem_type = problem_type
self.population = [] # Armazena os indivíduos da população.
self.best_individual_overall = None
self.best_fitness_overall = float('-inf') if problem_type == "max" else
float('inf')
```

Inicializa o algoritmo de Evolução Diferencial. Cria a população inicial de indivíduos com cromossomos gerados aleatoriamente.

```
def _initialize_population(self):
self.population = [Individual(self.D, self.lower_bound,
self.upper_bound) for _ in range(self.NP)]
self._evaluate_population()
```

Avalia o *fitness* de cada indivíduo na lista fornecida. Se nenhuma lista for fornecida, avalia a população principal.

```
def _evaluate_population(self, individuals=None):
if individuals is None:
individuals = self.population
for individual in individuals:
individual.fitness = self.objective_function(individual.chromosome)
```

Aplica os operadores de mutação e cruzamento para gerar um **vetor de teste (Vetor Ruído)** para um indivíduo alvo. 1. Seleciona três indivíduos aleatórios distintos (a, b, c) da população. 2. Aplica mutação para criar um vetor doador: v = a + F * (b - c). 3. Realiza cruzamento binomial entre o indivíduo alvo e o vetor doador para criar o vetor de teste.

```
def _mutate_and_crossover(self, target_idx):
target_individual = self.population[target_idx]
indices = list(range(self.NP))
indices.pop(target_idx)

if len(indices) < 3:
return target_individual.chromosome # Retorna uma cópia do cromossomo
do alvo para evitar erro.

a_idx, b_idx, c_idx = random.sample(indices, 3)
a = self.population[a_idx].chromosome
b = self.population[b_idx].chromosome
c = self.population[c_idx].chromosome

# Mutação: Calcula o vetor doador.
donor_vector = [a[j] + self.F * (b[j] - c[j]) for j in range(self.D)]

# Cruzamento (Crossover): Cria o vetor de teste.
trial_vector = list(target_individual.chromosome)
j_rand = random.randint(0, self.D - 1) # Garante que pelo menos uma dimensão
venha do vetor doador.

for j in range(self.D):
if random.random() < self.CR or j == j_rand:
trial_vector[j] = donor_vector[j]
```

```
# Garante que os valores dos genes estejam dentro dos limites definidos.
trial_vector[j] = max(self.lower_bound, min(self.upper_bound, trial_vector[j]))
return trial_vector
```

Executa o algoritmo de Evolução Diferencial por um **número especificado de gerações.** Em cada geração, cada indivíduo é submetido a mutação e cruzamento para criar um vetor de teste. O vetor de teste é então comparado com o indivíduo original, e o melhor é mantido para a próxima geração.

```
def run(self):
self._initialize_population()
history = []

for generation in range(self.generations):
new_population = []
for i in range(self.NP):
target_individual = self.population[i]
trial_chromosome = self._mutate_and_crossover(i)

trial_individual = Individual(self.D, self.lower_bound, self.upper_bound)
trial_individual.chromosome = trial_chromosome
self._evaluate_population(individuals=[trial_individual])

if self.problem_type == "min":
if trial_individual.fitness < target_individual.fitness:
new_population.append(trial_individual)
else:
new_population.append(target_individual)
else: # problem_type == "max"
if trial_individual.fitness > target_individual.fitness:
new_population.append(trial_individual)
else:
new_population.append(target_individual)
self.population = new_population

#Atualiza o melhor indivíduo geral encontrado após a formação da nova população.
current_best = max(self.population, key=lambda ind: ind.fitness) if self.problem_type == "max" else min(self.population, key=lambda ind: ind.fitness)
if self.best_individual_overall is None or \
```

```
(self.problem_type == "max" and current_best.fitness > 
self.best_fitness_overall) or \
(self.problem_type == "min" and current_best.fitness < 
self.best_fitness_overall):
self.best_individual_overall = Individual(self.D, self.lower_bound, 
self.upper_bound)
self.best_individual_overall.chromosome = list(current_best.chromosome)
self.best_individual_overall.fitness = current_best.fitness
self.best_fitness_overall = current_best.fitness

# Registra o melhor fitness da geração atual para o histórico.
best_fitness_in_gen = max(self.population, key=lambda ind: 
ind.fitness).fitness if self.problem_type == "max" else 
min(self.population, key=lambda ind: ind.fitness).fitness
history.append(best_fitness_in_gen)
return self.best_individual_overall, history
```

Classe Ant: Classe que define o comportamento de uma formiga individual da colônia.

- position: vetor de variáveis reais (a solução proposta pela formiga).
- fitness: valor da função objetivo para essa posição.

```
class Ant:

def __init__(self, n_variables, lower_bound, upper_bound):
self.position = [random.uniform(lower_bound, upper_bound) for _ in 
range(n_variables)]
self.fitness = None
```

Classe AntColonyOptimization: Representa a lógica do algoritmo ACO para otimização em espaço contínuo.

- *archive*: histórico das melhores soluções anteriores (formigas de elite).
- alpha, beta, rho: parâmetros típicos de ACO (feromônio, influência heurística e taxa de evaporação).
- objective_function e problem_type: função de avaliação e tipo de problema (min/max).

```
class AntColonyOptimization:

def __init__(self, n_ants, generations, alpha, beta, rho, n_variables, 
lower_bound, upper_bound, objective_function, problem_type):
self.n_ants = n_ants
self.generations = generations
self.alpha = alpha
self.beta = beta
self.rho = rho
self.n_variables = n_variables
```

```
self.lower_bound = lower_bound
self.upper_bound = upper_bound
self.objective_function = objective_function
self.problem_type = problem_type
self.archive = []
self.archive_limit = 50
```

Quando o *archive* ainda está vazio, gera uma solução totalmente aleatória. Posteriormente, a geração de novas soluções é baseada na distribuição normal (Gaussiana) centrada na média das melhores soluções já encontradas (intensificação). Essa implementação é típico do algoritmo *Ant Colony Optimization for Continuous Domains*.

```
def _generate_solution(self):
if not self.archive:
return [random.uniform(self.lower_bound, self.upper_bound) for _ in
range(self.n_variables)]
mean = np.mean([a.position for a in self.archive], axis=0)
std = np.std([a.position for a in self.archive], axis=0)
return [random.gauss(mu, s if s > 1e-5 else 0.1) for mu, s in zip(mean,
std)]
```

O método `run(self, snapshot_interval=14)` executa o algoritmo por um número fixo de gerações. A cada geração:

- Gera um conjunto de novas soluções (*`Ants`*) com base na média/variância do *`archive`*.
- Avalia todas as formigas e ordena pelo *fitness*.
- Atualiza o *`archive`* com as melhores.
- Armazena a melhor solução (para convergência e gráficos).
- Salva *snapshots* das populações para o gráfico de curva de nível.

