UNIVERSIDADE FEDERAL DO RIO DE JANEIRO

INSTITUTO DE COMPUTAÇÃO

CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

JOÃO PEDRO LOPES MURTINHO

# OTIMIZAÇÃO POR ENXAME DE PARTÍCULAS E COLÔNIA DE FORMIGAS PARA DOMÍNIOS CONTÍNUOS

RIO DE JANEIRO

2023

JOÃO PEDRO LOPES MURTINHO

# OTIMIZAÇÃO POR ENXAME DE PARTÍCULAS E COLÔNIA DE FORMIGAS PARA DOMÍNIOS CONTÍNUOS

Trabalho de conclusão de curso de graduação apresentado ao Instituto de Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. João Carlos Pereira da Silva

RIO DE JANEIRO

2023

CIP - Catalogação na Publicação

M984o

Murtinho, João Pedro
Otimização por enxame de particulas e colônia de formigas para domínios continuo / João Pedro Murtinho. -- Rio de Janeiro, 2024.
48 f.

Orientador: João Carlos da Silva.
Trabalho de conclusão de curso (graduação) - Universidade Federal do Rio de Janeiro, Instituto de Computação, Bacharel em Ciência da Computação, 2024.

1. Algoritmos bioinspirados. 2. Colônia de formigas. 3. Particulas de enxame. 4. Otimização de funções de domínio continuo. I. da Silva, João Carlos, orient. II. Titulo.

Elaborado pelo Sistema de Geração Automática da UFRJ com os dados fornecidos pelo(a) autor(a), sob a responsabilidade de Miguel Romeu Amorim Neto - CRB-7/6283.

JOÃO PEDRO LOPES MURTINHO

# OTIMIZAÇÃO POR ENXAME DE PARTÍCULAS E COLÔNIA DE FORMIGAS PARA DOMÍNIOS CONTÍNUOS

Trabalho de conclusão de curso de graduação apresentado ao Instituto de Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Aprovado em 13 de dezembro de 2023.

BANCA EXAMINADORA:

Documento assinado digitalmente
gov.br JOAO CARLOS PEREIRA DA SILVA
Data: 06/03/2024 17:45:03-0300
Verifique em https://validar.iti.gov.br

João Carlos Pereira da Silva D.Sc. - (UFRJ)

Documento assinado digitalmente
gov.br CAROLINA GIL MARCELINO
Data: 06/03/2024 19:18:45-0300
Verifique em https://validar.iti.gov.br

Carolina Gil Marcelino D.Sc. - (UFRJ)

Documento assinado digitalmente
gov.br LUZIANE FERREIRA DE MENDONCA
Data: 06/03/2024 19:59:26-0300
Verifique em https://validar.iti.gov.br

Luziane Ferreira de Mendonça D.Sc. - (UFRJ)

## AGRADECIMENTOS

Gostaria de principalmente agradecer minha mãe e meu pai pelas oportunidades, amor e apoio incondicional durante todos os anos. Palavras não comportam minha gratidão. Agradecimentos especiais também a meus amigos de dentro e fora da universidade que me mantiveram são durante os longos anos de graduação, não estaria aqui sem o apoio de vocês. Grato também ao professor João Carlos por sua paciência e atenciosidade, demonstrada diversas vezes ao longo do meu tempo de curso e repetidamente reforçada ao longo de seu período como orientador deste trabalho.

## RESUMO

O trabalho busca apresentar e explorar o impacto de diferentes parametrizações de dois populares algoritmos baseados em meta-heurísticas bioinspiradas para problemas de otimização com funções de domínios contínuos: o algoritmo de otimização por enxame de partículas (PSO) e uma adaptação do algoritmo padrão de otimização de colônia de formigas para funções de domínios discretos voltado para o caso contínuo ($ACO_R$). São introduzidas as implementações dos dois algoritmos e suas execuções são a seguir exemplificadas com visualizações e instruções para reprodução fazendo uso do código desenvolvido. Em sequência, uma exploração da influência de diferentes valores para cada parâmetro dos dois algoritmos apresentados é realizada, com o objetivo de se obter uma compreensão generalizada dos impactos observados. Nessa exploração, e dado o objetivo de buscar um entendimento do papel funcional dos diferentes parâmetros de forma agnóstica à função que o algoritmo seja aplicado, diferentes valores de parâmetros são considerados preliminarmente em um contexto anterior à uma busca empírica aplicada a uma função específica a ser otimizada. Devido tanto à forte interdependência entre os próprios parâmetros e também entre os parâmetros e a função objetivo que o algoritmo esteja aplicado, constata-se ser difícil obter aproximações adequadas para valores ótimos gerais dos diferentes parâmetros dos algoritmos. Apesar disso, a exploração realizada acaba oferecendo um aprofundamento valioso na compreensão da influência dos diferentes conceitos das meta-heurísticas descritos por cada um dos parâmetros na aplicação dos dois algoritmos.



**Palavras-chave:** otimização; algoritmos bioinspirados; partículas de enxame; colônia de formigas; domínio discreto; domínio contínuo.

## ABSTRACT

This study aims to introduce and explore the impact of different parameterizations of two popular metaheuristic algorithms for optimization problems in continuous domain functions: the Particle Swarm Optimization (PSO) algorithm and an adaptation of the standard Ant Colony Optimization algorithm for discrete domain functions tailored to the continuous case ($ACO_R$). The implementations of the two algorithms are introduced, and their executions are exemplified with visualizations and instructions for reproduction using the developed code. Subsequently, an exploration of the influence of different values for each parameter of the two presented algorithms is carried out, aiming to obtain a generalized understanding of the observed impacts. In this exploration, given the goal of seeking an understanding of the functional role of the different parameters regardless of the function to which the algorithm is applied, different parameter values are considered in a context prior to the actual empirical search applied to a specific function to be optimized. Due to both the strong interdependence among the parameters themselves and also the dependence between the parameters and the objective function to which the algorithm is applied, it is found difficult to obtain adequate approximations for general optimal values of the different parameters of the algorithms. Nevertheless, the conducted study ultimately provides valuable insight into understanding the influence of the different concepts of metaheuristics described by each of the parameters in the application of the two algorithms.



**Keywords:** optimization; bioinspired algorithms; swarm particles; ant colony; discrete domain; continuous domain.

## LISTA DE ILUSTRAÇÕES

## LISTA DE SIGLAS

PSO – Particle Swarm Optimization (otimização por enxame de partículas)

ACO – Ant Colony Optimization (otimização de colônia de formiga)

$ACO_R$ – Ant Colony Optimization for continuous domain (otimização de colônia de formiga para domínios contínuos)

**SUMÁRIO**

# 1 INTRODUÇÃO

Problemas de otimização se mostram presentes nas mais diversas áreas onde uma análise quantitativa é empregada. Exemplos relevantes incluem as áreas de logística (CAUNHYE; NIE; POKHAREL, 2012), economia (INTRILIGATOR, 2002) e engenharias no geral (MARLER; ARORA, 2004). A ubiquidade de problemas de otimização nas mais diversas áreas fica clara ao notar sua recorrente presença no desenvolvimento científico na história, com sua genealogia alcançando até os tempos de Euclides, em seu tratamento da distância mínima entre reta e ponto no espaço. O desenvolvimento de métodos numéricos computacionais, impulsionado pelos avanços tecnológicos do século XX que permitiram sua aplicabilidade, trouxeram inovações sem precedentes na capacidade de solucionar problemas dessa classe. Métodos de gradiente (AMARI, 1993) e programação linear (DANTZIG, 2002) são exemplos de técnicas que se destacaram nesse contexto.

Apesar dos avanços, muitos dos métodos propostos possuíam limitações graves que impediam sua aplicação para problemas complexos de otimização. Tomando como exemplo os dois métodos citados anteriormente, observa-se que o método de gradiente requer uma função diferenciável por toda região do espaço de busca, já métodos de programação linear demandam que tanto a função objetivo quanto suas restrições sejam lineares para sua aplicação.

A dificuldade encontrada na busca por soluções eficazes, eficientes e generalizáveis para uma ampla gama de problemas de otimização complexos levou pesquisadores a explorarem abordagens inovadoras. Essa exploração voltou a atenção deles para fenômenos naturais biológicos. Esse movimento por parte dos pesquisadores pode ter sua origem traçada até o influente artigo *Computing machinery and intelligence* (TURING, 2009) de Alan Turing, publicado originalmente em 1950, onde o autor sugere um processo evolutivo de aprendizagem para o desenvolvimento de inteligências artificiais. Nas décadas seguintes, em especial devido a popularização de algoritmos genéticos (HOLLAND, 1992) nos anos 70, com sua teoria inspirada no mecanismo de seleção natural Darwiniano, outras soluções com inspirações similares começaram a ser propostas.

Denominados algoritmos bioinspirados (KAR, 2016), essa classe de soluções que vieram a surgir a partir dos anos 90 substituíram uma inspiração por processos biológicos, como no caso do algoritmo genético, por uma inspiração no comportamento coletivo de populações animais na natureza. Os algoritmos dessa classe são projetados para resolver problemas de forma análoga a como diferentes seres vivos lidam coletivamente com os desafios de seu ambiente de maneira cooperativa. Duas dessas soluções mais notáveis incluem os algoritmos de Otimização de Colônia de Formigas (ACO, sigla em inglês) e de Otimização por Enxame de Partículas (PSO, sigla novamente em inglês).

Este trabalho possui como objetivo realizar uma introdução a algoritmos de meta-heurística bioinspiradas, apresentando as implementações dos algoritmos PSO, ACO e $ACO_R$ (uma adaptação do algoritmo ACO original para sua aplicação em problemas de domínios contínuos). Além da apresentação dos algoritmos listados, também será realizada uma exploração de diferentes parametrizações do PSO e $ACO_R$, buscando uma compreensão das diferentes influências dos parâmetros de forma agnóstica à função a ser otimizada, com o objetivo de se obter um entendimento generalizável sobre o impacto de diferentes valores para os mesmos. Essa compreensão será desenvolvida analisando individualmente cada um dos parâmetros dos dois algoritmos, observando como valores diferentes afetam o processo de otimização contido na descrição do algoritmo em questão.

A apresentação dos três algoritmos mencionados será realizada a seguir no capítulo 2. No capítulo 3, serão apresentados dois exemplos práticos de execução dos algoritmos PSO e

$ACO_R$ fazendo uso do código desenvolvido para este trabalho. Em sequência, será realizada a exploração da influência dos diferentes parâmetros dos algoritmos PSO e $ACO_R$ no capítulo 4. Por fim, no capítulo 5, o trabalho é concluído dando continuidade às observações obtidas no capítulo anterior e listando possíveis trabalhos futuros a serem realizados.

O código, explorado no capítulo 3, contendo as implementações e visualizações dos algoritmos PSO e $ACO_R$ apresentados ao longo do trabalho, pode ser encontrado no repositório público do trabalho[1], com suas instruções de uso e instalação.

[1] Repositório disponível em: https://github.com/VideoJope/AntColony-and-ParticleSwarm-Optimization

# 2 ALGORITMOS BIOINSPIRADOS

Neste capítulo serão introduzidas três diferentes implementações de algoritmos bioinspirados para otimização. Primeiramente, o algoritmo de Otimização de Enxame de Partículas (PSO) será apresentado. Em seguida, o algoritmo de Otimização de Colônia de Formigas padrão (ACO) será introduzido, seguido de sua adaptação para problemas de domínio contínuo ($ACO_R$).

## 2.1 OTIMIZAÇÃO DE ENXAME DE PARTÍCULAS

Um dos algoritmos bioinspirados de maior notoriedade, ocasionada por sua ampla gama de aplicações, é o algoritmo de Otimização por Enxame de Partículas (KENNEDY; EBERHART, 1995). Esse algoritmo foi inspirado no comportamento cooperativo de enxame de certos animais — como abelhas e algumas espécies de pássaros — que trabalham em conjunto para realizar tarefas como buscar alimento, construir ninhos e afastar predadores. A Otimização por Enxame de Partículas é utilizada para resolver problemas com domínios contínuos, como por exemplo buscar o mínimo ou máximo global de uma função real n-dimensional.

O PSO — assim como os demais algoritmos que serão apresentados neste capítulo — se trata de um algoritmo de otimização baseado em meta-heurística, um conjunto de técnicas com bases conceituais de alto nível (no caso do PSO, este sendo sua inspiração no comportamento de enxames) independentes de uma implementação específica que se preocupam em buscar soluções aproximadamente ótimas para problemas NP-difíceis, onde encontrar uma das soluções verdadeiramente ótimas deterministicamente se demonstra computacionalmente inviável.

Similarmente ao algoritmo de Otimização de Colônia de Formigas (ACO), apresentado a fundo na seção 2.2 deste capítulo, cada partícula do enxame (de forma análoga a cada formiga no ACO) descreve uma solução candidata parcial ao problema em questão. Essa solução será aprimorada a cada iteração do algoritmo, convergindo eventualmente para a solução ótima final do problema. Essa solução, descrita por cada partícula, se dará na forma de um vetor de posição n-dimensional, descrevendo onde a partícula se encontra no espaço de soluções possíveis do problema modelado na iteração corrente. As diferentes partículas a cada iteração, sabendo sua coordenada de posição atual e seu respectivo valor quando aplicada na função objetivo a ser otimizada, irão se comunicar com partículas vizinhas, visando mover-se dentro do espaço de soluções em direção à solução ótima parcial obtida até então.

Além de sua posição atual, cada partícula no algoritmo PSO também possui um fator descrevendo sua velocidade no espaço de busca. A velocidade, como o nome sugere, é um vetor indicando a direção e o quão longa foi a distância percorrida pela partícula entre a última iteração e a corrente.

### 2.1.1 Descrição da Execução do PSO

Para efetivamente dar início a aplicação do algoritmo PSO padrão, será necessário antes realizar algumas decisões de implementação. Estas escolhas possuem grande impacto na capacidade de convergência do algoritmo e na qualidade da solução final encontrada, sendo

também altamente dependente da função específica que está a ser otimizada no problema. Nesse momento então serão definidos os seguintes hiperparâmetros:

- Número total de partículas.
- Fator de inércia, definindo a influência do vetor velocidade corrente sobre a atualização da coordenada da partícula.
- Coeficiente cognitivo e coeficiente social, indicando respectivamente a influência na atualização da posição pela melhor posição encontrada pela própria partícula e a melhor posição encontrada por seus vizinhos.
- Posições iniciais de cada partícula.
- Topologia do enxame, isto é, a definição do conjunto de partículas consideradas vizinhas durante a etapa de troca de informações.

Após inicializar cada partícula com uma velocidade inicial igual a 0, suas respectivas posições, definir os parâmetros listados acima e estabelecer a condição de parada, podemos dar início a execução do algoritmo. A seguir se encontra uma descrição do procedimento que será repetido para cada partícula do sistema por iteração. Considerando a i-ésima iteração do algoritmo, as seguintes operações serão efetuadas:

1. É computado o novo vetor velocidade, levando em consideração os seguintes fatores:
    a. A posição da partícula na iteração anterior: $\boldsymbol{p_{i-1}}$
    b. A velocidade da partícula na iteração anterior: $\boldsymbol{v_{i-1}}$
    c. A melhor posição já visitada pela partícula até então: $\boldsymbol{p_{best\_i}}$
    d. A melhor posição já visitada por seus vizinhos até então: $\boldsymbol{g_{best\_i}}$
    e. O parâmetro fator de inércia (descreve a influência da velocidade de uma partícula no cálculo de sua nova velocidade): $\boldsymbol{w}$
    f. O parâmetro coeficiente cognitivo (descreve a influência que a distância da posição atual para a melhor posição já visitada pela partícula afetará no cálculo de sua nova velocidade): $\boldsymbol{\varphi_p}$
    g. O parâmetro coeficiente social (descreve a influência que a distância da posição atual para a melhor posição já visitada por partículas vizinhas afetará no cálculo de sua nova velocidade): $\boldsymbol{\varphi_g}$

    Sendo $r_p$ e $r_g$ dois números amostrados aleatoriamente de maneira uniforme no intervalo *(0,1)*, o vetor de velocidade $v_i$ de uma dada partícula obtido na iteração *i* se dará então pela seguinte fórmula:

$$v_i = wv_{i-1} + \varphi_p r_p (p_{best_i} - p_{i-1}) + \varphi_g r_g (g_{best_i} - p_{i-1}) \quad \textbf{Eq.1}$$

2. Cada partícula terá então sua posição atualizada conforme o cálculo acima, somando o novo vetor velocidade obtido no passo anterior à sua coordenada atual:

$$p_i = v_i + p_{i-1} \quad \textbf{Eq.2}$$

3. Um procedimento comum nesse momento envolve validar a nova posição obtida caso ela se encontre fora do domínio da função do problema no espaço de busca. Nessa situação, existem diferentes tratamentos aplicáveis, denominados métodos de confinamento. Alguns exemplos incluem:

    a. Não efetuar o movimento desta partícula nessa iteração.

b. Permitir o movimento, mas confinar as coordenadas no interior da região de busca definida.

c. Permitir que a partícula saia do espaço de busca, apenas atribuindo um valor maximamente custoso ao retorno da função objetivo quando aplicada com esta coordenada de fora do espaço de busca.

