



UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO, DE CIÊNCIAS EXATAS E EDUCAÇÃO
DEPARTAMENTO DE ENG. DE CONTROLE, AUTOMAÇÃO E COMPUTAÇÃO
CURSO DE GRADUAÇÃO EM ENGENHARIA DE CONTROLE E AUTOMAÇÃO

Marianne Gaertner

**Implementação do Algoritmo de Otimização por Enxame de Partículas
utilizando Aritmética de Ponto Fixo**

Marianne Gaertner

**Implementação do Algoritmo de Otimização por Enxame de Partículas
utilizando Aritmética de Ponto Fixo**

Trabalho de Conclusão de Curso de Graduação em Engenharia de Controle e Automação do Centro Tecnológico, de Ciências Exatas e Educação da Universidade Federal de Santa Catarina como requisito para a obtenção do título de Engenheira de Controle e Automação.

Orientador: Prof. Marcos Vinicius Matsuo, Dr.

Blumenau

2026

Ficha catalográfica gerada por meio de sistema automatizado gerenciado pela BU/UFSC.
Dados inseridos pelo próprio autor.

Gaertner, Marianne

Implementação do Algoritmo de Otimização por Enxame de Partículas utilizando Aritmética de Ponto Fixo / Marianne Gaertner ; orientador, Marcos Vinicius Matsuo, 2026.
59 p.

Trabalho de Conclusão de Curso (graduação) -
Universidade Federal de Santa Catarina, Campus Blumenau,
Graduação em Engenharia de Controle e Automação, Blumenau,
2026.

Inclui referências.

1. Engenharia de Controle e Automação. 2. Otimização por Enxame de Partículas. 3. Aritmética de ponto fixo. 4. Identificação de Sistemas. 5. FPGA. I. Matsuo, Marcos Vinicius. II. Universidade Federal de Santa Catarina. Graduação em Engenharia de Controle e Automação. III. Título.

Marianne Gaertner

Implementação do Algoritmo de Otimização por Enxame de Partículas utilizando Aritmética de Ponto Fixo

Este Trabalho de Conclusão de Curso foi julgado adequado para obtenção do Título de “Engenheira de Controle e Automação” e aprovado em sua forma final pelo Curso de Graduação em Engenharia de Controle e Automação.

Blumenau, 08 de julho de 2026.

Banca Examinadora:



Documento assinado digitalmente

Marcos Vinicius Matsuo

Data: 13/07/2026 08:30:23-0300

CPF: ***,580.029-**

Verifique as assinaturas em <https://v.ufsc.br>

Prof. Marcos Vinicius Matsuo, Dr.
Universidade Federal de Santa Catarina (UFSC)



Documento assinado digitalmente

Janaina Gonçalves Guimarães

Data: 13/07/2026 08:34:47-0300

CPF: ***,475.831-**

Verifique as assinaturas em <https://v.ufsc.br>

Profa. Janaina Gonçalves Guimarães, Dra.
Universidade Federal de Santa Catarina (UFSC)



Documento assinado digitalmente

Sheila Santisi Travessa

Data: 13/07/2026 09:39:53-0300

CPF: ***,584.507-**

Verifique as assinaturas em <https://v.ufsc.br>

Profa. Sheila Santisi Travessa, Dra.
Universidade Federal de Santa Catarina (UFSC)

Dedico este trabalho aos meus pais, ao meu irmão e aos meus colegas, pelo apoio incondicional e pela parceria ao longo desta jornada.

AGRADECIMENTOS

Agradeço, primeiramente, a Deus pela força, saúde e discernimento concedidos ao longo desta jornada acadêmica.

Aos meus pais, Sergio Gaertner e Marise Buzzi Gaertner, que estiveram presentes nos momentos mais importantes da minha vida, pelo apoio incondicional, presença, conselhos e valores que fizeram que eu me tornasse quem sou, deixo minha profunda gratidão. Ao meu irmão, Leonardo Gaertner, pela parceria, conversas e por compartilhar comigo as experiências desta caminhada.

Ao meu *nono* José Ignácio Buzzi (*in memoriam*), que sempre me ensinou sobre o poder da fé e da educação, e cujo exemplo de sabedoria, paciência e felicidade continua a guiar os meus passos, dedico minha eterna gratidão. À minha família estendo meus agradecimentos, pela compreensão das minhas ausências e por fazerem parte de quem eu sou.

Aos meus colegas e amigos que fiz ao longo da graduação, que compartilharam dificuldades e conquistas, obrigada por me incentivarem nos vales e por celebrarem cada um dos picos. Por proporcionarem uma experiência única, repleta de alegrias, e por formarem um grupo que se impulsiona mutuamente para em busca do sucesso.