```
def run(self, snapshot_interval=14):
best_ant = None
best_fitness = float('-inf') if self.problem_type == "max" else
float('inf')
history = []
snapshots = {}

for gen in range(self.generations):
```

```
ants = []
for _ in range(self.n_ants):
ant = Ant(self.n_variables, self.lower_bound, self.upper_bound)
ant.position = self._generate_solution()
ant.position = [min(max(x, self.lower_bound), self.upper_bound) for x
in ant.position]
ant.fitness = self.objective_function(ant.position)
ants.append(ant)

if self.problem_type == "min":
ants.sort(key=lambda x: x.fitness)
else:
ants.sort(key=lambda x: -x.fitness)

self.archive = ants[:self.archive_limit]

if (self.problem_type == "min" and ants[0].fitness < best_fitness) or \
(self.problem_type == "max" and ants[0].fitness > best_fitness):
best_ant = ants[0]
best_fitness = ants[0].fitness

history.append(best_fitness)

if gen % snapshot_interval == 0 or gen == self.generations - 1:
snapshots[gen] = ants

return best_ant, history, snapshots
```

O método `run_experiment_aco(...)` executa 30 execuções independentes do algoritmo ACO, o que permite:

- Obter **estatísticas de desempenho**: melhor, pior, média, desvio-padrão.
- Observar a **convergência média** da população.
- Os retornos são:
    - `best`, `worst`, `avg`, `std`: estatísticas de desempenho.
    - `mean_gen`, `std_gen`, `best_gen`, `worst_gen`: séries temporais (por geração).
    - `results`: lista com o melhor fitness de cada execução.
    - `snapshots_list`: populações salvas ao longo das execuções.

```
def run_experiment_aco(objective_function, problem_type, n_variables,
lower_bound, upper_bound,
n_ants=50, generations=100, alpha=1.0, beta=1.0, rho=0.1):
results = []
histories = []
snapshots_list = []

for i in range(30):
aco = AntColonyOptimization(n_ants, generations, alpha, beta, rho,
n_variables, lower_bound, upper_bound, objective_function,
problem_type)
best_solution, history, snapshots = aco.run()
results.append(best_solution.fitness)
histories.append(history)
snapshots_list.append(snapshots)
print(f"Execucao {i+1:02d} - Melhor: {best_solution.fitness:.6f}")

best = min(results) if problem_type == "min" else max(results)
worst = max(results) if problem_type == "min" else min(results)
avg = np.mean(results)
std = np.std(results)
histories = np.array(histories)

return best, worst, avg, std, np.mean(histories, axis=0),
np.std(histories, axis=0), \
np.min(histories, axis=0) if problem_type == "min" else
np.max(histories, axis=0), \
np.max(histories, axis=0) if problem_type == "min" else
np.min(histories, axis=0), \
results, snapshots_list
```

Função Quadrática: Função quadrática *(f(x) = sum(x_i^2))*. É uma função de teste simples e convexa, com mínimo global em x = [0, 0, ...] e valor 0. Para problemas de maximização com limites [-10, 10], o máximo é em x = [10, 10, ...] ou [-10, -10, ...] e valor 100 (para n=2).

```
def quadratic_function(x):
return sum(val**2 for val in x)
```

Função Rastrigin: É uma função de teste multimodal, com muitos mínimos locais, tornando a otimização mais desafiadora. O mínimo global está em x = [0, 0, ...] e tem valor 0.