4. A nova coordenada é então aplicada na função objetivo, atualizando $p_{best\_i}$ com o novo valor, caso necessário. Do mesmo modo, $g_{best\_i}$ também é atualizado em todas partículas vizinhas caso seu valor atual seja pior que aquele obtido pela nova coordenada.

O processo descrito acima se repetirá para toda partícula em repetidas iterações até a condição de parada ser satisfeita. Como meta-heurística, a escolha da condição de parada do PSO se dará arbitrariamente pela implementação definida, comumente sendo estipulada como quando um limite no número de iterações é atingido ou quando é deixado de ser observado mudanças expressivas dentro de uma certa precisão nas soluções parciais de cada partícula de uma iteração a outra.

Ao final desse processo, as diferentes partículas tenderão a se encontrar próximas a valores de mínimo ou máximo global (a depender da modelagem do problema). Uma vantagem do algoritmo PSO se dá pela possibilidade de se identificar múltiplos pontos ótimos em uma única execução (no caso da função otimizada possuir essa característica), sendo possível que diferentes aglomerações de partículas possam ter, de maneira independente, alcançado pontos distintos de máximo ou mínimo global.

Devido a sua poderosa capacidade de otimização de funções n-dimensionais de domínio contínuo e pelo fato do algoritmo não necessitar de certos pré-requisitos do problema a ser atacado, comumente demandados por soluções alternativas (a função a ser otimizada não precisa ser diferenciável para a obtenção de seu gradiente, por exemplo), a meta-heurística PSO se demonstrou extremamente útil na aplicação de diferentes soluções das mais diversas áreas. Um desses exemplos inclui problemas de parametrização de sistemas mais complexos, como por exemplo no ajuste dos pesos de redes neurais como alternativa ao método de treinamento por *backpropagation* (GUDISE; VENAYAGAMOORTHY, 2003).

## 2.2 COLÔNIA DE FORMIGAS PARA DOMÍNIOS DISCRETOS

Um dos primeiros algoritmos bioinspirados a serem propostos na literatura foi o algoritmo de Otimização de Colônia de Formigas, desenvolvido originalmente por Marco Dorigo em sua tese de doutorado (DORIGO, 1992). A inspiração na natureza para o algoritmo se dá no comportamento observado de colônias de certas espécies de formigas, um sistema complexo onde diversos agentes trabalham cooperativamente, através da liberação de feromônios no ambiente, para encontrar caminhos progressivamente mais curtos entre o formigueiro de origem e suas fontes de alimento.

Em sua proposta original, a Otimização de Colônia de Formigas foi implementada com o intuito de prover soluções apenas a problemas de otimização combinatória, isto é, considerando domínios discretos, especificamente caminhos de custos otimizados em um grafo ponderado. Ao final deste capítulo será apresentada uma extensão do algoritmo ACO usual para problemas em espaço discreto voltada para otimização de funções com domínios contínuos.

Como descrito anteriormente, o procedimento definido na meta-heurística ACO é baseado em uma simulação da atividade de formigas reais em busca de alimento. A partir de um mesmo nó inicial (formigueiro), as formigas vem a percorrer o grafo do problema à procura do caminho de menor custo entre seu ponto de partida e um nó objetivo conhecido (fonte de alimento). O ACO utiliza uma estratégia de construção iterativa para encontrar a solução final otimizada, com cada formiga contendo em sua memória uma solução parcial para o problema a ser resolvido ao final de cada passo do algoritmo. Ao final de uma iteração, cada formiga poderá ter aprimorado sua respectiva solução ótima parcial até que a solução aproximadamente ótima do problema seja obtida por uma das formigas, após uma quantidade suficientemente grande de execuções. Esse processo de otimização das soluções parciais de formigas individuais se dá através do uso de uma função não determinística na escolha da ordem dos nós que uma dada formiga irá visitar em sua busca, baseado no conceito de feromônios.

Na natureza, uma formiga em busca de alimentos libera uma quantidade fixa de feromônios no ambiente por onde passa. Partindo do formigueiro até uma fonte de alimento, a formiga retornará ao formigueiro para novamente iniciar a viagem à caminho do alimento. Esse processo se repetirá até a fonte de alimentos se esgotar. A formiga que encontrar o menor caminho entre o formigueiro e o alimento, com o passar do tempo, depositará mais feromônios por área percorrida nesse processo de ida e volta do que qualquer outra formiga que encontrar uma rota menos otimizada. Essa rota de feromônios mais intensos, detectável por outras formigas, por sua vez, incentivará futuras formigas a escolher esse mesmo caminho no longo prazo, resultando em uma convergência entre os agentes do formigueiro em uma única rota otimizada.

### 2.2.1 Descrição da Execução do ACO

Desse modo, de forma análoga ao comportamento observado na natureza, uma simples implementação do algoritmo ACO terá então o seguinte procedimento por iteração:

1. Construção de Solução Candidata por Formiga:

   Cada formiga realizará uma busca pelo grafo a procura de um dos nós objetivo descrito pelo problema modelado. A decisão da ordem das arestas visitadas por cada formiga em sua busca é decidida de forma não determinística, a partir de uma função para escolha da aresta, parametrizada por dois fatores de incentivo:
   a. A quantidade de feromônios depositada na aresta candidata para a travessia em questão.
   b. Um fator fixo definido a priori na modelagem do problema, descrevendo a favorabilidade padrão da formiga escolher uma aresta sobre outra. Um exemplo possível para esse fator é o inverso do custo da aresta (ou a distância entre os dois nós ligados pela mesma). Desse modo, a formiga teria uma preferência inata, independente da quantidade de feromônios depositada nela, maior pela aresta que contempla a menor distância para o nó seguinte, por exemplo.

   A probabilidade de uma dada formiga transicionar de um estado $x$ para um estado $y$ se dá pela seguinte fórmula:

$$p_{xy} = \frac{(\tau_{xy})^{\alpha}(\eta_{xy})^{\beta}}{\Sigma_{z \in estadosAdjacentes}(\tau_{xz})^{\alpha}(\eta_{xz})^{\beta}} \quad \textbf{Eq.3}$$

Sendo $\tau_{xy}$ a quantidade de feromônio depositada na transição do estado $x$ para o estado $y$ e $\eta_{xy}$ o fator fixo a priori dessa transição de estados, com os parâmetros $\alpha$ e $\beta$ controlando respectivamente a influência dos feromônios depositados e do fator a priori na decisão da transição escolhida.

2. Atualização de Feromônios:

Após as formigas efetuarem suas buscas, o feromônio contido nas arestas visitadas é atualizado de acordo com o desempenho de cada formiga. A quantidade de feromônio depositada em uma aresta visitada por uma formiga será inversamente proporcional ao custo, como a distância, do caminho final obtido por ela em sua busca. Nessa etapa também é geralmente aplicado um decremento global na quantidade de feromônio em todas as arestas do grafo. Esse fator é denominado o coeficiente de evaporação do feromônio e tem como função acelerar a convergência da solução final, desincentivando formigas de visitar caminhos pouco visitados em iterações passadas.

A fórmula abaixo descreve a atualização dos feromônios contidos na transição do estado $x$ para o estado $y$, sendo $\rho$ o coeficiente de evaporação, $m$ o total de formigas instancionadas e $\Delta\tau^{k}_{xy}$ a quantidade de feromônio depositada pela formiga $k$:

$$\tau_{xy} \leftarrow (1 - \rho)\tau_{xy} + \sum_{k}^{m} \Delta\tau^{k}_{xy} \quad \textbf{Eq.4}$$

A quantidade $\Delta\tau^{k}_{xy}$ de feromônios depositada pela formiga $k$ é arbitrária, a depender do problema atacado. Como exemplo, em problemas como do caixeiro viajante, é tipicamente utilizada a seguinte fórmula:

$$\Delta\tau^{k}_{xy} = Q/L_{k} \; se\ a\ formiga\ k\ passou\ por\ xy$$

$$\Delta\tau^{k}_{xy} = 0 \; caso\ contrário \quad \textbf{Eq.5}$$

Sendo $L_k$ o custo total da solução candidata encontrada pela formiga $k$ nessa iteração e $Q$ uma constante arbitrária.

3. Fim da Iteração e Verificação da Condição de Parada:

Se encontrada, a melhor solução é atualizada para retorno no término do algoritmo e uma nova iteração é iniciada no caso da condição de parada não ter sido alcançada. Iniciada uma nova iteração, as formigas partirão novamente do nó inicial em busca de um nó objetivo, agora com o espaço de busca atualizado com novas quantidades de feromônios. Nota-se que a condição de parada pode ser decidida arbitrariamente, dependendo do problema atacado. Alguns exemplos de condições de parada possíveis incluem estabelecer um limite fixo de iterações, ou quando for observado que o melhor resultado obtido em um certo número de iterações consecutivas não esteja apresentando melhora significativa em seu resultado ótimo, indicando que a convergência já teria sido alcançada.

O algoritmo de Otimização de Colônia de Formigas tem sido utilizado em uma variedade de aplicações práticas desde sua introdução no início dos anos 90. Por sua simples implementação, relativo baixo custo e ampla aplicabilidade — necessitando apenas que o problema a ser otimizado possa ser modelado como um grafo ponderado — o ACO foi rapidamente adotado (assim como outras meta-heurísticas apresentadas na época, como o algoritmo genético) em diferentes áreas como uma solução efetiva para diferentes problemas NP-difíceis de otimização.

Alguns problemas de otimização combinatória notórios na literatura atendidos pelo ACO incluem o problema do caixeiro viajante (*Traveling Salesman Problem*) (STÜTZLE et al., 1999), que envolve encontrar o menor caminho possível que passa por todas as cidades de uma lista uma vez e retorna à cidade de origem, e o problema de sequenciamento de tarefas (*Job Scheduling Problem*) (KANT et al., 2010). Este problema, por sua vez, envolve a alocação de tarefas a máquinas ou trabalhadores de modo a minimizar o tempo total de conclusão ou o custo total.

Outras aplicações práticas na indústria incluem a otimização de rotas de transporte (MONTEMANNI et al., 2005), como rotas de entrega de caminhões ou rotas de voo, e a otimização de diferentes processos industriais, como de sistemas de telefonia (CARO; DORIGO, 2004), sistemas de gerenciamento de energia (MARZBAND et al., 2016) e serviços de computação em nuvem (FARAHNAKIAN et al., 2014).

## 2.3 COLÔNIA DE FORMIGAS PARA DOMÍNIOS CONTÍNUOS

Com os avanços na pesquisa em meta-heurísticas bioinspiradas em problemas de otimização no fim dos anos 90 e nos anos 2000, impulsionada pela inovação do algoritmo ACO e pela subsequente expansão na aplicabilidade desses algoritmos ocasionada por outras inovações de sucesso (um exemplo paradigmático sendo o algoritmo PSO), foi observada uma grande proliferação na publicação de soluções algorítmicas bioinspiradas para essa classe de problemas de otimização. Alguns exemplos notáveis incluem algoritmos Artificial Bee Colony (KARABOGA et al., 2014), Firefly Swarm Optimization (FISTER et al., 2013), Bat Optimization (YANG; HE, 2013) e Cuckoo Search (FISTER et al., 2014).

Nesse contexto, Krzysztof Socha e Marco Dorigo (autor do algoritmo ACO original) publicam uma extensão ao ACO, denominado $ACO_R$, tornando-o capaz de otimizar variáveis de domínios contínuos sem grandes mudanças conceituais na meta-heurística originalmente proposta para otimização combinatória (SOCHA; DORIGO, 2008). O novo algoritmo estendido, desse modo, manterá o mesmo procedimento conceitual do ACO original, em particular seu processo de construção de soluções parciais de maneira iterativa. Este processo, que se dá início após a inicialização dos diferentes parâmetros do algoritmo, ainda será caracterizado pelas mesmas etapas a cada passo de iteração:

1. Construção de Solução Candidata por Formiga;
2. Atualização de Feromônios;
3. Fim da Iteração e Verificação da Condição de Parada;

Para explicitar as diferenças do $ACO_R$ para o ACO, é importante observar que a ideia central para o funcionamento do algoritmo original envolve a construção incremental de soluções parciais baseada em uma escolha probabilística (enviesada pela presença de feromônios no espaço de busca) de componentes que virão a compor a solução final obtida na iteração.

No caso do algoritmo ACO original para domínios discretos, essa escolha de componentes a cada passo da construção se dá através de uma distribuição discreta de probabilidade. Exemplificando no contexto de um problema como o do caixeiro viajante, a formiga a cada passo da construção de sua solução candidata irá amostrar essa distribuição discreta para decidir qual aresta (componente da solução) seguir a partir do caminho que já caminhou até então (sua solução parcial). Após ser feita a decisão, o componente escolhido é adicionado à solução parcial na memória da formiga e o processo de construção é retomado até uma solução candidata ser obtida.

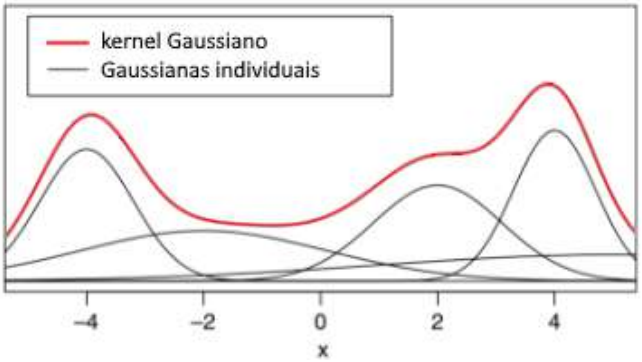
A mudança fundamental realizada na extensão de domínios discretos para domínios contínuos no algoritmo $ACO_R$ se dá na utilização de uma distribuição contínua de probabilidade no lugar de uma distribuição discreta. Assim, a formiga a cada passo do processo de construção amostrará uma função de densidade de probabilidade na escolha dos componentes para a construção de sua solução candidata. A função proposta, no artigo que o $ACO_R$ foi introduzido, é um conjunto de $n$ funções de densidade de probabilidade kernel Gaussiana $G^i(x)$, uma para cada variável a ser otimizada no problema, com cada uma sendo definida como uma soma ponderada de $k$ distintas funções Gaussianas unidimensionais $g^i_l(x)$:

$$G^i(x) = \sum_{l=1}^{k} \omega_l \, g^i_l(x) = \sum_{l=1}^{k} \omega_l \frac{1}{\sigma^i_l \sqrt{2\pi}} e^{\frac{(x-\mu^i_l)^2}{2\sigma^{i\,2}_l}} \qquad \textit{Eq.6}$$

Na formulação acima de uma dessas n funções kernel Gaussiana $G^i(x)$ (com $i = 1, \ldots, n$ para as $n$ diferentes variáveis a serem otimizadas do problema) temos que esta será parametrizada fazendo uso de três vetores: $\omega$ sendo o vetor de pesos associados a cada função Gaussiana, $\mu^i$ sendo o vetor de médias e $\sigma^i$ sendo o vetor de desvios-padrões. Os três vetores terão a mesma cardinalidade, sendo esta o número de funções Gaussianas que constituem o kernel Gaussiano, definida previamente pelo parâmetro $k$. Esse parâmetro, a ser definido na inicialização do algoritmo, será determinante no formato do gráfico da distribuição final.

A escolha da função de densidade de probabilidade kernel Gaussiano se dá por, além de sua facilidade de amostragem, sua grande flexibilidade no formato de seu gráfico. Uma função Gaussiana simples, não será capaz de descrever situações onde duas ou mais áreas disjuntas do espaço de busca se demonstrem promissoras devido ao fato de possuir apenas um único ponto de máximo local em seu gráfico. Esse problema é contornado pelo uso da soma ponderada de $k$ diferentes distribuições Gaussianas na formulação do kernel, podendo prover até $k$ distintos pontos de máximo local para a distribuição resultante.