```
def rastrigin_function(x, A=10):
```

```
n = len(x)
return A * n + sum(xi**2 - A * math.cos(2 * math.pi * xi) for xi in x)
```

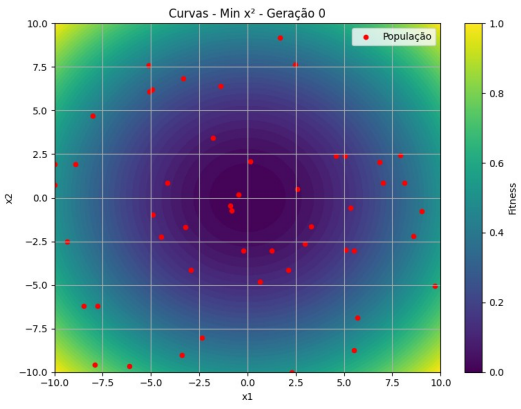
A solução apresentada encontra-se no seguinte repositório do GitHub: https://github.com/mihanne/algoritmos_geneticos.

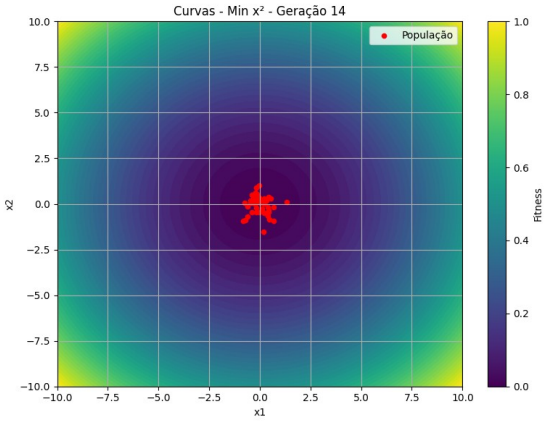
3- DISCUSSÃO DOS RESULTADOS

Foram executadas as 3 funções (quadrática de minimização e maximização e a Rastrigin) para cada Algoritmo (Genético, Diferencial e Colônia de Formigas). Cada instância foi executada **30 vezes** para apuração dos resultados. O ambiente de execução foi o Google Colab[1], executando Python 3, utilizando a CPU Padrão (13 GB de RAM).

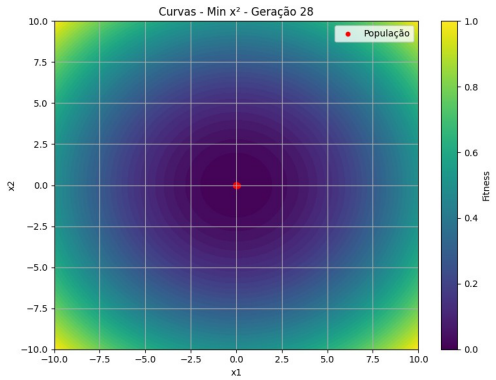
Foram plotadas 8 curvas de convergência a cada 14 execuções do Algoritmo Diferencial e Colônia de Formigas. A seguir, são mostradas as tendências de convergência da função quadrática de minimização, $Min\ f_1(x) = x^2$.

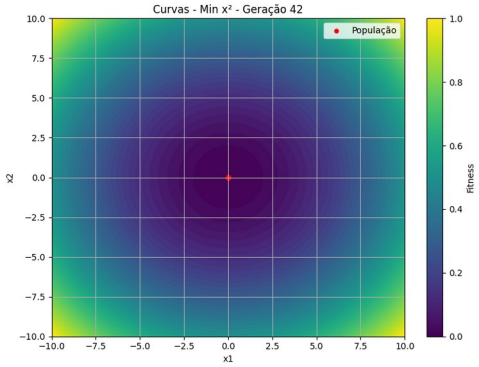
No Algoritmo Diferencial, percebe-se a convergência para o mínimo global.

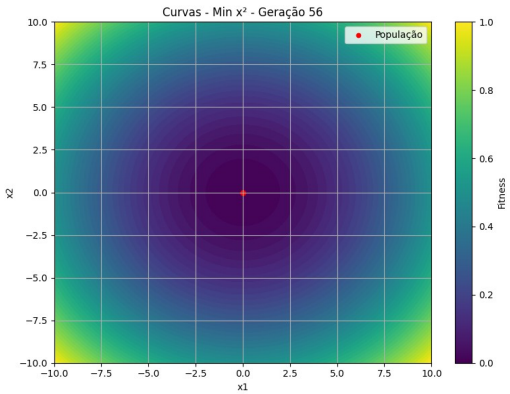


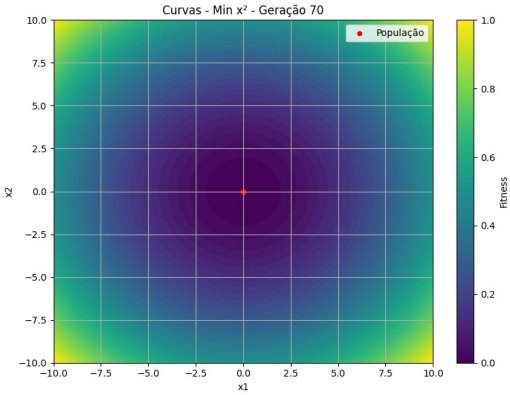


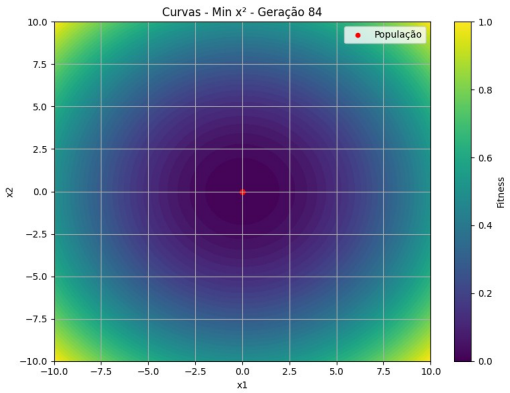
[1] - https://colab.research.google.com/

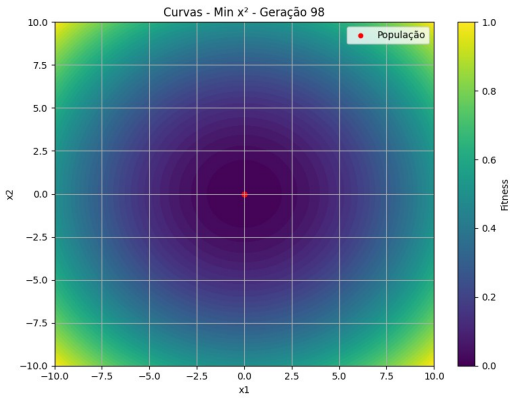




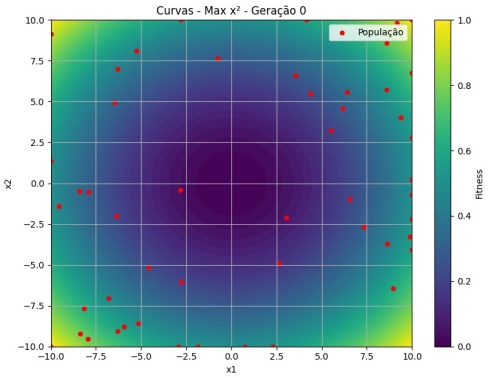


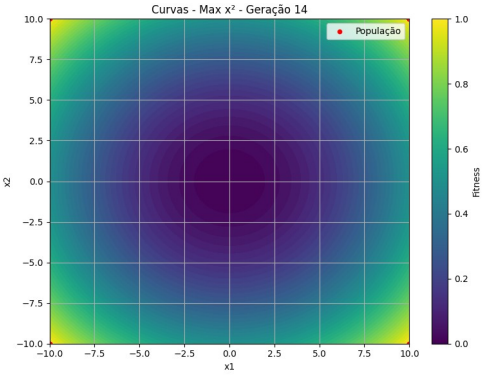


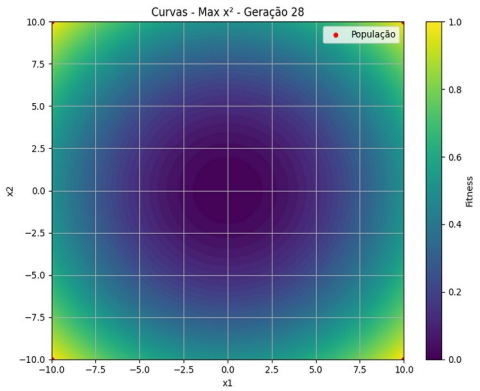


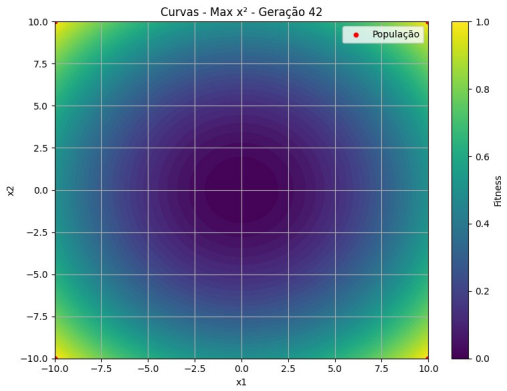


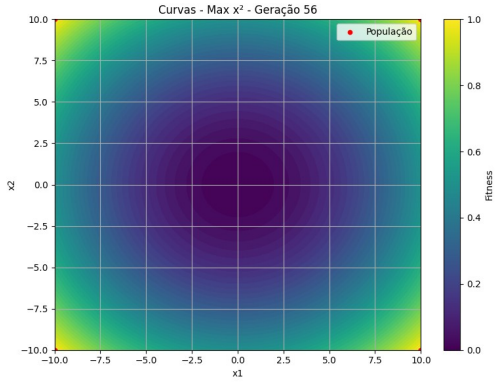
Para a função quadrática de maximização, *Max* $f_1(x) = x^2$, há tendência para que os indivíduos se desloquem para um dos quadrantes, atingindo a proximidade com a convergência. A seguir são apresentadas as curvas de nível para a maximização do Algoritmo Diferencial.

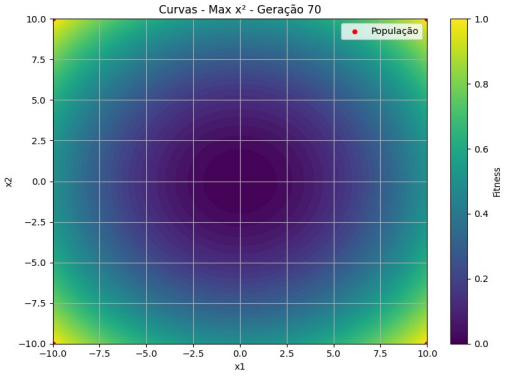

Curvas - Max x² - Geração 0
População
x1
x2
Fitness

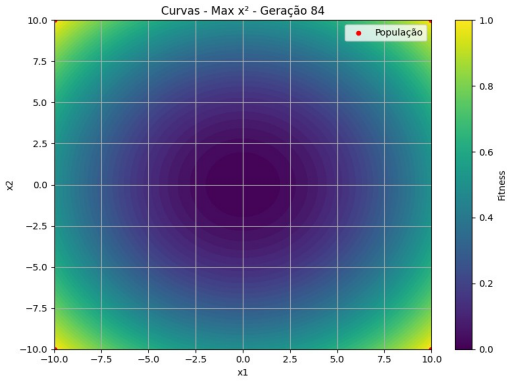

Curvas - Max x² - Geração 14
População
x1
x2
Fitness

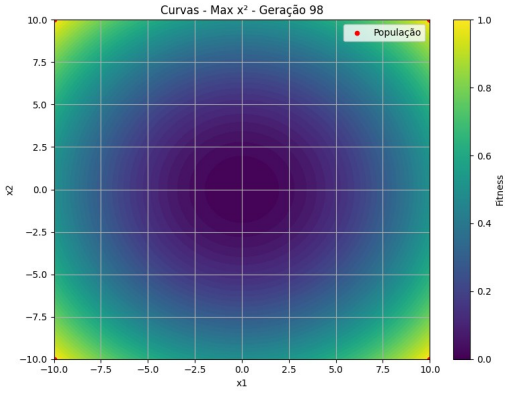

Curvas - Max x² - Geração 28
População
x1
x2
Fitness


Curvas - Max x² - Geração 42
População
x1
x2
Fitness

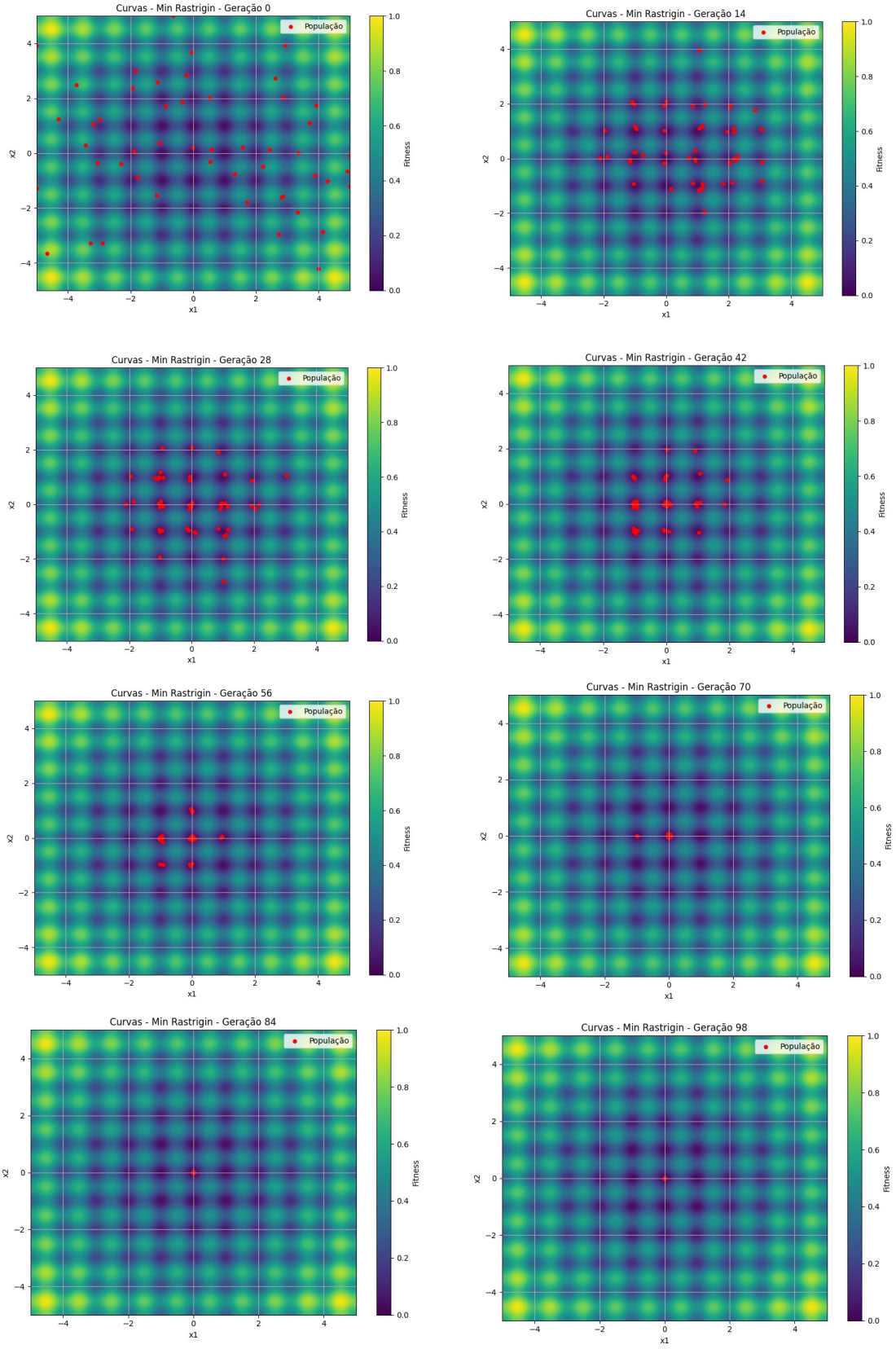

Curvas - Max x² - Geração 56
População
x1
x2
Fitness


Curvas - Max x² - Geração 70
População
x1
x2
Fitness

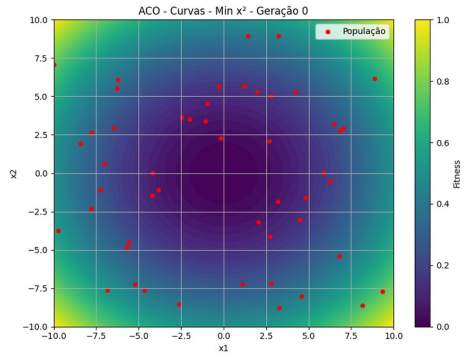

Curvas - Max x² - Geração 84
População
x1
x2
Fitness

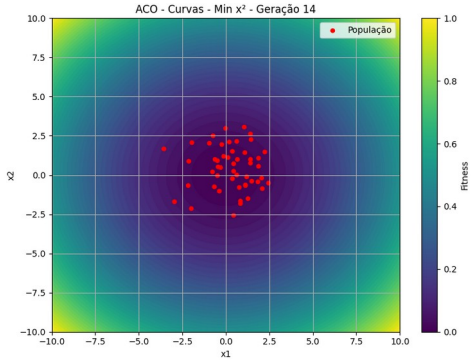

Curvas - Max x² - Geração 98
População
x1
x2
Fitness

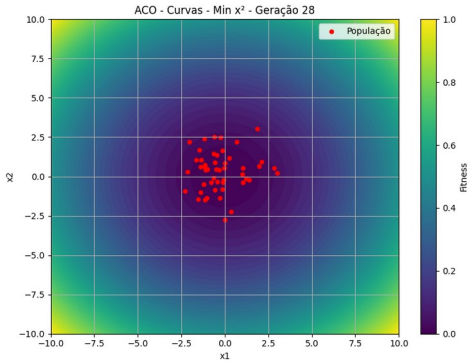
Para a função Rastringin de minimização, $Min\ f_2(x) = An + \sum_{i}^{n}=0[x_i^2 - A\cos(2\pi x_i)]$, há tendência para que os indivíduos se desloquem para o centro do gráfico, nas curvas de nível do Algoritmo Diferencial.

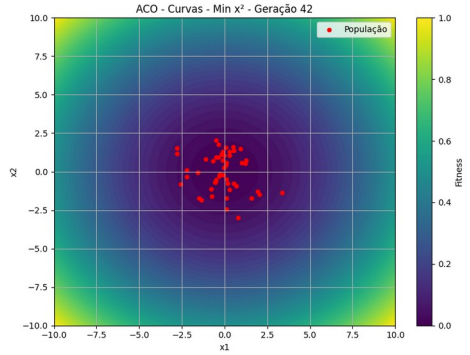


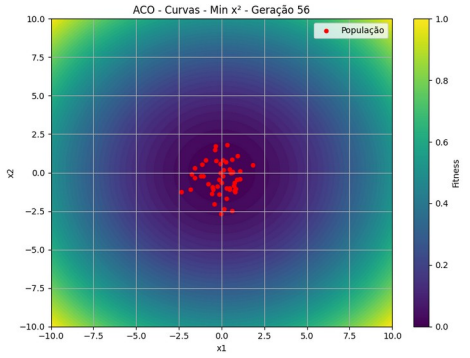
Já na execução do Algoritmo de Colônia de Formigas, a função quadrática de minimização, $Min\ f_1(x) = x^2$, também mostrou tendência para a convergência do mínimo. Porém, percebe-se que nem toda a população convergiu (Geração 98).

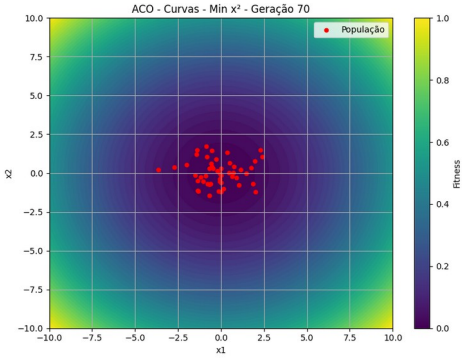


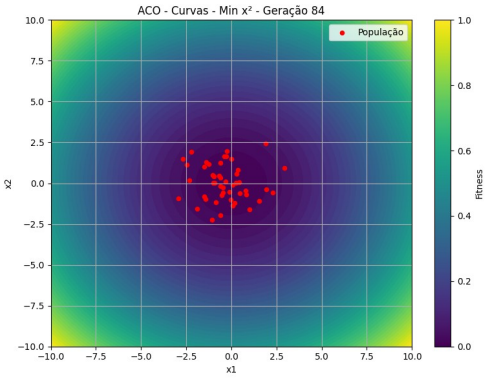


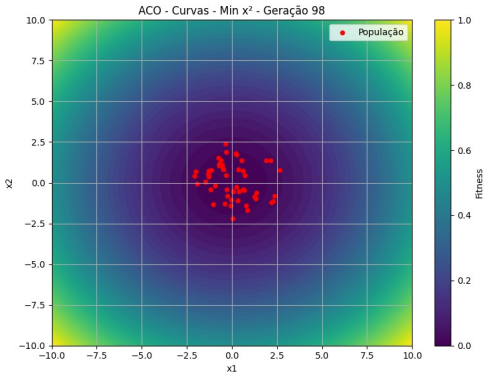




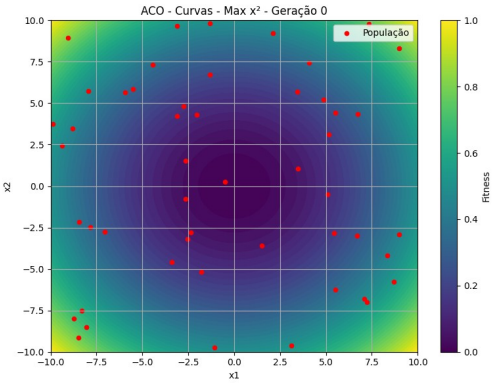


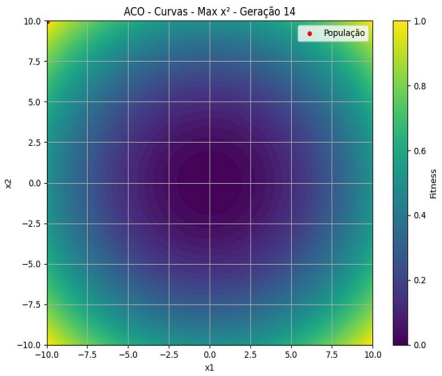


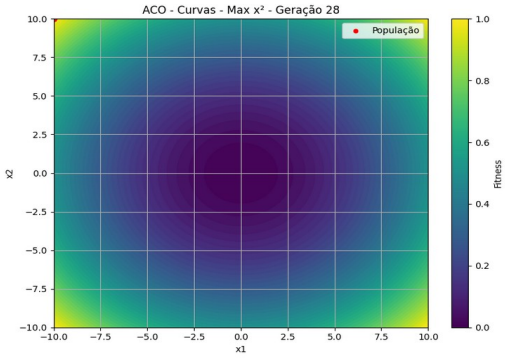


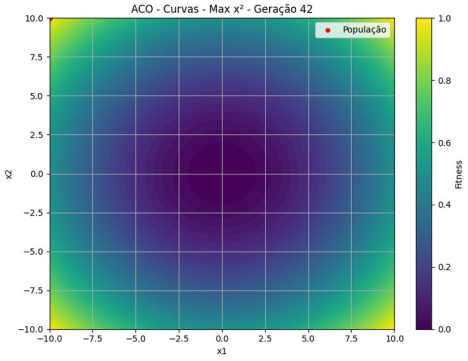


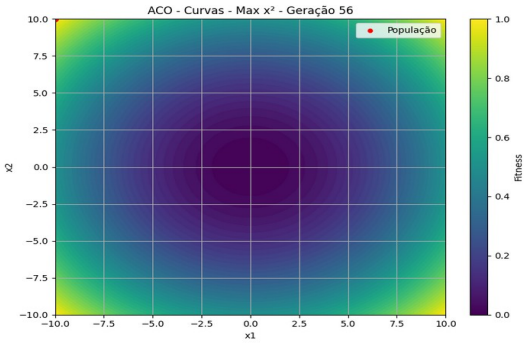
Já na função quadrática de maximização, *Max* $f_1(x) = x^2$, houve tendência de convergência para os pontos extremos do Algoritmo Colônia de Formigas.

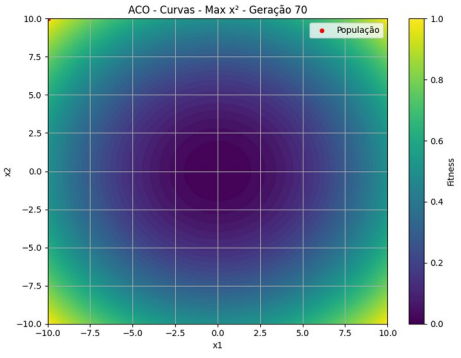


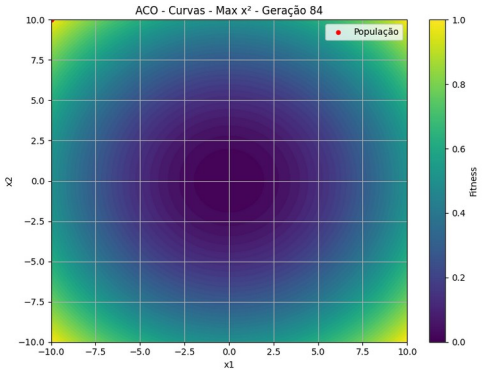


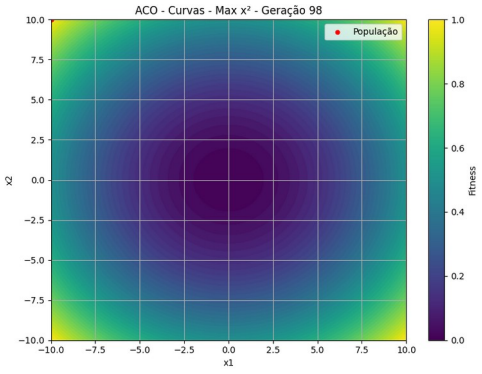




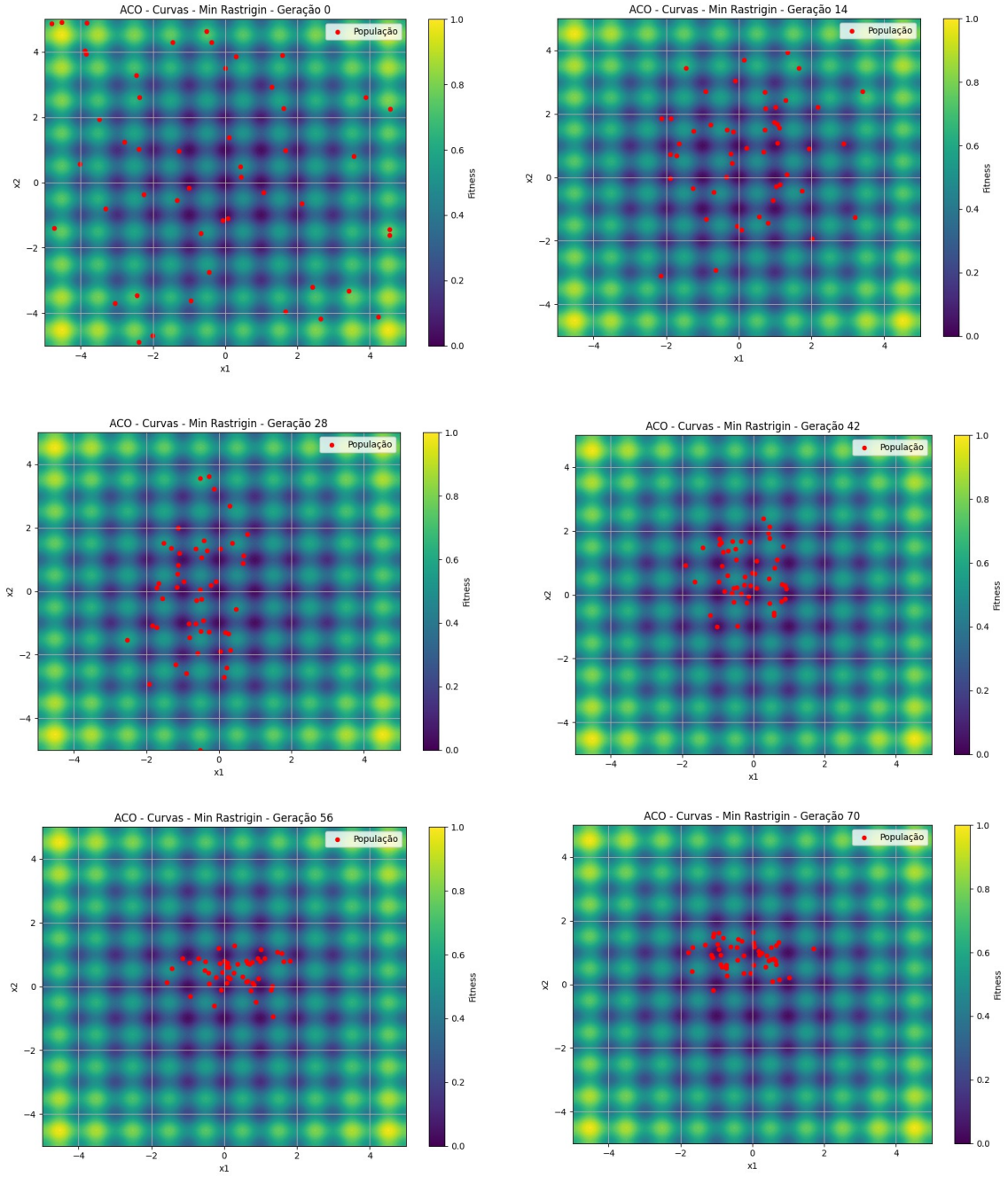


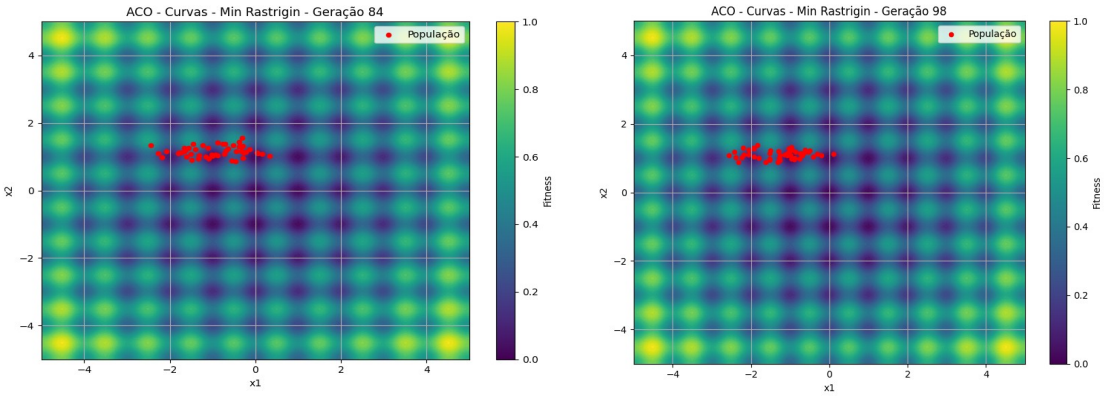



ACO - Curvas - Max x² - Geração 84
População
x1
x2
Fitness


ACO - Curvas - Max x² - Geração 98