**Gráfico 1:** Exemplo de Função Kernel Gaussiana




*Função kernel Gaussiana composta pela soma ponderada de 5 distintas funções Gaussianas.* Fonte: Socha e Dorigo (2008, p. 1159)

Uma outra diferença na implementação do algoritmo $ACO_R$ quando comparado com o ACO padrão se dá no processo de utilização dos feromônios para atualizar o comportamento da busca ao longo de suas iterações. Enquanto que no algoritmo original os componentes de soluções candidatas encontradas eram utilizados para diretamente modificar uma tabela de feromônios (representando as transições entre os diferentes nós do grafo do espaço de busca e sua respectiva quantidade de feromônio), no $ACO_R$ estes componentes serão utilizados para dinamicamente gerar funções de densidade de probabilidade. Nota-se que a utilização de uma tabela de feromônios como a utilizada no ACO não seria possível para o caso contínuo, visto que seria necessário uma quantidade infinita de campos para descrever o espaço de busca dessa classe de problemas.

No $ACO_R$, essa geração dinâmica de funções de densidade de probabilidade também se dará através da manutenção de uma tabela, denominada tabela de soluções *T*. As soluções contidas nessa tabela serão utilizadas para calcular os valores dos três vetores $\omega$, $\mu^i$, $\sigma^i$ (todos de cardinalidade *k*) mencionados previamente, que nos fornecerão o formato das funções de densidade de probabilidade kernel Gaussiana utilizadas para guiar as formigas em seu processo de busca.

Na tabela *T* estarão contidas *k* diferentes soluções candidatas memorizadas, determinando a quantidade de funções Gaussianas $g^i_l(x)$ ($l = 1, ..., k$) que irão compor uma função de densidade de probabilidade kernel Gaussiana $G^i(x)$ ($i = 1, ..., n$) para cada variável contínua *n* a ser otimizada. O parâmetro *k*, como descrito anteriormente, determinará a complexidade da distribuição resultante. A seguir se encontra descrito a tabela de soluções *T* mantida pelo $ACO_R$, com diferentes soluções candidatas descritas em cada linha, ordenadas por seus resultados na função objetivo *f* a ser otimizada. Cada solução também terá associada a ela um peso proporcional à qualidade da solução. Considerando o caso de um problema de minimização teremos então, segundo a tabela *T* abaixo, que $f(s_1) \leq f(s_2) \leq \ldots \leq f(s_k)$ e $\omega_1 \geq \omega_2 \geq \ldots \geq \omega_k$.

**Figura 1:** Tabela de soluções *T*

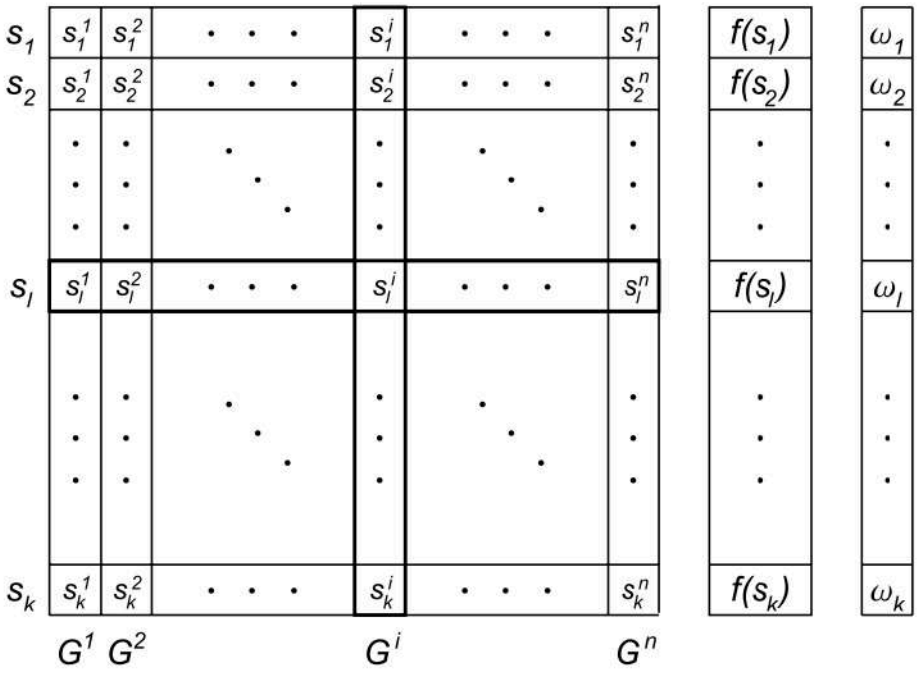



*Tabela T de tamanho k para um problema com domínio de n dimensões.* Fonte: Socha e Dorigo (2008, p. 1160)

Cada linha da matriz principal de *T* representa uma solução candidata, com cada uma das *n* colunas contendo o vetor que descreve o valor de uma mesma variável do problema

para as diferentes $k$ soluções candidatas da tabela. O vetor com os resultados das diferentes soluções candidatas aplicadas na função objetivo $f$ também é mantido em $T$. Os valores contidos em cada coluna dessa matriz, em conjunto com o vetor de pesos $\omega$ também armazenado em $T$, serão utilizados para definir n funções de densidade de probabilidade kernel Gaussiana $G^i$. Essas funções serão utilizadas pelas formigas durante o processo de iteração para gerar novas soluções candidatas.

### 2.3.1 Descrição da Execução do $ACO_R$

Como mencionado, a meta-heurística $ACO_R$ segue conceitualmente os mesmos passos que o algoritmo ACO em seu processo iterativo. Essas etapas, análogas às listadas previamente no caso do algoritmo para o caso discreto, se encontram descritas a seguir:

1. Construção de Solução Candidata por Formiga:

   Uma solução candidata composta por $n$ diferentes componentes será construída, sendo estes $n$ componentes as variáveis contínuas da função a ser otimizada. Essas variáveis $X_i$ ($i = 1, ..., n$) serão construídas sequencialmente por uma formiga em $n$ passos de construção. Como mencionado previamente, a função de densidade de probabilidade kernel Gaussiano é composta pela soma ponderada de múltiplas funções Gaussianas, com esse número de funções sendo definido pelo parâmetro $k$ do algoritmo que também definirá o número de linhas da tabela de solução $T$ descrita na *Figura 1*. No passo de construção $i$ da solução candidata de uma formiga, para decidir o componente $X_i$ da solução candidata, iremos amostrar da função Gaussiana kernel $G^i$.

   Cada uma das diferentes funções $G^i$ são definidas a partir de três vetores de parâmetro $\mu^i$, $\omega$, $\sigma^i$. Os dois primeiros desses vetores poderão ser calculados diretamente a partir dos valores contidos na tabela $T$:

   **O vetor de médias $\boldsymbol{\mu^i}$** será simplesmente definido como sendo composto pelos diferentes valores de uma mesma variável de diferentes soluções contidas na coluna $i$:

   $$\mu^i = [\mu^i_1, \dots, \mu^i_k] = [s^i_1, \dots, s^i_k] \quad \textit{Eq.7}$$

   **O vetor de pesos $\boldsymbol{\omega}$**, incluído na figura da tabela de soluções $T$ acima, é obtido pelo processo descrito a seguir. Primeiramente, cada solução candidata adicionada a $T$ é avaliada na função objetivo $f$ e ranqueada baseada em seu resultado (eventuais empates são resolvidos aleatoriamente), sendo adicionada a $T$ em sua devida posição no ranque. Dada uma solução $s_l$ de ranque $l$, o elemento $\omega_l$ do vetor $\omega$, respectivo a essa solução, será definido seguindo a seguinte fórmula:

   $$\omega_l = \frac{1}{qk\sqrt{2\pi}} e^{\frac{-(l-1)^2}{2q^2k^2}} \quad \textit{Eq.8}$$

   A expressão acima descreve o peso $\omega_l$ como sendo um valor da função Gaussiana padrão com argumento $l$, média 1 e desvio padrão $qk$, com $q$ sendo um outro parâmetro do algoritmo $ACO_R$. Esse parâmetro $q$, quando pequeno, determinará uma preferência por soluções com melhor ranque $l$. Para valores de $q$ maiores, o algoritmo assumirá uma probabilidade mais uniforme de utilizar resultados das

diferentes soluções contidas em $T$ no seu processo de busca. Esse comportamento será explorado mais a fundo no capítulo 4.

**O vetor de desvios-padrões $\sigma^i$**, por sua vez, não será calculado por completo devido a forma que a amostragem da função kernel Gaussiana $G^i$ será realizada na prática. Esse processo será descrito a seguir.

A amostragem de $G^i$ será realizada em duas partes. Primeiramente é escolhida uma única das $k$ funções Gaussianas $g^i_l$ ($l = 1, ..., k$) que compõem a função Gaussiana kernel $G^i$ para ser amostrada. A decisão de $l$ é feita a partir da seguinte probabilidade:

$$p_l = \frac{\omega_l}{\sum_{r=1}^{k} \omega_r} \quad \textit{Eq.9}$$

É de certa importância para o algoritmo que a escolha de $l$ realizada nessa etapa seja feita uma única vez por iteração por cada formiga ao começar a construir sua solução candidata, com o intuito de explorar a correlação entre as diferentes variáveis. Nota-se que isso não significa que a função $g^i_l$ a ser amostrada na etapa seguinte será a mesma para toda variável componente da solução candidata (visto que para o passo $i$, $\mu^i = s^i_l$ e também $\sigma^i_l$ variará para diferentes valores de $i$).

Na segunda etapa, amostramos diretamente da função Gaussiana $g^i_l$, com $l$ sendo o valor obtido na etapa anterior. Isto será feito fazendo uso do elemento $l$ dos vetores $\mu^i$ e $\omega$ já calculados. O desvio-padrão $\sigma^i_l$, necessário para realizar a amostragem de $g^i_l$, será calculado dinamicamente a partir da seguinte fórmula:

$$\sigma^i_l = \xi \sum_{e=1}^{k} \frac{|s^i_e - s^i_l|}{k-1} \quad \textit{Eq.10}$$

Com os valores de $\mu^i_l$ e $\sigma^i_l$ obtidos, podemos finalmente realizar a amostragem da função Gaussiana $g^i_l$, obtendo o componente $X_l$ da solução candidata nesse passo de construção. Esse processo será repetido para cada uma das $n$ variáveis do problema até uma solução candidata completa ser obtida pela formiga.

2. Atualização de Feromônios:

A atualização de feromônios no caso do $ACO_R$, diferentemente do algoritmo ACO, não se dá diretamente no espaço de busca do problema, mas sim na própria tabela $T$ de soluções candidatas. Esse processo se dá simplesmente substituindo o conjunto das piores soluções ranqueadas em $T$ pelo conjunto de soluções novas geradas pelas formigas nesta iteração, mantendo o mesmo total de $k$ soluções armazenadas em $T$. Este procedimento descrito funciona simultaneamente como a atualização dos feromônios com dados de soluções melhores, de forma análoga à evaporação de feromônios do ACO padrão, removendo as piores soluções da tabela $T$.

3. Fim da Iteração e Verificação da Condição de Parada:

No fim desse passo de iteração a tabela de soluções $T$ é reordenada baseado no custo das diferentes soluções. Caso a condição de parada (podendo ser decidida arbitrariamente) tenha sido alcançada, a solução com melhor ranque é retornada pelo algoritmo. Caso contrário, uma nova iteração se inicia.

Similarmente ao algoritmo PSO, o $ACO_R$ é capaz da otimização, sem grandes restrições, de uma ampla gama de funções n-dimensionais de domínio contínuo. A meta-heurística $ACO_R$ se apresentou como uma alternativa a outras meta-heurísticas concebidas desde a popularização do algoritmo PSO para atacar essa classe de problemas, com o benefício adicional de ser uma extensão do algoritmo ACO, um dos algoritmos bio-inspirados de otimização mais estudados e utilizados.

# 3 EXECUÇÕES DOS ALGORITMOS

Neste capítulo serão apresentados exemplos de execuções dos algoritmos PSO e $ACO_R$ fazendo uso do código desenvolvido para a visualização e realização de experimentos disponível no repositório do github do trabalho[2] com as instruções para execução.

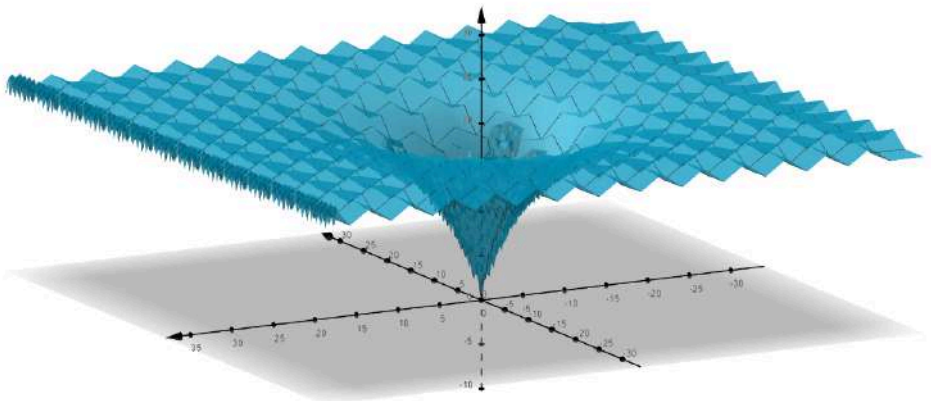
O código desenvolvido inclui as implementações das meta-heurísticas PSO e $ACO_R$ conforme suas descrições no capítulo anterior. O programa oferece uma interface de fácil uso ao usuário para a escolha do algoritmo a ser utilizado, seguido da seleção da função objetivo a ser otimizada e da escolha dos valores do conjunto de parâmetros do algoritmo selecionado. Com isso definido, o algoritmo de otimização é executado para a função selecionada, com informações sobre o resultado do processo de otimização realizado sendo exibidas no terminal em conjunto com uma janela contendo uma visualização do gráfico do histórico de de melhor custo obtido a cada passo de iteração. Caso o algoritmo selecionado tenha sido a Otimização de Enxame de Partículas (PSO), também será exibida uma animação descrevendo o deslocamento das diferentes partículas no espaço de busca.

Para possibilitar essa exibição gráfica sem necessitar de uma parametrização excessiva do usuário para fins de renderização (o que escaparia da proposta central do trabalho prático de focar na exploração de diferentes parametrizações dos algoritmos em si), o código oferece uma lista fixa de diferentes funções objetivas com sua região de busca e configurações de visualização já previamente definidas para o usuário escolher. Essa lista inclui 4 diferentes funções objetivo, sendo estas um paraboloide de revolução simples e as três funções populares de referência *Ackley*, *Rastrigin* e *Three-hump*.

Em ambos exemplos de execução que seguirão, faremos uso da função objetivo *Ackley* de referência, nos limitando à região contida no intervalo *(-10, 10)* em ambas variáveis do domínio, como definido no código. Essa função não linear é caracterizada pela presença de múltiplos pontos de mínimo local por todo seu espaço de busca, com apenas um único ponto ótimo de mínimo global. Sua equação para um domínio de duas dimensões e ponto ótimo global em *(0, 0)* é dada por:

$$f(x,y) = -\ 20e^{-0.2\sqrt{0.5(x^2+y^2)}} - e^{0.5(cos(2\pi x)+cos(2\pi y))} + 20 + e \quad \textit{Eq.11}$$

**Gráfico 2:** Função Ackley de referência



[2] Repositório disponível em: https://github.com/VideoJope/AntColony-and-ParticleSwarm-Optimization. Para executar o código e acompanhar os resultados das execuções deste capítulo, siga as instruções contidas no arquivo *readme.md* e execute o script do arquivo *main.py*, localizado no diretório */src* do repositório. Após isso, simplesmente acompanhe as instruções exibidas no terminal como ilustradas nos exemplos que seguirão.