População
x1
x2
Fitness

No Algoritmo Colônia de Formigas, a função Rastringin de minimização, $Min\ f_2(x) = An + \sum_i^n=0[x_i^2 - A\cos(2\pi x_i)]$, há lenta tendência para que os indivíduos se desloquem para o centro do gráfico. Percebe-se que na geração 98 as “formigas” estão se deslocando para o centro, porém, ainda não atingiram.





Para a realização da comparação entre os dois Algoritmos (Genético, Diferencial e Colônia de Formigas), optou-se por executar a partir da mesma população inicial os testes, com o mesmo número de execuções (30) e de avaliações. Visando um olhar crítico, foram realizados **3 Testes com 30 execuções cada**. Em cada execução foi mantido o histórico e exibido o melhor resultado das iterações das gerações.

Percebe-se que há variação entre o melhor indivíduo de cada execução, principalmente entre o Algoritmo Genético nos testes das funções quadráticas (*Min* e *Max*) e na Rastrigin. O Algoritmo Colônia de Formigas, mostrou bom resultado na função de maximização. As tabelas a seguir mostram resultados das execuções.

1º Teste com 30 execuções

Tipo de Algoritmo	Função	Melhor	Pior	Média	Desvio Padrão
GA	$Min\ f_1(x) = x^2$	0.000007	0.045597	0.009889	0.013508
DE	$Min\ f_1(x) = x^2$	0.000000	0.000000	0.000000	0.000000
ACO	$Min\ f_1(x) = x^2$	0.000031	0.083291	0.011285	0.017090
GA	$Max\ f_1(x) = x^2$	199.207084	184.342699	194.664916	3.846131
DE	$Max\ f_1(x) = x^2$	200.000000	200.000000	200.000000	0.000000
ACO	$Max\ f_1(x) = x^2$	200.000000	200.000000	200.000000	0.000000
GA	Rastrigin	0.000003	1.048474	0.224991	0.297955
DE	Rastrigin	0.000000	0.000000	0.000000	0.000000
ACO	Rastrigin	0.004816	1.306397	0.420185	0.432844

Já a função Rastrigin no Algoritmo Colônia de Formigas teve piores resultados acima de 1 nos testes realizados. Enquanto que o Algoritmo Diferencial a função Rastrigin atingiu na maioria dos testes os valores melhor, pior e média igual a 0.

2º Teste com 30 execuções