## 3.1 EXEMPLO DE EXECUÇÃO DO PSO

O algoritmo PSO será executado no código por 100 iterações, utilizando as seguintes parametrizações escolhidas arbitrariamente (definidas como padrão no código):

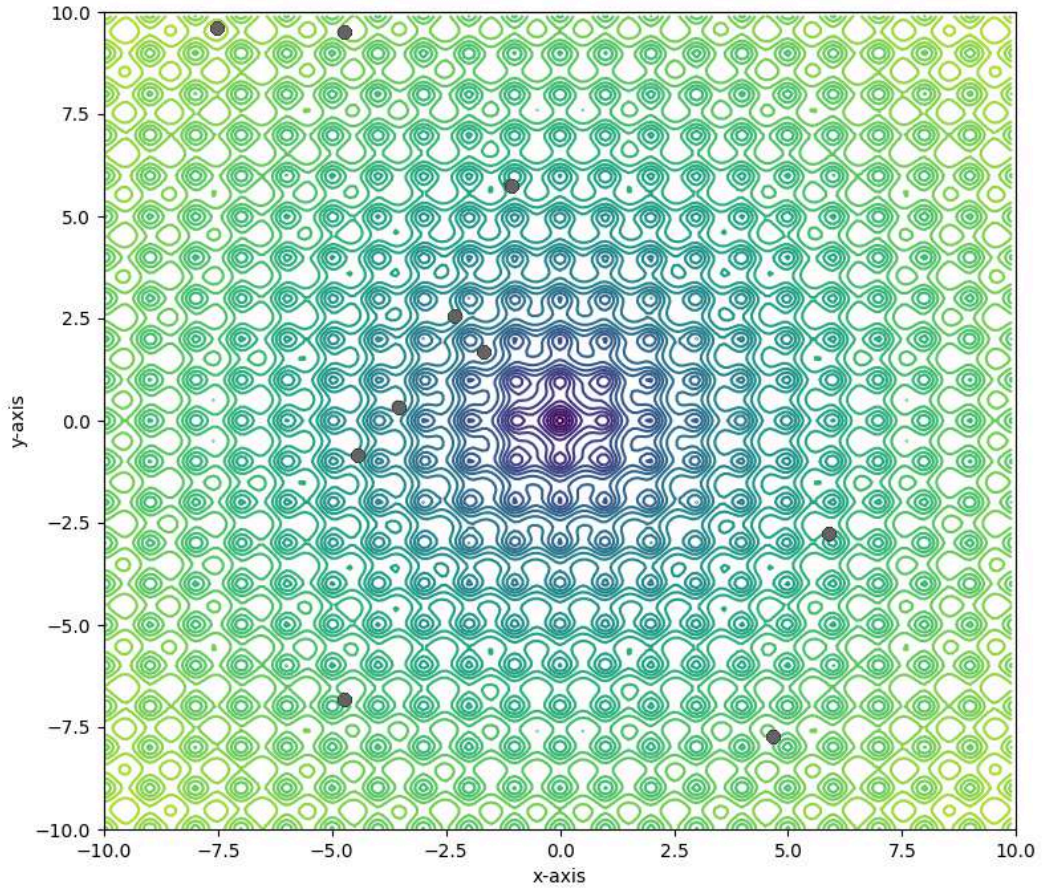
- Número total de partículas: *10*
- Coeficiente cognitivo: *0,5*
- Coeficiente social: *0,3*
- Fator de inércia: *0,9*

No terminal, essa parametrização é configurada da seguinte maneira a partir da execução do código contido em *main.py*:

```
############
# Welcome! #
############

Select which algorithm you wish to execute by typing their respective number:
  [1] Particle Swarm Optimization
  [2] Ant Colony Optimization (Continuous Domain)
1

Select which objective function you wish to optimize:
  [1] Sphere function
  [2] Ackley function
  [3] Threehump function
  [4] Rastrigin function
2

Insert the number of iterations the selected algorithm will run for [default=100]:
100

Insert the number of particles [default=10]:
10

Insert the cognitive coefficient parameter value [default=0.5]:
0.5

Insert the social coefficient parameter value [default=0.3]:
0.3

Insert the inertia weight parameter value [default=0.9]:
0.9
```

O programa irá então inicializar as 10 partículas de maneira uniformemente aleatória na região do espaço de busca descrito e fará uso de uma topologia completa (cada partícula se comunicará com todas as demais sobre sua posição). Dessa forma, teremos as seguintes posições iniciais para as 10 partículas:

**Gráfico 3:** Posições iniciais das partículas




*Posições escolhidas de maneira arbitrariamente uniforme no espaço de busca. Cores mais escuras na malha indicam uma melhor qualidade da solução descrita por esta região na função objetivo.*

As 10 partículas inicializadas conforme a figura acima terão as seguintes coordenadas como suas posições iniciais:

```
Posições iniciais:
[-4.7595065  -6.82632056]
[-4.43746961 -0.81366226]
[-3.57998919  0.36785641]
[-4.76114149  9.5217057 ]
[ 4.65629105 -7.69451547]
[-2.27449863  2.57002359]
[-7.49884147  9.6709721 ]
[-1.13550263  5.79116684]
[ 5.88237152 -2.77476855]
[-1.67792121  1.68516256]
```

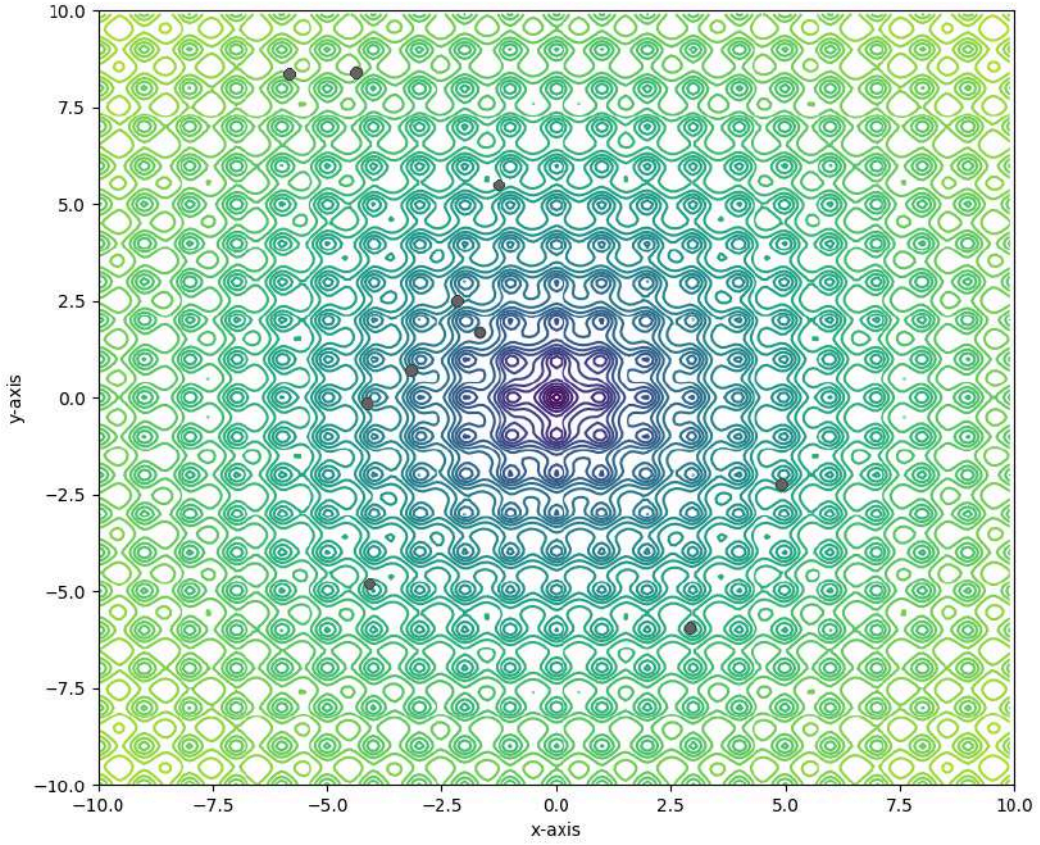
Seguindo a implementação descrita no capítulo anterior, o código considerará uma velocidade inicial igual a 0 para todas as partículas. Nessa primeira iteração então o fator de inércia não terá influência na obtenção da nova velocidade calculada. O valor dos vetores de velocidade de cada uma das respectivas 10 partículas listada anteriormente, calculados na primeira iteração resultam em:

```
Velocidades atualizadas na primeira iteração:
       [ 0.66927216  2.02494065]
       [ 0.29101433  0.68191944]
       [ 0.40678815  0.35458495]
       [ 0.38867845 -1.09911302]
       [-1.71770003  1.78274458]
       [ 0.09536221 -0.06304921]
       [ 1.65285715 -1.34025073]
       [-0.12324046 -0.2908889 ]
       [-0.97473054  0.52731585]
       [ 0.          0.        ]
```

Nota-se que a última partícula da lista, a partícula de melhor posição *(-1.67792121, 1.68516256)*, manteve sua velocidade igual a 0. Sua velocidade não será atualizada pois, conforme descrita na *Eq.1*, sua posição $p_1$, sua melhor posição $p_{best\_1}$ e o melhor global $g_{best\_1}$ terão o mesmo valor, resultando em um vetor velocidade de módulo igual a 0.

Após a obtenção dos vetores de velocidade, as posições de cada partícula são atualizadas conforme a *Eq.2* para se dar início a iteração seguinte. A seguir se encontram as novas posições das partículas ao fim dessa primeira iteração:

**Gráfico 4:** Posições das partículas ao fim da primeira iteração




Percebe-se que a partícula mais perto da origem de fato não teve sua posição alterada, visto que sua velocidade corrente é igual a zero pelas razões dadas previamente. De maneira geral, o comportamento observado no sistema de uma iteração a outra é uma aproximação das demais partículas em sua direção.

Com a velocidade e posição das partículas atualizadas, a iteração seguinte será iniciada. A partir do momento que uma partícula ganha velocidade, o fator de inércia passará a fazer efeito no cálculo de sua nova velocidade como descrito na *Eq.1*. As influências desse e dos demais parâmetros serão exploradas em profundidade no capítulo 4. Após as 100 iterações estabelecidas como critério de parada serem executadas, teremos a seguinte posição como melhor visitada pelo sistema de partículas:

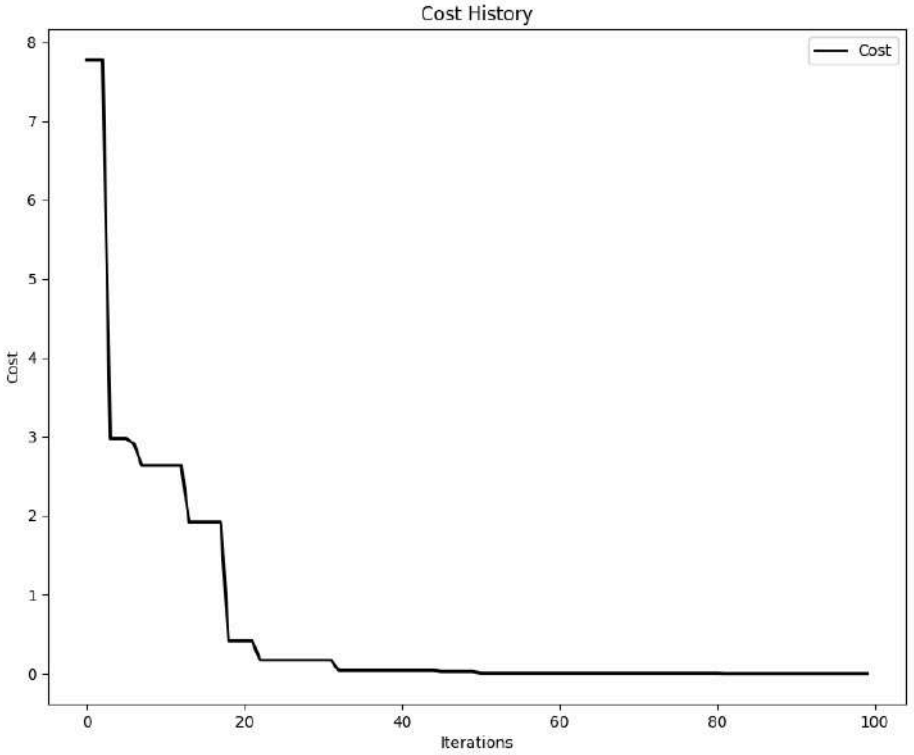
```
Melhor posição:
[8.44141327e-05 6.73990208e-04]
```

Essa posição, próxima do ponto de ótimo global *(0, 0)* conhecido da função, quando avaliada na função *Ackley* objetivo retorna o seguinte valor:

```
Custo da melhor posição:
0.0019335117277674563
```

Como sabemos que a função possui um valor de mínimo global ótimo igual a 0 na origem, temos que esse valor obtido também equivale ao erro obtido ao final do processo de otimização. A seguir se encontra descrito o histórico do custo da melhor posição encontrada até então ao longo das 100 iterações que o algoritmo foi executado:

**Gráfico 5:** Histórico do custo da melhor solução conhecida (PSO)




Como já mencionado, além do gráfico do histórico do custo da melhor solução também será exibida ao fim da otimização uma animação descrevendo o histórico de posições da população de partículas no espaço de busca ao longo das diferentes iterações do algoritmo. Essa visualização ilustra de maneira clara o movimento das diferentes partículas durante a evolução do algoritmo até sua eventual convergência.

## 3.2 EXEMPLO DE EXECUÇÃO DO $ACO_R$

Para o exemplo de execução do algoritmo $ACO_R$, faremos novamente uso da mesma função *Ackley* de referência utilizada no exemplo de execução do PSO. Como condição de parada, o algoritmo $ACO_R$ também será executado por 100 iterações, fazendo uso das seguintes parametrizações definidas arbitrariamente (e novamente definidas como padrões no código):

- Tamanho *k* da tabela de soluções *T*: *10*
- Número de formigas: *3*
- Parâmetro *q*: *1*
- Parâmetro $\xi$: *1*

Novamente, para acompanhar esta execução no código, a parametrização acima é configurada da seguinte forma a partir da execução do script *main.py*:

```
#############
# Welcome! #
#############

Select which algorithm you wish to execute by typing their respective number:
  [1] Particle Swarm Optimization
  [2] Ant Colony Optimization (Continuous Domain)
2

Select which objective function you wish to optimize:
  [1] Sphere function
  [2] Ackley function
  [3] Threehump function
  [4] Rastrigin function
2

Insert the number of iterations the selected algorithm will run for [default=100]:
100

Insert the number of ants [default=10]:
3

Insert the size k (number of lines) of the solutions archive T [default=10]:
10

Insert the q coefficient parameter value [default=1]:
1

Insert the epsilon coefficient parameter value [default=1]:
1
```

A tabela de soluções *T* é então inicializada de maneira uniformemente aleatória com 10 soluções candidatas para dar início ao algoritmo. Esta também já irá ser ordenada se baseando na qualidade de suas soluções quando aplicadas à função objetivo, resultando na seguinte tabela *T* inicial:

```
Soluções candidatas iniciais da tabela de soluções T:
             [ 5.87309905 -1.71426564]
             [-6.12536675 -3.16525605]
             [-0.08418199  7.96742466]
             [-8.32371194 -2.15808729]
             [ 7.56203801 -6.28944077]
             [-3.20857547 -9.55395067]
             [-9.87082762 -5.45177066]
             [ 7.49016181 -7.68514474]
             [ 8.4180091   8.93131728]
             [ 7.80876632 -9.52260635]
```

Inicializaremos também o vetor de pesos $\omega$ respectivo de cada solução candidata de $T$, como definido na *Eq.8*. É possível perceber que, como não estamos fazendo uso de um parâmetro $q$ dinâmico (isto é, ele não será alterado a depender de qual iteração o algoritmo se encontra), esse vetor não será mais atualizado após essa inicialização. Isso significa que a solução de ranque 1 contida na iteração corrente em $T$ terá sempre o peso descrito pelo primeiro elemento do vetor $\omega$ abaixo, a solução de ranque 2 terá o peso do segundo elemento e assim sucessivamente:

```
Vetor ω de pesos:
   [0.03989423
    0.03969525
    0.03910427
    0.03813878
    0.03682701
    0.03520653
    0.03332246
    0.03122539
    0.02896916
    0.02660852]
```

Dando início efetivamente à primeira iteração do algoritmo, cada uma das 3 formigas irão sortear agora uma das soluções candidatas contidas em $T$ para se basear em sua construção de uma nova solução candidata. Esse sorteio é realizado fazendo uso dos pesos contidos em $\omega$ conforme a função massa de probabilidade descrita na *Eq.9*. Cada formiga então, a partir do resultado desse primeiro sorteio, construirá sua solução candidata, efetuando os passos detalhados na seção anterior. Esse processo se dará através da amostragem de funções Gaussianas definidas pelos parâmetros $\mu^i_l$ e $\sigma^i_l$, descritos respectivamente em *Eq.7* e *Eq.10*, para cada componente da solução candidata.

Na execução da primeira iteração realizada, a primeira formiga sorteou a solução de ranque 8, a segunda a de ranque 5 e a terceira a de ranque 1. As soluções candidatas construídas ao fim do processo descrito por cada uma das três formigas nesta execução foram respectivamente:

```
Soluções candidatas construídas pelas 3 formigas:
         Ant 1 [10.         -9.76583329]
         Ant 2 [10.         -0.13920195]
         Ant 3 [ 5.59529867 -6.45041127]
```

Note que as duas primeiras formigas da lista acima encontraram soluções no limite do espaço de busca definido pelo intervalo *(-10, 10)* nas duas dimensões. Isso se dá pois a implementação do código está limitando os valores amostrados na etapa de construção de solução candidata para nunca saírem do espaço de definição do problema. Nesse caso, valores maiores que 10 e menores que -10 serão sobrescritos para permanecerem dentro da região definida. Esse procedimento de correção foi aplicado no primeiro elemento da solução das duas primeiras formigas acima.

De qualquer modo, após obtidas as 3 novas soluções candidatas, as 3 piores soluções contidas na tabela T são substituídas por elas e a tabela é novamente ordenada conforme o custo dos resultados obtidos quando aplicados à função *Ackley* de objetivo. Após esse processo, a tabela *T* terá a seguinte configuração:

```
Soluções candidatas da tabela T ao fim da primeira iteração:
                    [ 5.87309905 -1.71426564]
                    [-6.12536675 -3.16525605]
                    [-0.08418199  7.96742466]
               Ant 2 [10.         -0.13920195]
                    [-8.32371194 -2.15808729]
               Ant 3 [ 5.59529867 -6.45041127]
                    [ 7.56203801 -6.28944077]
                    [-3.20857547 -9.55395067]
                    [-9.87082762 -5.45177066]
               Ant 1 [10.         -9.76583329]
```

Podemos ver que as soluções geradas formigas se posicionaram em ranques diferentes ao fim da iteração na tabela de soluções *T*, com a solução gerada pela segunda formiga em uma melhor posição, seguida da terceira formiga e, por fim, com o pior ranque, a solução da primeira formiga.

Esse procedimento então se repetirá em toda iteração, onde, em cada uma delas, as 3 piores soluções de *T* serão substituídas pelas soluções novas construídas pelas 3 formigas. Ao final das 100 iterações estipuladas, a tabela *T* se encontrará na seguinte configuração:

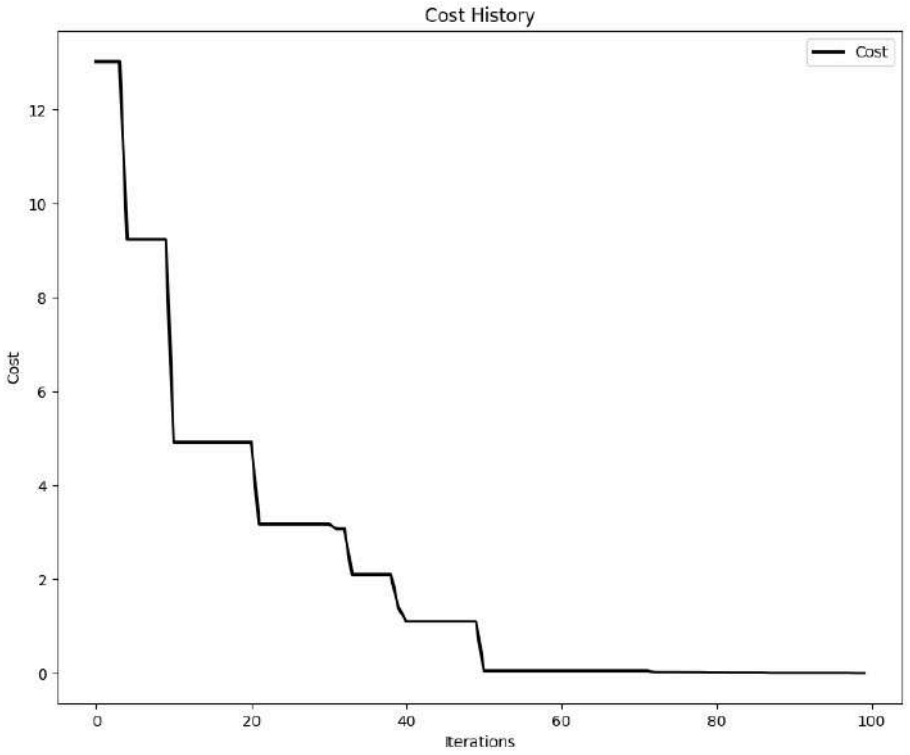
```
Configuração final de soluções candidatas da tabela T:
                [ 0.00040853 -0.00027405]
                [-0.00060742  0.00186385]
                [-0.00127491  0.00151542]
                [-0.00149291  0.00160444]
                [-0.0012322   0.00183804]
                [-0.00175087  0.00139392]
                [-0.00238054 -0.00031149]
                [-0.00268275  0.00162988]
                [-0.00175523  0.00331722]
                [-0.0038199   0.00219554]
```

A melhor solução, de ranque 1, descrita em *T* será a saída do algoritmo $ACO_R$ como ponto de ótimo global encontrado para a função otimizada. De fato, vemos que esta saída se encontra próxima do ponto de ótimo global *(0, 0)* conhecido da função. Quando avaliada na função *Ackley* objetivo temos o seguinte valor de custo:

```
Custo da melhor solução candidata final da tabela T:
0.00139786
```

Novamente, como sabemos que a função *Ackley* utilizada possui um valor de mínimo global ótimo igual a 0 na origem, temos que esse mesmo valor de custo também equivale ao erro obtido ao final do processo de otimização. A seguir se encontra descrito o histórico do custo da melhor posição encontrada até então ao longo das 100 iterações que o algoritmo foi executado:

**Gráfico 6:** Histórico do custo da melhor solução conhecida ($ACO_R$)




Apesar do valor do erro obtido ter sido consideravelmente maior que o observado no exemplo de execução do algoritmo PSO, uma escolha adequada de parâmetros pode vir a melhorar drasticamente os resultados de ambos os algoritmos. No capítulo seguinte, nos focaremos em explorar o efeito de diferentes parametrizações nos algoritmos apresentados em maior profundidade.

# 4 ESTUDO DOS PARÂMETROS

Os algoritmos PSO e $ACO_R$ possuem uma vasta variedade de aplicações, com sua eficácia dependendo fortemente dos parâmetros escolhidos para a sua execução. Os parâmetros de cada algoritmo — como o tamanho do enxame, o coeficiente de inércia no PSO e o parâmetro $k$ para definir o tamanho da tabela de soluções $T$ no $ACO_R$ — desempenham um papel crítico na eficácia da busca pela solução ótima de um dado problema. Portanto, é de grande interesse investigar o impacto das diferentes parametrizações nesses algoritmos e entender como eles afetam a sua performance para diferentes funções a serem otimizadas. A obtenção do melhor conjunto possível de parâmetros é um problema extremamente difícil, tanto por sua alta dependência nas características da função a ser otimizada quanto pelo alto número de parâmetros dessas meta-heurísticas a serem decididos.

Dada tamanha dificuldade, a proposta deste capítulo será então nos limitar a realizar uma investigação preliminar sobre o efeito de diferentes parametrizações nos algoritmos PSO e $ACO_R$ conforme apresentados. O objetivo principal será construir uma intuição sobre a influência de cada um dos diferentes parâmetros, explorando seu impacto no comportamento da otimização resultante. Dada essa meta, buscaremos não assumir um conhecimento prévio da função a ser otimizada (com a exceção de sua dimensão, com esta sendo sempre igual a 2 no estudo realizado, e a região do espaço em que a busca será realizada, para fins práticos), limitando nossas observações desse modo com o fim de tentar obter conclusões generalizáveis e de mais fácil interpretação.

## 4.1 ESTUDO DOS PARÂMETROS DO PSO

O algoritmo PSO, como apresentado no capítulo 2, possui os seguintes parâmetros a serem definidos:

- Número total de partículas.
- Fator de inércia, definindo a influência do vetor velocidade corrente sobre a atualização da coordenada da partícula.
- Coeficiente cognitivo e coeficiente social, indicando respectivamente a influência na atualização da posição pela melhor posição encontrada pela própria partícula e a melhor posição encontrada por seus vizinhos.
- Posições iniciais de cada partícula.
- Topologia do enxame, isto é, a definição do conjunto de partículas consideradas vizinhas durante a etapa de troca de informações.

Visto que nossa proposta será tentar nos aproximar de uma parametrização candidata agnóstica à função a ser otimizada, iremos nos limitar a considerar apenas alguns dos parâmetros listados acima. Em especial iremos considerar que as posições iniciais de cada partícula serão determinadas de maneira uniformemente aleatória na inicialização do algoritmo dentro do espaço de busca do problema. Do mesmo modo também não nos preocuparemos com a seleção de uma topologia específica, simplesmente fazendo uso de uma topologia completa, onde toda partícula se comunicará com todas as demais partículas. Isso significa que nesta implementação, toda partícula terá conhecimento da posição do melhor global obtido pelo conjunto total de partículas. Essa opção simplificará o estudo do impacto

dos demais parâmetros do PSO, considerando a alta dependência da topologia aos demais parâmetros a serem estudados na determinação de seu impacto na performance do algoritmo.

Dadas essas considerações, nos focaremos então a considerar o impacto de diferentes valores apenas para os parâmetros de **número de partículas**, **fator de inércia**, **coeficiente cognitivo** e **coeficiente social** na otimização por PSO.

### 4.1.1 Número de Partículas

Começando pela decisão do número de partículas a serem instanciadas no espaço de busca, algumas observações preliminares devem ser feitas.

Primeiramente, este parâmetro, diferente dos demais que veremos mais à frente, se diferencia por afetar diretamente no custo computacional da execução do algoritmo de otimização. Quanto maior o número de partículas, mais memória será alocada e também mais computações serão realizadas por passo de iteração para a atualização da posição e velocidade de todas as partículas. Visto que essa atualização dependerá apenas de resultados obtidos na iteração anterior, é possível atenuar o custo computacional paralelizando essa etapa por iteração. Essa técnica, porém, por não ser trivialmente aplicável dependendo da implementação e dimensão da função a ser otimizada, não será considerada nos resultados obtidos nesse capítulo (o mesmo será verdade para o caso do $ACO_R$ apresentado a seguir), visto que o objetivo proposto é uma investigação apenas sobre o impacto funcional dos diferentes parâmetros desses algoritmos em sua execução. A paralelização dessa etapa de atualização das novas posições e novos vetores de velocidade para cada partícula (e analogamente no $ACO_R$ a geração de novas soluções candidatas por cada formiga) ofereceria um ganho fixo de aproximadamente *n* vezes maior velocidade no tempo de processamento gasto nessa etapa, sendo *n* o número de processadores empregados.

Um segundo ponto a ser observado sobre o número de partículas a ser decidido envolve o tamanho da região do espaço de busca, isto é, o intervalo da função a ser otimizada em que esta é definida. Uma possível intuição inicial que se pode ter é que uma região de tamanho maior demandará mais partículas para a realização da busca pelo ponto ótimo global. Contudo, assumindo novamente um desconhecimento sobre a função a ser otimizada, isso não será necessariamente o caso. Uma região de busca extensa não implica na necessidade de mais partículas para a busca ser realizada com sucesso. Um contra-exemplo que refuta essa possível inclinação é apresentado a seguir.

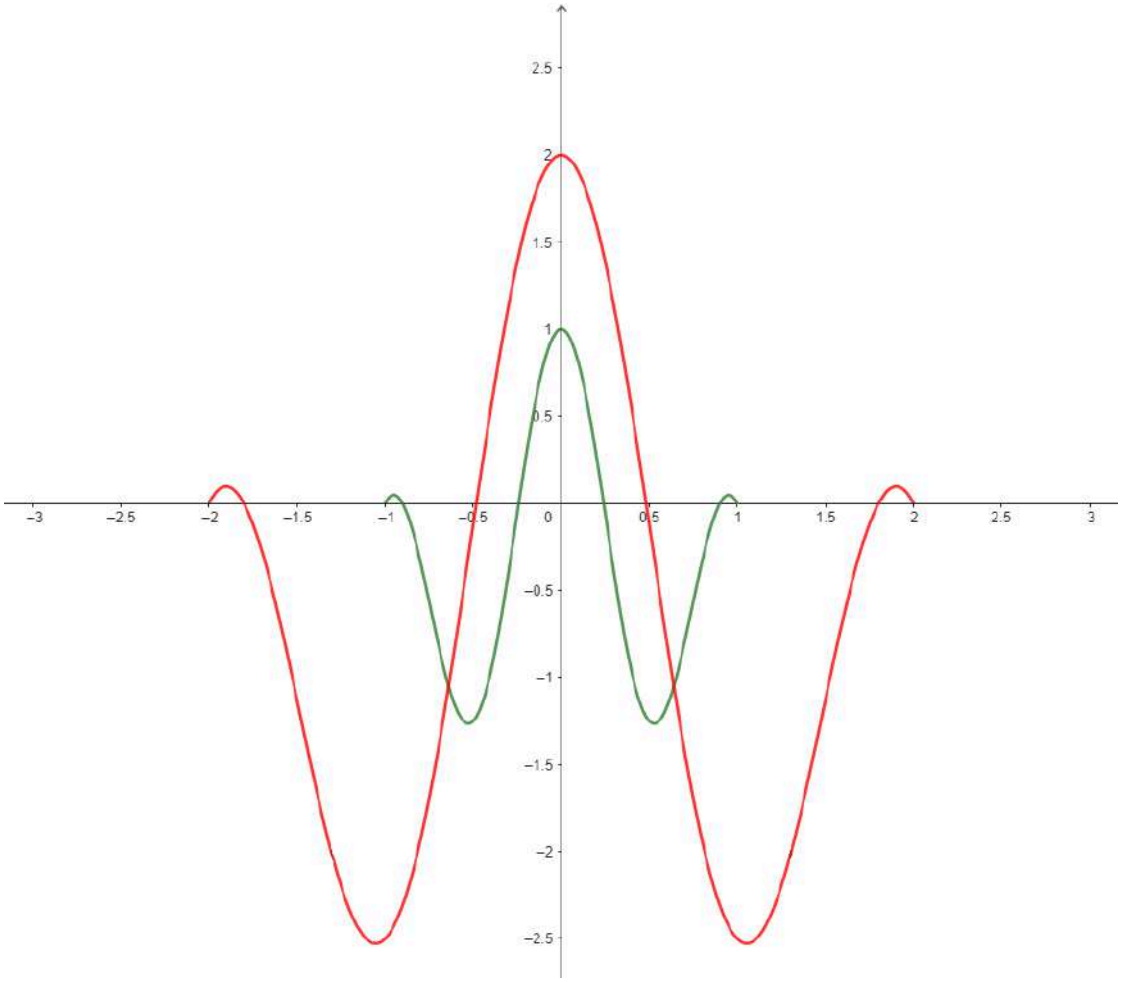
Considerando uma função arbitrária unidimensional *f(x)* definida no intervalo *[a, b]*, podemos gerar infinitas outras funções *g(x)* com seu mesmo formato escalando-a "horizontalmente" por um fator arbitrário *k*, e "verticalmente" por um outro fator arbitrário *z* através do seguinte procedimento:

$$f(x)\ definida\ em\ [a, b]$$

$$g(x) = z \cdot f(\frac{x}{k})\ definida\ em\ [k \cdot a,\ k \cdot b]$$

Percebe-se que o intervalo em que a função é definida também escalou por um fator *k* (devido à extensão realizada no eixo da variável *x*), indicando uma expansão no espaço de busca que deverá ser considerado pelo algoritmo de busca. É válido ressaltar que esse procedimento pode ser estendido para funções com maiores dimensões. Abaixo um exemplo ilustrativo desse procedimento de mudança de escala aplicada em uma função arbitrária unidimensional:

**Gráfico 7:** Funções com regiões de busca diferentes equivalentes para o PSO




*Em verde, a função f(x) = cos(2π) - $x^2$ limitada ao intervalo [-1, 1]. Em vermelho, a mesma função estendida por k = 2 e z = 2 segundo o procedimento descrito anteriormente, isto é, g(x) = 2f(x/2) limitada ao intervalo [-2, 2].*

O importante a ser notado nesse exemplo é que mesmo considerando a expansão do espaço de busca, mantendo as mesmas posições iniciais relativas (apenas tomando o cuidado de também escalar suas coordenadas pelo fator *k* quando traduzindo as posições iniciais de uma função para outra) e os demais parâmetros fixos, percebe-se que não haverá mudança alguma no comportamento do algoritmo PSO em uma busca partindo dessas condições iniciais nas duas funções.