Tipo de Algoritmo	Função	Melhor	Pior	Média	Desvio Padrão
GA	$Min\ f_1(x) = x^2$	0.000001	0.111996	0.010475	0.021005
DE	$Min\ f_1(x) = x^2$	0.000000	0.000000	0.000000	0.000000
ACO	$Min\ f_1(x) = x^2$	0.000004	0.002599	0.000493	0.000574
GA	$Max\ f_1(x) = x^2$	199.790970	187.253610	194.959857	3.052084
DE	$Max\ f_1(x) = x^2$	200.000000	200.000000	200.000000	0.000000
ACO	$Max\ f_1(x) = x^2$	200.000000	200.000000	200.000000	0.000000
GA	Rastrigin	0.000055	1.106831	0.249869	0.316016
DE	Rastrigin	0.000000	0.000007	0.000000	0.000001
ACO	Rastrigin	0.001263	0.996101	0.207767	0.237861

De modo geral, o comportamento do Algoritmo Genético teve pior resultado em comparação com o Algoritmo Diferencial em todos as funções, quadrática e Rastrigin.

3º Teste com 30 execuções

Tipo de Algoritmo	Função	Melhor	Pior	Média	Desvio Padrão
GA	$Min\ f_1(x) = x^2$	0.000004	0.070516	0.012095	0.016175
DE	$Min\ f_1(x) = x^2$	0.000000	0.000000	0.000000	0.000000
ACO	$Min\ f_1(x) = x^2$	0.000013	0.002888	0.000711	0.000664
GA	$Max\ f_1(x) = x^2$	198.813934	186.838045	195.230451	2.605209
DE	$Max\ f_1(x) = x^2$	200.000000	200.000000	200.000000	0.000000

ACO	$Max\ f_1\ (x) = x^2$	200.000000	200.000000	200.000000	0.000000
GA	Rastrigin	0.000018	0.914972	0.151581	0.231012
DE	Rastrigin	0.000000	0.000007	0.000000	0.000000
ACO	Rastrigin	0.002259	0.870350	0.229367	0.226661

Figura de execução do **1° Teste** na função quadrática, $Min\ f_1\ (x) = x^2$ no **Algoritmo Genético**.

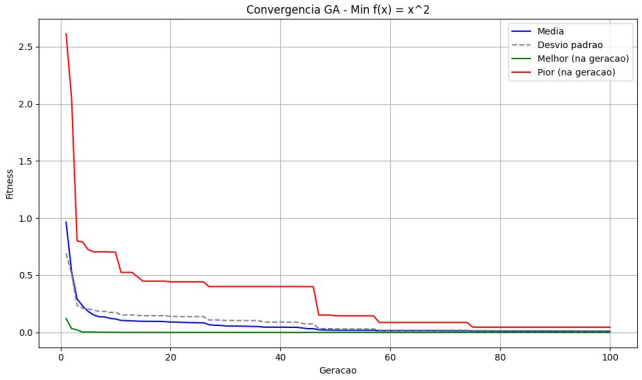


Figura de execução do **1° Teste na** execução da função quadrática, $Min\ f_1\ (x) = x^2$ no **Algoritmo Diferencial.**

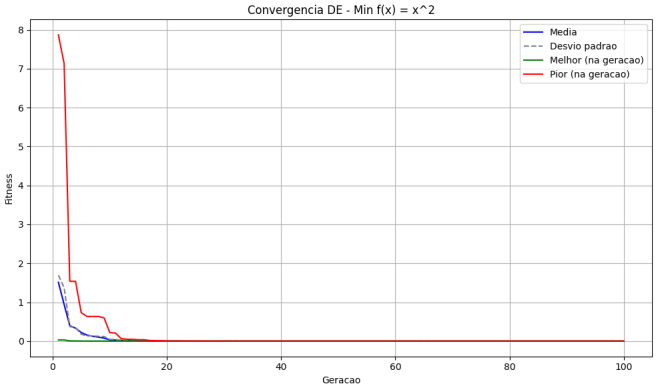


Figura de execução do **1° Teste na** execução da função quadrática, $Min\ f_1\ (x) = x^2$ no **Algoritmo Colônia de Formigas.**

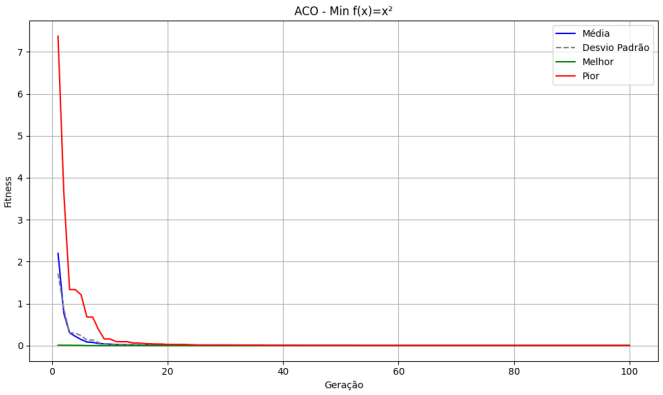


Figura de execução do **1° Teste na** execução da função quadrática, *Max* $f_1(x) = x^2$ no **Algoritmo Genético.**