Podemos ver isso ao considerar a primeira iteração do algoritmo na função escalada pelo fator *k*. Teremos que uma partícula terá sua velocidade inicializada pela primeira vez conforme a seguinte fórmula (considerando $p_1$, $p_{best\_1}$ e $g_{best\_1}$ como sendo os valores dessa partícula na primeira iteração da função original, na fórmula a seguir todos estes fatores se encontram multiplicados por *k* pela escala realizada no eixo da solução da função):

$$v_1 = \varphi_p r_p (kp_{best_1} - kp_1) + \varphi_g r_g (kg_{best_1} - kp_1)$$

$$v_1 = k(\varphi_p r_p (p_{best_1} - p_1) + \varphi_g r_g (g_{best_1} - p_1))$$

Nota-se que a velocidade da partícula será exatamente a mesma que a velocidade original da função antes do escalonamento, apenas multiplicada pelo próprio escalar *k*. Essa

diferença garantirá que a partícula irá percorrer uma distância *k* vezes maior no espaço de busca em um único passo de iteração, removendo o possível impacto da extensão do comprimento total do espaço de busca pelo fator *k* no procedimento do processo de otimização. Não só isso, mas para o decorrer do processo de otimização o fator *z* escolhido se demonstra completamente irrelevante, podendo assumir qualquer valor sem afetar a convergência da busca do algoritmo.

Vemos então que essa possível preocupação referente ao tamanho do espaço de busca no momento da escolha do número de partículas não se concretiza na execução do algoritmo, assumindo o desconhecimento da função. Mesmo mantendo o número de partículas, podemos conceber uma função com um espaço de busca arbitrariamente maior que a original, onde a performance do PSO não será impactada. Isso significa que ao atacar um problema com uma extensa região de espaço de busca, esse fato por si só não deve ser considerado um indicador válido para a utilização de um número maior de partículas no algoritmo. O fator determinante principal será a complexidade da malha da função objetivo (com indicadores de alta complexidade sendo, por exemplo, a quantidade de pontos de ótimo local), não o tamanho de sua região de busca.

Tomando então o objetivo proposto de investigar o impacto de diferentes parametrizações sem levar em consideração a função a ser otimizada, a escolha do número de partículas dependerá nessa etapa somente no balanceamento do impacto previsto pelo custo de performance computacional com o ganho oferecido na busca por mais partículas no sistema. Outros fatores se demonstram altamente dependentes da função do problema em questão, não podendo ser tomadas de maneira informada a priori como buscamos.

### 4.1.2 Fator de Inércia

O segundo parâmetro do PSO que iremos avaliar seu impacto será o fator de inércia. Esse fator, como descrito no algoritmo apresentado no capítulo 2, descreve o quanto a velocidade atual de cada partícula influenciará em sua nova velocidade calculada.

Em um primeiro momento, observa-se que o fator de inércia do PSO deverá ter um valor menor que 1, caso contrário garantidamente não ocorrerá convergência dos resultados do conjunto de partículas instanciadas. Isso pode ser verificado, ao notar o exemplo da operação de atualização da velocidade de uma partícula arbitrária em uma iteração que não seja a primeira do algoritmo (na primeira iteração, $v_0$ será igual a 0 e o fator de inércia *w* não fará efeito):

$$v_i \leftarrow wv_{i-1} + influência\ da\ iteração\ atual$$

Para valores de *w* maiores ou iguais a 1, observa-se que o vetor de velocidade crescerá sempre mais na direção do vetor de velocidade da iteração prévia do que na direção indicada pelas informações coletadas pelo conjunto de partículas na iteração atual. Esse fato acarretará ou na dispersão das partículas no espaço de busca (com suas velocidades crescendo descontroladamente em diferentes direções dadas suas posições iniciais), ou em um movimento periódico perpétuo pelo conjunto de partículas no espaço de busca. Em ambos os casos, não havendo convergência do enxame no ponto ótimo.

Um exemplo simples desse movimento periódico pode ser observado ao considerar uma função de apenas uma dimensão, com ponto ótimo em 0 e somente duas partículas no enxame com uma delas já posicionada no ponto ótimo. Considerando um fator de inércia igual a 1, a outra partícula acelerará em direção à partícula na posição ótima, passando direto

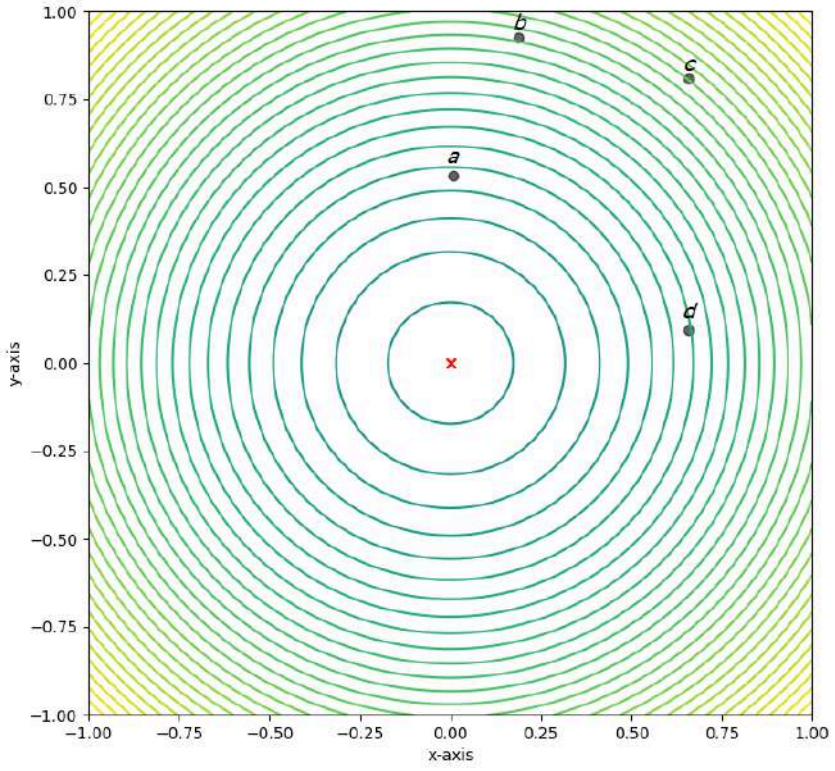
por ela até atingir a mesma distância que originalmente existia entre elas. Isso se repetirá ad infinitum (assumindo que a função seja simétrica em torno do ponto ótimo).

Considerando agora um valor de *w* negativo, podemos ver que este não se adequa ao conceito proposto pela meta-heurística de modelar o movimento de partículas no espaço de busca. Ao atribuir a velocidade nova calculada com um valor de *w* negativo, a partícula poderá "pular" no espaço, alternando sua posição entre duas regiões a cada iteração. Esse comportamento é o resultado da mudança de sinal ocasionada na multiplicação do escalar w negativo no vetor $v_{i-1}$ (velocidade da iteração anterior) na atualização da velocidade $v_i$ (da iteração atual). O vetor atualizado com o possível sinal alternado será então atribuído na posição da partícula somado ao vetor da posição atual dela, conforme descrito na *Eq.2*, resultando assim nesses possíveis "saltos" das partículas no espaço de busca.

O comportamento descrito, além de não representar satisfatoriamente o movimento de partículas de enxame no espaço, também não garante a convergência do algoritmo (apesar de ainda ser possível em certas situações). Desse modo, valores negativos de *w* não são considerados como candidatos na parametrização do PSO.

Por fim, exploraremos como o algoritmo decorre a partir de um fator de inércia *w* fixado no valor 0. Para ilustrar o comportamento dessa parametrização, consideremos como exemplo a função $f(x, y) = x^2 + y^2$, isto é, um paraboloide de revolução com mínimo global em *(0, 0)*. Limitaremos o espaço de busca nesse exemplo para o intervalo *[-1, 1]* em ambas dimensões do domínio para simplificar a inicialização das partículas no espaço de busca (podemos interpretar a função como uma semi-esfera de raio 1). Inicializaremos então nessa região do espaço 4 partículas, com suas devidas posições decididas de forma uniformemente aleatória:

**Gráfico 8:** Posições iniciais de 4 partículas no espaço de busca




Na configuração inicial obtida acima, temos que a partícula *a* se encontra mais próxima ao ponto ótimo *(0, 0)*. Isso significa que as demais partículas se moverão em direção

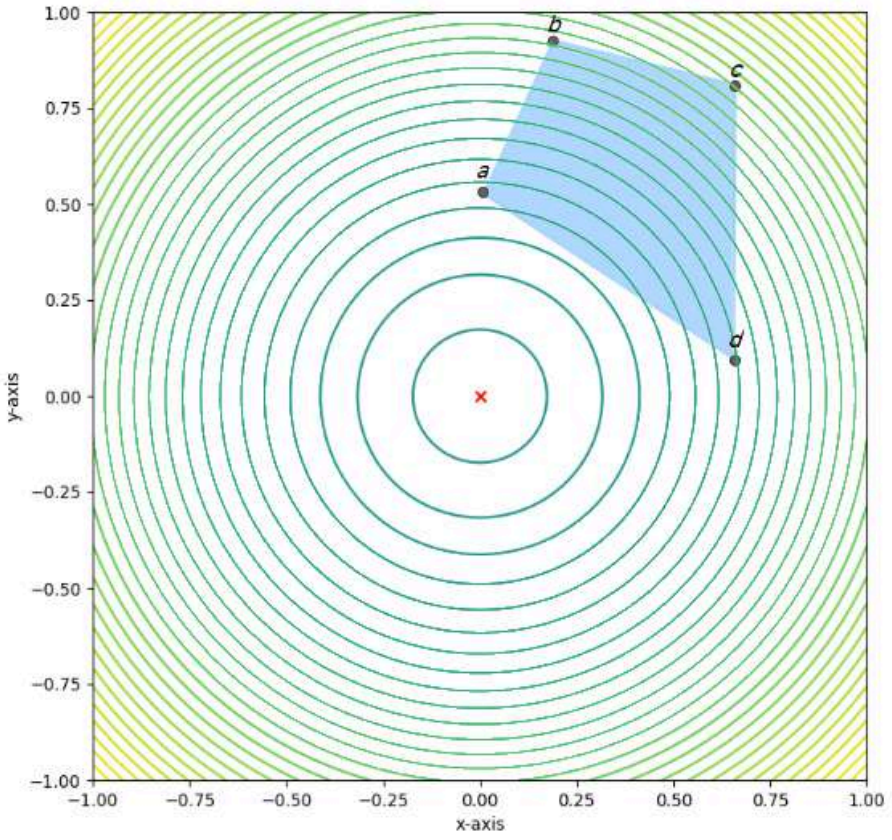
a ela, enquanto que a partícula *a* em si permanecerá parada. Isso ocorrerá visto que sua posição $p_1$, sua melhor posição $p_{best\_1}$ e o melhor global $g_{best\_1}$ terão o mesmo valor:

$$v_1 = r_p(p_{best_1} - p_1) + \varphi_g r_g(g_{best_1} - p_1)$$

$$v_1 = 0$$

Conforme as demais partículas se aproximam da partícula *a*, suas velocidades irão diminuir progressivamente até convergirem com a posição da melhor partícula. Isso ocorre pois, não havendo fator de inércia, a velocidade atual de qualquer partícula em movimento não influenciará no cálculo de sua nova velocidade obtida no passo de iteração corrente. Esse comportamento ocorre pois a velocidade nova da partícula em curso nunca será maior que o vetor distância $g_{best\_i}$ - $p_i$ entre a partícula em questão e a partícula na melhor posição conhecida (considerando um coeficiente social $\varphi_g$ definido em *(0, 1]*). Desse modo, as partículas em movimento nunca irão ultrapassar a posição da melhor partícula, o que significa que a região que a busca será realizada irá se limitar àquela região definida pelas coordenadas que as partículas se inicializaram no começo da execução do algoritmo:

**Gráfico 9:** Região do espaço definida entre as 4 partículas




*Dado um fator de inércia w = 0 e considerando um coeficiente social $\varphi_g$ definido em (0, 1], a busca pelo ponto de mínimo global será realizada somente na região compreendida pelas coordenadas das posições iniciais das partículas no espaço de busca. No exemplo acima, somente na região azul do espaço.*

Note que se, por acaso, o ponto ótimo global estivesse compreendido dentro da região destacada acima, o algoritmo o poderia encontrar sem grandes problemas. Isso se daria pois enquanto as demais partículas se dirigem em direção à posição da partícula *a* (nesse processo de busca, progressivamente reduzindo a área azul) elas provavelmente viriam a encontrar uma nova posição melhor que a posição obtida na primeira iteração por *a*. A posição dessa outra

partícula então se tornaria a melhor posição conhecida e as demais partículas viriam a se direcionar a caminho dela, podendo vir a repetir esse processo diversas vezes, melhorando a precisão da solução.

Sobre o fator de inércia *w* então, podemos tirar a conclusão que sua definição no intervalo *(0, 1)* possui a função de expandir a região de exploração no espaço de busca sem prejudicar de maneira destrutiva a convergência do PSO. Para valores muito pequenos de *w*, os movimentos das partículas tenderão à explotação do melhor valor já visitado, minimizando movimentos extras além do espaço já conhecido. Já valores maiores garantem maior exploração do espaço, com um possível custo maior na quantidade de iterações necessárias para a convergência no resultado ótimo.

### 4.1.3 Coeficientes Cognitivo e Social

Por fim, exploraremos o efeito da escolha dos coeficientes cognitivo $\varphi_p$ e social $\varphi_g$ na parametrização do PSO. Como apresentados no capítulo 2, esses coeficientes se encarregam de balancear, respectivamente, a influência da melhor posição $p_{best}$ já visitada pela partícula e a influência da melhor posição $g_{best}$ conhecida pelo conjunto de partículas vizinhas na atualização de sua velocidade. Visto que estamos considerando uma topologia completa, onde todas partículas são vizinhas de todas as demais, teremos que $g_{best}$ será a melhor posição já visitada por qualquer partícula do sistema.

Essa função de balanceamento entre duas influências antagônicas na atualização da velocidade das partículas faz com que a razão $\varphi_p / \varphi_g$ entre esses dois coeficientes descreva a preferência que uma dada partícula terá para a melhor posição conhecida quando comparada com a melhor posição já visitada por ela própria. Isso significa que um mesmo escalar *k* arbitrário multiplicado em ambos $\varphi_p$ quanto $\varphi_g$ não afetará essa razão, resultando apenas em um salto *k* vezes maior na atualização da posição a partir da nova velocidade, como demonstrado a seguir:

$$v_i = wv_{i-1} + k\varphi_p r_p(p_{best_i} - p_i) + k\varphi_g r_g(g_{best_i} - p_i)$$

$$v_i = wv_{i-1} + k(\varphi_p r_p(p_{best_i} - p_i) + \varphi_g r_g(g_{best_i} - p_i))$$

Percebe-se que o vetor velocidade $v_{i-1}$ da iteração anterior, encontrado acima multiplicado pelo fator de inércia *w*, também conterá o escalar *k* dessa mesma forma em sua fórmula, ocasionado pelo caráter recursivo da definição das velocidades computadas em iterações sequenciais. A função recursiva que descreve o vetor velocidade $v_i$ para uma iteração *i* arbitrária se encontra definida a seguir:

$$v_0 = 0$$

$$v_i = wv_{i-1} + k(\varphi_p r_p(p_{best_i} - p_i) + \varphi_g r_g(g_{best_i} - p_i))$$

A função acima pode ser expressada explicitamente através do seguinte somatório:

$$v_i = \sum_{j=1}^{i} w^{j-1} k(\varphi_p r_p(p_{best_j} - p_j) + \varphi_g r_g(g_{best_j} - p_j))$$

$$v_i = k \sum_{j=1}^{i} w^{j-1}(\varphi_p r_p(p_{best_j} - p_j) + \varphi_g r_g(g_{best_j} - p_j))$$

Temos que o escalar *k* pôde ser isolado na fórmula, indicando que o vetor velocidade terá seu módulo modificado diretamente pelo fator *k* em todo passo de iteração.

Observamos então dois impactos distintos que a escolha desses parâmetros terá para o funcionamento do algoritmo PSO. O primeiro envolve a razão entre os dois coeficientes, que descreve a influência que um terá em relação ao outro nas diferentes iterações do algoritmo. O segundo, como acaba de ser demonstrado, se dá na capacidade de estender ou reduzir o tamanho do vetor de velocidade obtido a cada iteração, alterando o tamanho de cada salto de partícula no espaço de busca por um fator fixo *k* arbitrário, multiplicado em ambos coeficientes.

Contudo, assim como os demais parâmetros explorados neste capítulo, uma análise a priori de seu impacto como a apresentada não nos permitirá decidir um valor sem o conhecimento prévio do problema a ser otimizado. É válido mencionar que, em muitos casos, até mesmo com esse conhecimento isso não será possível. Isso pode se dar tanto pela dificuldade de obter uma descrição completa da função a ser otimizada, quanto em casos onde uma fórmula é conhecida mas, devido a sua alta complexidade, a obtenção do conjunto de parâmetros ótimos se demonstra inviável. Assim, uma abordagem metodológica diferente deve ser tomada e estudos empíricos devem ser realizados especificamente aplicados na função a ser otimizada.

## 4.2 ESTUDO DE PARAMETRIZAÇÕES DO $ACO_R$

Partiremos agora para uma exploração de diferentes parametrizações do algoritmo $ACO_R$, implementado conforme a descrição contida no fim do capítulo 2. Devido a sua similaridade conceitual com a meta-heurística ACO, em determinados pontos nos preocuparemos também em estabelecer analogias entre o impacto de diferentes parametrizações do $ACO_R$ com a influência de parâmetros comparáveis no algoritmo original para domínios discretos. Com isso em mente, os parâmetros da meta-heurística $ACO_R$ que nos encarregaremos de avaliar serão:

- Tamanho *k* da tabela de soluções *T*.
- Número de formigas.
- Parâmetro *q*, utilizado na definição do vetor de pesos $\omega$.
- Parâmetro $\xi$, incluído no cálculo do desvio padrão $\sigma^i_l$.

Assim como foi realizado para o PSO, nos permaneceremos considerando uma exploração do espaço de parâmetros agnóstica à uma função específica a ser otimizada, com o fim de desenvolver intuições generalizáveis sobre o impacto de diferentes parametrizações no algoritmo. Desse modo, diferenças de implementação que dependem da função a ser otimizada serão novamente tomadas como dadas. Essas diferenças são o número *n* de colunas da tabela *T* (fator dependente da dimensão do domínio da função a ser otimizada) e o procedimento de inicialização da tabela T, onde consideraremos uma amostragem uniformemente aleatória na região do espaço de busca para a inicialização de cada solução candidata em *T*.

### 4.2.1 Tamanho *k* da Tabela de Soluções *T*

Tendo em mente que *k* descreve a quantidade de linhas da tabela de soluções *T*, podemos perceber que o parâmetro influenciará na quantidade de memória que deverá ser alocada ao longo do funcionamento do algoritmo. No caso do PSO, vimos que o número de partículas possui um impacto similar na memória ocupada. A diferença que temos é que o número de partículas do PSO também possui um impacto na quantidade de operações que serão realizadas por iteração, o que não é o caso no $ACO_R$ para valores de *k* maiores. O parâmetro que reproduz esse comportamento no $ACO_R$, influenciando na quantidade de operações efetuadas por passo de iteração do algoritmo, será o número de formigas, apresentado na seção seguinte. É válido ressaltar que, em relação ao impacto na memória, valores de *k* grandes não terão um custo consequente, estando este apenas na ordem de $O(nk)$, sendo *n* a dimensão do domínio da função a ser otimizada.

Um segundo ponto a ser notado, esse de maior impacto na performance do resultado da otimização do algoritmo, se refere à maior complexidade conferida para as diferentes funções kernel Gaussianas descritas por cada coluna da tabela de soluções *T* para maiores valores de *k*. Como já descrito, temos que cada uma das *n* variáveis do domínio a ser otimizada terá em um dado momento na tabela *T* uma função kernel Gaussiana $G^i(x)$ ($i = 1, \ldots, n$) atrelada a ela. Cada uma dessas funções, por sua vez, serão compostas por *k* distintas funções Gaussianas $g^i_l(x)$ ($l = 1, ..., k$), uma para cada linha da tabela *T*.

Isso significa que para valores de *k* maiores, a função kernel Gaussiana $G^i(x)$ obtida a partir do conjunto de funções $g^i_l(x)$, para $l = 1, ..., k$, será mais complexa, podendo possuir até *k* distintos pontos de máximo local em sua função de densidade de probabilidade, como apresentado no *Gráfico 1*. O impacto dessa maior complexidade no algoritmo pode ser compreendido pelo fato de que cada solução candidata contida na tabela *T* terá então uma certa influência na decisão de cada variável no processo de amostragem a partir de sua respectiva função kernel Gaussiana realizada por cada formiga. Desse modo, a tabela de soluções *T* descreve, para cada variável a ser otimizada em sua respectiva coluna, uma função kernel Gaussiana contendo a informação dos *k* distintos pontos ótimos candidatos conhecidos.

Esse ganho no poder descritivo da tabela *T*, conquistado armazenando um histórico maior de soluções candidatas conhecidas em memória, é bastante vantajoso em funções complexas com múltiplos pontos de ótimo local em seu domínio pelos motivos discutidos anteriormente. Considerando então casos de funções mais simples, onde valores de *k* grandes se demonstrariam desnecessários, poderíamos pensar que seria válida uma preocupação com a possibilidade de ocorrer uma diluição do impacto de soluções melhores em relação ao conjunto total de soluções candidatas contidas em *T*. Apesar disso poder de fato ocorrer, é possível fazer uso do parâmetro *q* do $ACO_R$ para realizar esse ajuste na preferência por melhores soluções mesmo em tabelas de soluções *T* extensas, contornando o risco de diluição de soluções próximas à ótima. Os impactos do parâmetro *q* no algoritmo $ACO_R$ serão apresentados mais a fundo na seção 4.2.3 deste capítulo.

### 4.2.2 Número de Formigas

O número de formigas, como discutido no começo da seção anterior, será o principal parâmetro que definirá o custo computacional de cada iteração do algoritmo. Quanto maior o número de formigas, maior a quantidade de novas soluções candidatas serão geradas por

iteração, o que por sua vez significa mais componentes de soluções candidatas amostrados no total durante o processo de construção realizado pelas formigas. É importante ressaltar que esse custo maior por iteração não significa que necessariamente o algoritmo gastará mais processamento no total ao longo de sua execução para um número maior de formigas. Isso pode vir a ocorrer visto que o algoritmo poderá convergir na solução ótima em menos passos de iteração nessa situação. Dada uma condição de parada que não envolva exclusivamente o total de iterações realizadas, um aumento no número de formigas pode vir a trazer ganhos de performance por poupar no número de iterações necessárias para convergência.

Uma característica importante da influência desse parâmetro que o distingue, por exemplo, do parâmetro de número de partículas do PSO, e que também contribuirá no processo de convergência do algoritmo, é o fato que a decisão do número de formigas no $ACO_R$ terá um efeito análogo à decisão dos parâmetros encarregados em regular a atualização dos feromônios no algoritmo ACO padrão. Isso se dá pois, no algoritmo $ACO_R$, o processo de evaporação de soluções candidatas de baixa qualidade se dá simplesmente substituindo na tabela $T$ as $m$ piores soluções pelas $m$ novas soluções obtidas na iteração atual, sendo $m$ o número de formigas. Desse modo, esse parâmetro influenciará tanto no processo de evaporação quanto no processo de atualização do conjunto de soluções candidatas em memória. Uma outra consequência desse procedimento de atualização da tabela de soluções $T$ é que o número de formigas será limitado pelo número de soluções candidatas $k$ comportado na tabela $T$.

### 4.2.3 Parâmetro *q*

O parâmetro $q$, por sua vez, terá a função de regular os diferentes valores contidos no vetor $\omega$ de pesos inicializado no começo da execução do algoritmo. Temos que para uma solução de ranque $l$ (as soluções na tabela $T$ são avaliadas na função objetivo e ordenadas, com a melhor tendo o menor ranque), o elemento $\omega_l$ do vetor de pesos $\omega$ respectivo a essa solução será definido seguindo a seguinte fórmula:

$$\omega_l = \frac{1}{qk\sqrt{2\pi}} e^{\frac{-(l-1)^2}{2q^2k^2}}$$

Vimos que essa expressão nada mais é do que o valor da função de densidade de probabilidade de uma Gaussiana em $l$ com média igual a 1 e desvio padrão igual a $qk$.

Recapitulando a maneira que a amostragem da função kernel Gaussiana $G^i$ é realizada no algoritmo apresentado, temos que esse processo é realizado em duas etapas. Na primeira, o vetor $\omega$ é utilizado para decidir qual das $k$ soluções candidatas contidas na tabela $T$ será utilizada pela formiga nessa iteração para ela se basear na construção de sua nova solução candidata. A probabilidade de uma formiga decidir se basear em uma solução de ranque $l$ é descrita pela seguinte fórmula:

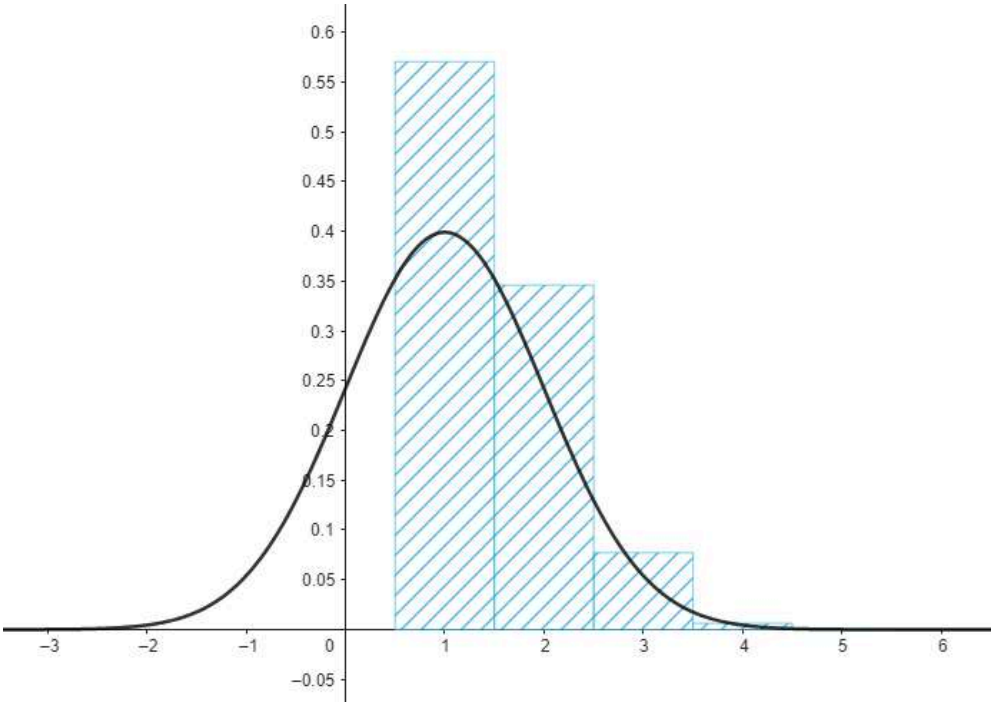
$$p_l = \frac{\omega_l}{\sum_{r=1}^{k} \omega_r}$$

O impacto do parâmetro $q$ na implementação da meta-heurística $ACO_R$ apresentada se dá no controle do formato da função de massa de probabilidade descrita na primeira etapa do procedimento. Vimos que a distribuição discreta que selecionará o ranque da solução candidata contida em $T$ na primeira etapa é definida a partir de uma função Gaussiana em $l$ com média igual a 1 e desvio padrão igual a $qk$. Isto é, a melhor solução (de ranque 1) terá a

maior chance de ser selecionada, se encontrando no ponto máximo da curva de distribuição normal.

A seguir se encontra uma visualização da função Gaussiana obtida e da função de massa de probabilidade resultante dela, aplicando o procedimento descrito na primeira etapa de amostragem, para uma parametrização arbitrária com $k = 5$ e $q = 0,2$:

**Gráfico 10:** Função de massa de probabilidade para as 5 soluções em T ($q$ = 0,2)

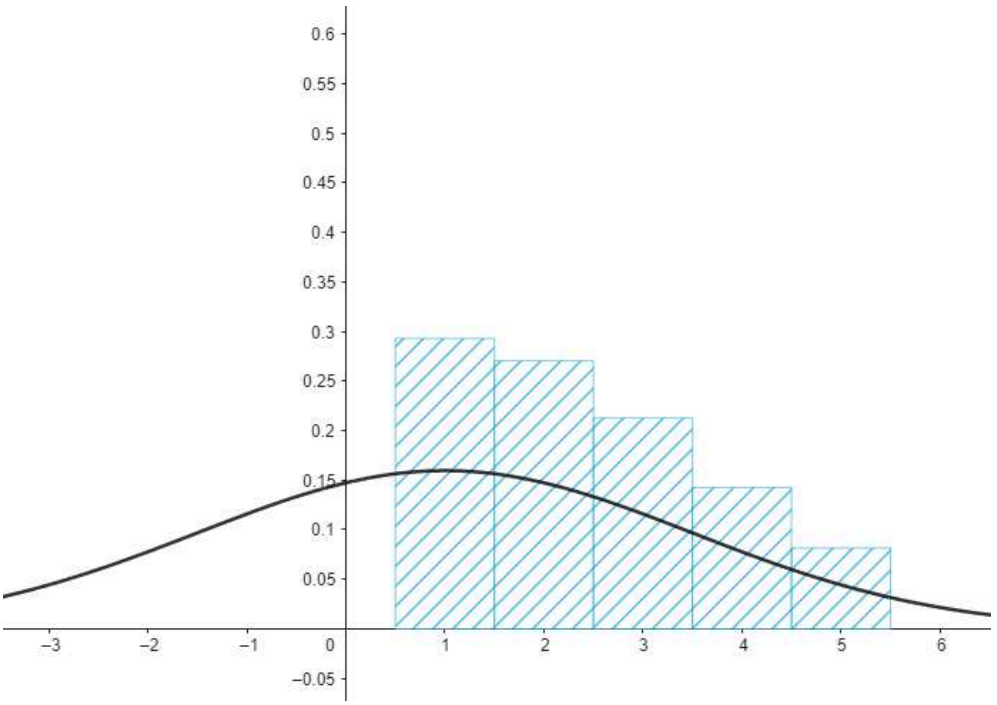



*A função Gaussiana (de média 1 e desvio padrão kq) resultante de uma parametrização com k = 5 e q = 0,2 se encontra em preto. O histograma em azul representa a função de massa de probabilidade para as 5 soluções candidatas contidas na tabela de solução T, obtida a partir da discretização e normalização da função Gaussiana.*

Para essa escolha de parâmetros, temos que a probabilidade da solução com ranque 1 em $T$ ser escolhida nesta iteração por uma formiga será de aproximadamente *0,57*. Em comparação a pior solução, de ranque 5, contida na tabela de soluções terá uma probabilidade de aproximadamente apenas *0,00019* de ser escolhida.

Mantendo ainda o valor de $k = 5$, vamos agora observar a consequência de uma parametrização de $q = 0.5$ nas probabilidades:

**Gráfico 11:** Função de massa de probabilidade para as 5 soluções em T ($q$ = 0,5)




*A função Gaussiana resultante de uma parametrização ainda com k = 5, mas agora com q = 0,5. Uma distribuição mais justa da função de massa de probabilidade entre as 5 soluções candidatas em T pode ser observada quando comparada com o resultado obtido anteriormente.*

Como previsto pela fórmula de sua função de densidade de probabilidade, esse aumento no valor do parâmetro $q$ promoveu um aumento proporcional no desvio padrão da função Gaussiana. Por consequência disso, a função de massa de probabilidade gerada a partir da distribuição contínua também apresentou uma desconcentração na probabilidade de soluções com os melhores ranques serem escolhidas para o restante. Nessa nova parametrização com $q = 0,5$, a solução com ranque 1 em $T$ possui agora uma probabilidade de aproximadamente apenas $0,29$ enquanto que a pior solução, de ranque 5, apresenta uma probabilidade de aproximadamente $0,081$ de ser escolhida, um salto significativo quando comparada à parametrização anterior. Parâmetros maiores ainda de $q$ resultarão em uma distribuição cada vez mais uniforme na decisão das $k$ soluções candidatas na tabela de soluções $T$.