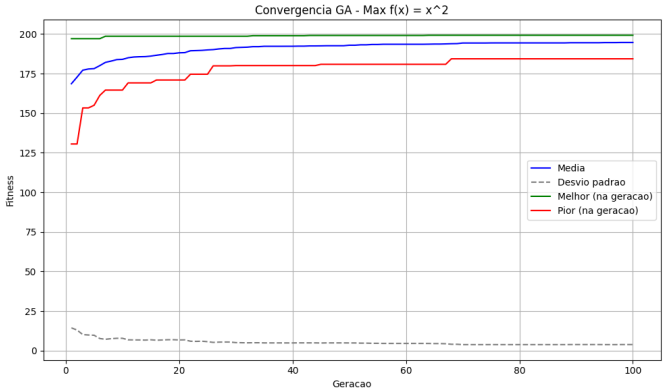


Figura de execução do **1° Teste na** execução da função quadrática, *Max* $f_1(x) = x^2$ no **Algoritmo Diferencial.**

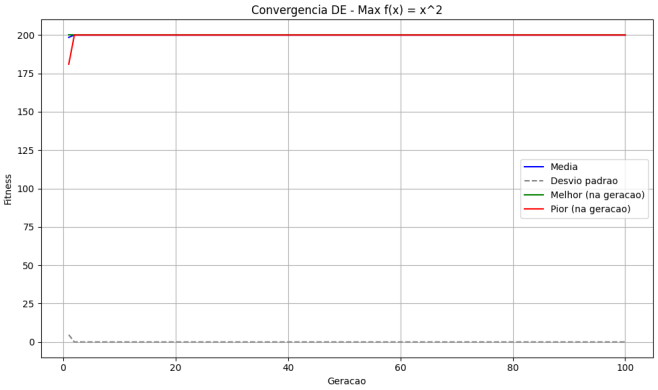


Figura de execução do **1° Teste na** execução da função quadrática, *Max* $f_1(x) = x^2$ no **Algoritmo Colônia de Formigas.**

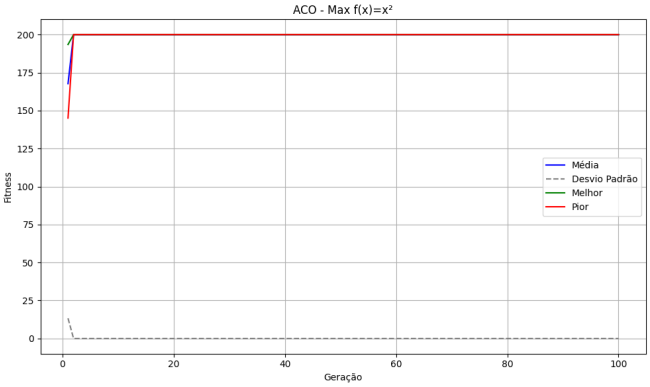


Figura de execução do **1° Teste na** execução da função Rastrigin, $Min\ f_2(x) = An + \sum_i^n=0[x_i^2 - A\cos(2\pi x_i)]$ no **Algoritmo Genético.**

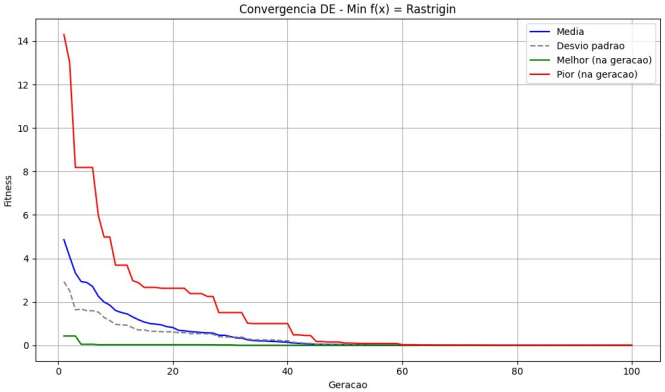


Figura de execução do **1° Teste na** execução da função Rastrigin, $Min\ f_2(x) = An + \sum_i^n=0[x_i^2 - A\cos(2\pi x_i)]$ no **Algoritmo Diferencial.**

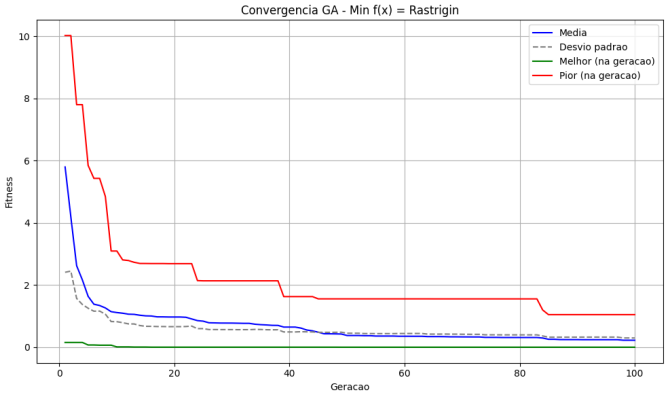
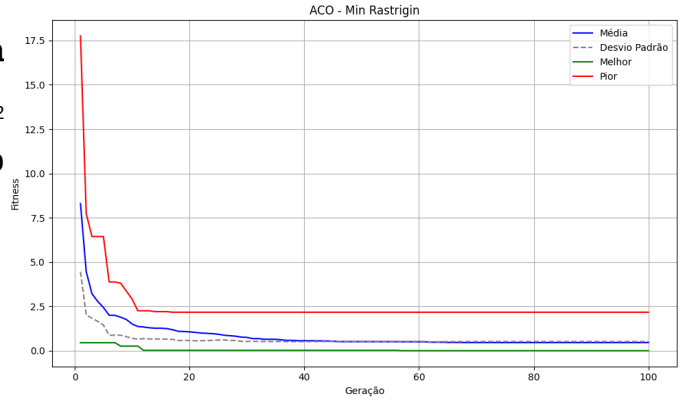


Figura de execução do **1° Teste na** execução da função Rastrigin, $Min\ f_2(x) = An + \sum_i^n=0[x_i^2 - A\cos(2\pi x_i)]$ no **Algoritmo Colônia de Formigas.**



Realizou-se a contagem de acertos de cada teste realizado nas 30 execuções. Foi adotado como parâmetro a tolerância de 1e-3. As análises revelaram que o Algoritmo Diferencial obteve 100% de acerto em todos os testes. Já o Algoritmo Colônia de Formigas obteve sucesso na função de maximização.

1º Teste com 30 execuções

Função	Algoritmo Genético	Algoritmo Diferencial	Algoritmo de Colônia de Formigas
$Min\ f_1(x) = x^2$	10/30	30/30	26/30
Max $f_1(x) = x^2$	0/30	30/30	30/30
$Min\ f_2(x) = An + \sum_i^n=0[x_i^2 - A\cos(2x_i)]$	4/30	30/30	0/30

2º Teste com 30 execuções

Função	Algoritmo Genético	Algoritmo Diferencial	Algoritmo de Colônia de Formigas
$Min\ f_1(x) = x^2$	22/30	30/30	22/30
Max $f_1(x) = x^2$	0/30	30/30	30/30
$Min\ f_2(x) = An + \sum_i^n=0[x_i^2 - A\cos(2\pi x_i)]$	5/30	30/30	0/30

3º Teste com 30 execuções

Função	Algoritmo Genético	Algoritmo Diferencial	Algoritmo de Colônia de Formigas