Em conclusão, a influência do parâmetro $q$ pode ser compreendida como sendo a preferência que o processo de amostragem do componente de solução candidata sendo construída pela formiga terá por soluções próximas àquelas de melhor ranque contidas na tabela $T$. Afinal, essa escolha sobre qual das $k$ soluções candidatas contidas na tabela $T$ será utilizada nessa iteração pela formiga, além de impactar no cálculo do desvio padrão $\sigma^i_l$ na segunda etapa da amostragem, já definirá diretamente a média $\mu^i_l$ que será utilizada junto ao desvio padrão na definição da função Gaussiana $g^i_l$ amostrada ao final deste passo de construção.

### 4.2.4 Parâmetro $\xi$

Continuando a partir do fim da primeira etapa de amostragem, desenvolvida na seção anterior junta à influência do parâmetro $q$ em sua aplicação, seguiremos agora para a segunda etapa do processo de amostragem descrito no capítulo 2 para uma análise do impacto da escolha do parâmetro $\xi$ no $ACO_R$. Nessa segunda etapa, o cálculo do desvio padrão $\sigma^i_l$ referente à função Gaussiana $g^i_l$ é realizado para sua eventual amostragem, sendo $l$ o ranque sorteado na primeira etapa e $i$ sendo o passo de construção da solução candidata que a formiga se encontra:

$$\sigma^{i}_{l} = \xi \sum_{e=1}^{k} \frac{|s^{i}_{e} - s^{i}_{l}|}{k-1}$$

Não levando em consideração o parâmetro $\xi$ no momento, a fórmula acima, já descrita previamente na *Eq.10* e introduzida no artigo de Socha e Dorigo em que o $ACO_R$ foi originalmente proposto (2008, p. 1161), pode ser compreendida como uma simplificação da definição usual de desvio padrão de uma dada população:

$$\sigma(X) = \sqrt{\sum_{n=1}^{m} \frac{(x_n - \mu)^2}{m}}$$

A média $\mu$ contida na fórmula corresponde, no caso do $ACO_R$, à média da função Gaussiana $g^i_l$, já conhecida pela definição apresentada no capítulo 2 como sendo $s^i_l$, valor este contido na tabela $T$ na linha $l$ (sorteada na primeira etapa da amostragem), coluna $i$ (definida pela etapa atual de construção da solução que a formiga se encontra). Do mesmo modo, os $m$ elementos $x_n$ da população na definição acima têm como sua contraparte os $k$ elementos $s^i$ contidos na coluna $i$ da tabela $T$ de soluções.

As diferenças entre as fórmulas se dão pela remoção da raiz e do expoente dentro do somatório, este sendo substituído pelo valor absoluto da subtração em cada parcela do somatório. Essa simplificação é possível ser feita pois, apesar das fórmulas não serem equivalentes, seus valores serão próximos, havendo ainda a possibilidade de ser modificado por uma escolha adequada do parâmetro $\xi$ para melhor desempenho. Uma outra forma de interpretar a fórmula do desvio padrão $\sigma^i_l$ apresentada é como esta sendo simplesmente a distância média entre a solução $s^i_l$ escolhida na primeira etapa do processo de amostragem para as demais soluções da tabela $T$ contidas na mesma coluna $i$.

Considerando agora o parâmetro $\xi$ na fórmula, temos que este fator influenciará diretamente o desvio padrão obtido pelas formigas na definição das funções Gaussianas que serão amostradas. Para valores maiores de $\xi$, maior o grau de dispersão da função Gaussiana $g^i_l$ obtida pela formiga nesse passo de construção. Enquanto que o parâmetro $q$ influencia no grau de dispersão da distribuição utilizada na primeira amostragem para a solução $s^i_l$ da tabela $T$ que servirá como média $\mu^i_l$ da função Gaussiana final, o parâmetro $\xi$ influenciará no grau de dispersão da própria função Gaussiana $g^i_l$ na segunda etapa de amostragem.

Conceitualmente, o parâmetro $\xi$ do algoritmo $ACO_R$ possui um efeito similar ao coeficiente de evaporação de feromônios no algoritmo ACO padrão. Quanto maior o valor de $\xi$, menor a velocidade de convergência do algoritmo devido a sua menor preferência por se manter próximo a valores já visitados. Enquanto que no ACO, o coeficiente de evaporação influencia na memória a longo prazo de iterações passadas (soluções piores são esquecidas mais rápido), $\xi$ no $ACO_R$ altera a maneira que esta memória é utilizada. Para valores maiores

de $\xi$, a busca se torna menos enviesada a preferir pontos no espaço de busca já explorados, isto é, aqueles contidos na tabela de soluções $T$.

Com os valores de $\sigma^i_l$ e $\mu^i_l$ calculados ao fim dessa segunda etapa, a função Gaussiana $g^i_l$ é finalmente amostrada e o componente da solução candidata é obtido pela formiga. Assim, neste procedimento em duas etapas, não efetuamos diretamente a amostragem na função kernel Gaussiana $G^i$. O que realizamos de fato é, no primeiro passo, amostrar uma das $k$ soluções candidatas baseado no vetor de pesos (definido pelo ranque das $k$ diferentes soluções em $T$, junto com o parâmetro $q$) para a seguir, no segundo passo, calcular o desvio padrão $\sigma^i_l$ que será utilizado para amostrar a função Gaussiana $g^i_l$ para a obtenção do componente da solução candidata. Este processo será repetido para todo componente da solução candidata por cada uma das formigas em cada iteração do algoritmo.

A escolha deste parâmetro dependerá fortemente da função do problema de otimização sendo atacado pelo algoritmo. Fora algumas limitações triviais, como por exemplo o fato que o valor de $\xi$ não poder ser negativo para não termos um desvio padrão menor que 0, não será possível nos aproximar de um valor ótimo para a escolha do valor desses parâmetros sem estudos empíricos aplicados diretamente ao problema em questão.

Neste capítulo nos focamos em buscar uma compreensão sobre o impacto dos diferentes parâmetros do PSO e $ACO_R$ de uma maneira generalizável, agnóstica à função a ser otimizada, buscando compreender a influência que diferentes valores exercem no desenvolver dos algoritmos. Contudo um estudo dessa característica não será suficiente para nos aproximar de um conjunto de parâmetros ótimos para uma função arbitrária. Como visto ao longo do capítulo, além de existir uma alta dependência entre o conjunto de parâmetros ótimos e a função a ser otimizada, observamos também uma forte influência entre os próprios valores de diferentes parâmetros no resultado processo de otimização (a exemplo disso, ver o caso dos parâmetros $q$ e $\xi$ do $ACO_R$).

# 5 CONCLUSÃO

Nos capítulos precedentes, diferentes meta-heurísticas bioinspiradas para problemas de otimização foram introduzidas, com um foco especial dado para a implementação dos algoritmos PSO e $ACO_R$, para, em sequência, uma exploração do impacto de diferentes parametrizações desses dois algoritmos ser realizada. Essa exploração, reiterando, tentou se conduzir de forma a não se condicionar em uma dada função objetivo específica, visando se obter uma compreensão generalizável sobre o impacto individual de cada parâmetro na busca efetuada no desenvolver dos algoritmos considerados. Contudo, como notado no decorrer do capítulo anterior, as conclusões obtidas por essa abordagem não se demonstraram suficientemente fortes para uma determinação devidamente aproximada de nenhum dos parâmetros considerados.

Isso ocorre principalmente devido a dois fatores, estes sendo a dependência dos valores ótimos de parâmetros em relação à função objetivo e a interdependência dos parâmetros entre si. Esses dois fatores resultam em um espaço de busca complexo demais para uma aproximação suficientemente vantajosa do conjunto de parâmetros ótimos aprioristicamente como proposto. O melhor que podemos fazer nessa situação preliminar, como foi desenvolvido para os diferentes parâmetros dos dois algoritmos apresentados no capítulo anterior, se resume a obter uma compreensão sobre o impacto observado de valores diferentes dos diversos parâmetros no desenvolver da execução dos algoritmos. Compreensão esta que, apesar de limitada, é essencial para informar a pesquisa no momento em que os estudos empíricos são empenhados de fato.

O espaço de busca de parâmetros dessa classe de algoritmos é altamente complexo, desse modo, para sua exploração em uma etapa empírica futura ser realizada com direção e eficiência, esse entendimento do impacto dos diferentes parâmetros nos resultados observados para guiar a busca de maneira informada se demonstra necessário. A exploração empírica de parâmetros adequados voltada para problemas específicos de otimização podem ser encontrados em abundância na literatura, com alguns exemplos incluindo na testagem de software (WINDISCH; WAPPLER; WEGENER, 2007), no desenvolvimento de sistemas hidráulicos de reatores nucleares (MENESES; MACHADO; SCHIRRU, 2009) e em problemas de modelagem para o desenvolvimento de sistemas de abastecimento de água (MONTALVO et al., 2008).

Uma observação interessante a se fazer em relação a essa dificuldade de obtenção de parâmetros ótimos se evidencia na aplicação comum dessa própria classe de algoritmos apresentados na busca do conjunto de parâmetros ótimos em outros sistemas. Um exemplo popular dessa aplicação se dá na busca por hiperparâmetros adequados em sistemas de redes neurais (LORENZO et al., 2017). Isso significa que um possível procedimento para a busca de parâmetros dos algoritmos PSO ou $ACO_R$, possa ser a aplicação do próprio algoritmo para essa busca.

Um ponto importante a se manter em mente sobre essa possível aplicação é que os dois conjuntos de parâmetros utilizados nas duas aplicações (na original e na segunda para a busca dos parâmetros da primeira) necessariamente deverão ser distintos, visto que a função objetiva das duas implementações se diferem (a primeira sendo a função objetiva original e a segunda sendo a eficácia, seja qual for esta métrica, do algoritmo utilizado para otimizar a função objetiva original). A aplicabilidade desta direção, contudo, não foi explorada para os fins deste trabalho, sendo uma possível candidata para pesquisas futuras.

Possíveis trabalhos futuros podem vir a explorar empiricamente o impacto de diferentes parametrizações dos algoritmos apresentados, dessa vez os aplicando diretamente

em problemas de otimização específicos, realizando uma análise quantitativa no resultado da otimização observada. Um segundo caminho de pesquisa possível se dá no estudo de técnicas de busca para a escolha do conjunto ótimo de parâmetros para esses algoritmos, como mencionado, na possível aplicação dos próprios algoritmos PSO e $ACO_R$ na decisão de seus próprios parâmetros.

Em conclusão, neste trabalho exploramos duas poderosas meta-heurísticas bioinspiradas, a Otimização por Enxame de Partículas e a Otimização de Colônia de Formigas, aplicadas ao contexto de otimização para funções de domínios contínuos. Por meio de uma análise detalhada, examinamos suas principais características, mecanismos de funcionamento e estratégias de adaptação. Observamos que esses algoritmos têm demonstrado eficácia em uma ampla gama de problemas de otimização graças à sua capacidade de explorar de forma eficiente um complexo espaço de busca multidimensional. No entanto, também destacamos a importância e dificuldade de considerar a escolha apropriada de parâmetros e estratégias específicas de adaptação para garantir o desempenho ótimo. Em última análise, as meta-heurísticas bioinspiradas oferecem uma abordagem altamente eficiente para a solução de problemas complexos, com uso ubíquo nas mais diversas indústrias. Este estudo visou contribuir para uma compreensão intuitiva do impacto de diferentes fatores na aplicação dessas abordagens, fornecendo uma visão ampla e generalizável dos dois algoritmos apresentados para os mais diferentes tipos de problemas.

# REFERÊNCIAS


CAUNHYE, Aakil M.; NIE, Xiaofeng; POKHAREL, Shaligram. Optimization models in emergency logistics: A literature review. **Socio-economic planning sciences**, v. 46, n. 1, p. 4-13, 2012.

INTRILIGATOR, Michael D. **Mathematical optimization and economic theory**. Society for Industrial and Applied Mathematics, 2002.

MARLER, R. Timothy; ARORA, Jasbir S. Survey of multi-objective optimization methods for engineering. **Structural and multidisciplinary optimization**, v. 26, p. 369-395, 2004.

AMARI, Shun-ichi. Backpropagation and stochastic gradient descent method. **Neurocomputing**, v. 5, n. 4-5, p. 185-196, 1993.

DANTZIG, George B. Linear programming. **Operations research**, v. 50, n. 1, p. 42-47, 2002.

TURING, Alan M. **Computing machinery and intelligence**. Springer Netherlands, 2009.

HOLLAND, John H. **Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence**. MIT press, 1992.

KAR, Arpan Kumar. Bio inspired computing–a review of algorithms and scope of applications. **Expert Systems with Applications**, v. 59, p. 20-32, 2016.

KENNEDY, James; EBERHART, Russell. Particle swarm optimization. **Proceedings of ICNN'95-international conference on neural networks**. IEEE, 1995. p. 1942-1948.

GUDISE, Venu G.; VENAYAGAMOORTHY, Ganesh K. Comparison of particle swarm optimization and backpropagation as training algorithms for neural networks. **Proceedings of the 2003 IEEE Swarm Intelligence Symposium. SIS'03 (Cat. No. 03EX706)**. IEEE, 2003. p. 110-117.

DORIGO, Marco. **Optimization, learning and natural algorithms**. Ph. D. Thesis, Politecnico di Milano, 1992.

STÜTZLE, Thomas et al. ACO algorithms for the traveling salesman problem. **Evolutionary algorithms in engineering and computer science**, v. 4, p. 163-183, 1999.

KANT, Ajay et al. An ACO approach to job scheduling in grid environment. **Swarm, Evolutionary, and Memetic Computing: First International Conference on Swarm, Evolutionary, and Memetic Computing, SEMCCO 2010**, Chennai, India, December 16-18, 2010. Proceedings 1. Springer Berlin Heidelberg, 2010. p. 286-295.

MONTEMANNI, Roberto et al. Ant colony system for a dynamic vehicle routing problem. **Journal of combinatorial optimization**, v. 10, p. 327-343, 2005.

CARO, Gianni; DORIGO, Marco. **Ant colony optimization and its application to adaptive routing in telecommunication networks**. 2004. Ph. D. Thesis, Faculté des Sciences Appliquées, Université Libre de Bruxelles, Brussels, Belgium.

MARZBAND, Mousa et al. Real time experimental implementation of optimum energy management system in standalone microgrid by using multi-layer ant colony optimization. **International Journal of Electrical Power & Energy Systems**, v. 75, p. 265-274, 2016.

FARAHNAKIAN, Fahimeh et al. Using ant colony system to consolidate VMs for green cloud computing. **IEEE transactions on services computing**, v. 8, n. 2, p. 187-198, 2014.

KARABOGA, Dervis et al. A comprehensive survey: artificial bee colony (ABC) algorithm and applications. **Artificial intelligence review**, v. 42, p. 21-57, 2014.

FISTER, Iztok et al. A comprehensive review of firefly algorithms. **Swarm and evolutionary computation**, v. 13, p. 34-46, 2013.

YANG, Xin-She; HE, Xingshi. Bat algorithm: literature review and applications. **International Journal of Bio-inspired computation**, v. 5, n. 3, p. 141-149, 2013.

FISTER, Iztok et al. Cuckoo search: a brief literature review. **Cuckoo search and firefly algorithm: Theory and applications**, p. 49-62, 2014.

SOCHA, Krzysztof; DORIGO, Marco. Ant colony optimization for continuous domains. **European journal of operational research**, v. 185, n. 3, p. 1155-1173, 2008.

WINDISCH, Andreas; WAPPLER, Stefan; WEGENER, Joachim. Applying particle swarm optimization to software testing. **Proceedings of the 9th annual conference on genetic and evolutionary computation**. 2007. p. 1121-1128.

MENESES, Anderson Alvarenga de Moura; MACHADO, Marcelo Dornellas; SCHIRRU, Roberto. Particle swarm optimization applied to the nuclear reload problem of a pressurized water reactor. **Progress in Nuclear Energy**, v. 51, n. 2, p. 319-326, 2009.

MONTALVO, Idel et al. Particle swarm optimization applied to the design of water supply systems. **Computers & Mathematics with Applications**, v. 56, n. 3, p. 769-776, 2008.

LORENZO, Pablo Ribalta et al. Particle swarm optimization for hyper-parameter selection in deep neural networks. **Proceedings of the genetic and evolutionary computation conference.** 2017. p. 481-488.