$Min\ f_1(x) = x^2$	20/30	30/30	26/30
Max $f_1(x) = x^2$	0/30	30/30	30/30
$Min\ f_2(x) = An + \sum_i^n=0[x_i^2 - A\cos(2\pi x_i)]$	10/30	30/30	1/30

4- CONCLUSÃO

O relatório apresenta um estudo comparativo entre o Algoritmo Genético (AG), Algoritmo de Evolução Diferencial (DE) e Algoritmo Colônia de Formigas (ACO) na resolução de problemas de otimização envolvendo uma função quadrática e a função Rastrigin, ambas com duas variáveis. Os testes foram realizados com as mesmas condições iniciais e parâmetros, totalizando 30 execuções por algoritmo em cada caso. Os resultados demonstram uma clara superioridade do DE, que alcançou soluções ótimas com precisão (erro ≤ 1e-3) em 100% das execuções em todos os testes. O AG apresentou maior variabilidade nos resultados e acurácia significativamente inferior, especialmente nas tarefas de maximização da função quadrática e na função Rastrigin. Enquanto que o ACO foi eficiente na função de maximização, e demostrou resultados inferiores nas funções de minimização (quadrática e Rastrigin).

Desse modo, os resultados alcançados evidenciam a maior robustez e eficiência do DE em contextos de otimização contínua.

5- REFERÊNCIAS

CHAKRABORTY, Uday. **Advances in Differential Evolution**. Springer, 2008.

DORIGO, Marco; MANIEZZO, Vittorio; COLORNI, Alberto. **Ant System: an autocatalytic optimizing process**. *Technical Report 91-016*, Dipartimento di Elettronica, Politecnico di Milano, 1991.

SOCHA, Krzysztof; DORIGO, Marco. **Ant colony optimization for continuous domains**. *European Journal of Operational Research*, Amsterdam, v. 185, n. 3, p. 1155–1173, 2008. Disponível em: https://www.sciencedirect.com/science/article/abs/pii/S0377221706006333. Acesso em: 14 jul. 2025.

STORNI, R.; PRICE, K. Differential Evolution - **A Simple and Efficient Adaptive Scheme for Global Optimization Over Continuous Spaces**. ICSI Technical Report TR-95-012, março de 1995.