Ao meu professor orientador, Prof. Dr. Marcos Vinicius Matsuo, pelo contínuo suporte, disponibilidade, dedicação e orientação, pela confiança depositada em mim e pelo auxílio fundamental para o desenvolvimento deste trabalho. Aos membros da banca, agradeço por aceitarem este convite. Aos demais professores e pessoas envolvidas na graduação, obrigada pelos ensinamentos e marcas que me fortaleceram, moldaram e prepararam para o futuro.

A todos que, por alguma razão, participaram dessa jornada comigo e deixaram suas marcas, muito obrigada.

"It is not knowledge, but the act of learning, not possession but the act of getting there, which grants the greatest enjoyment." (GAUSS, 1808)

RESUMO

Este trabalho apresenta a implementação do algoritmo de Otimização por Enxame de Partículas com aritmética de ponto fixo, aplicada à identificação dos parâmetros de uma função de transferência de segunda ordem que rege o funcionamento de um conversor CC-CC do tipo *boost*. O algoritmo é inicialmente implementado em aritmética de ponto flutuante no *Matlab*, etapa que estabelece a referência de comparação para as demais implementações. Em seguida, desenvolve-se uma implementação equivalente em aritmética de ponto fixo, definindo as operações de multiplicação, adição e subtração necessárias para reproduzir o comportamento do algoritmo com palavras de 32 bits com sinal. Adicionalmente, os dois principais blocos computacionais do algoritmo, o cálculo da função custo e a atualização da velocidade e posição das partículas, são também descritos em linguagem de descrição de *hardware* VHDL, visando a futura síntese do algoritmo em dispositivos FPGA. Para a atualização da velocidade e posição, adota-se uma máquina de estados finitos com reutilização sequencial de um único componente multiplicador, estratégia que reduz o consumo de blocos multiplicadores dedicados em relação a uma implementação totalmente paralela, viabilizando a síntese em FPGAs de menor porte. Os blocos desenvolvidos em VHDL são validados por meio de simulação no *ModelSim*, comparando os resultados obtidos com os valores calculados pela implementação de referência em ponto fixo no *Matlab*, para os mesmos estímulos de entrada. A implementação em ponto fixo estima os parâmetros da planta com precisão próxima à obtida pela implementação em ponto flutuante, indicando que a quantização introduzida pela representação em ponto fixo não compromete a capacidade de convergência do algoritmo no cenário testado. Os blocos desenvolvidos em VHDL reproduzem, em *hardware*, os mesmos resultados calculados em *software*, evidenciando a equivalência funcional entre as duas implementações. Os resultados obtidos demonstram a viabilidade da aritmética de ponto fixo para a implementação do algoritmo PSO em sistemas embarcados e estabelece uma base para futura integração dos blocos validados em um sistema completo executado em FPGA.

Palavras-chave: Otimização por Enxame de Partículas; ponto fixo; identificação de sistemas; VHDL; FPGA.

ABSTRACT

This work presents the implementation of the Particle Swarm Optimization algorithm using fixed-point arithmetic, applied to the parameter identification of a second-order transfer function that models a boost DC-DC converter. The algorithm is initially implemented in floating-point arithmetic in Matlab, a step that establishes the comparison reference for subsequent implementations. Next, an equivalent fixed-point implementation is developed, defining the specific multiplication, addition, and subtraction operations required to replicate the algorithm's behavior using signed 32-bit words. Additionally, the two main computational blocks of the algorithm, the cost function calculation and the particle velocity and position update, are also described in VHDL hardware description language, aiming at the future synthesis of the algorithm in FPGA devices. For the velocity and position update, a finite state machine with sequential reuse of a single multiplier component is adopted, a strategy that reduces the consumption of dedicated multiplier blocks compared to a fully parallel implementation, enabling synthesis in smaller FPGAs. The developed VHDL blocks are validated through simulation in ModelSim by comparing their outputs with the values computed by the fixed-point Matlab reference implementation under the same input stimuli. The fixed-point implementation estimates the plant parameters with an accuracy close to that of the floating-point version, indicating that the quantization noise introduced by the fixed-point representation does not compromise the algorithm's convergence capability in the tested scenario. The blocks developed in VHDL reproduce, in hardware, the same results calculated in software, demonstrating the functional equivalence between both implementations. The results obtained demonstrate the viability of fixed-point arithmetic for the implementation of the PSO algorithm in embedded systems and establish a base for future integration of the validated blocks into a complete system executed on an FPGA.

Keywords: Particle Swarm Optimization; fixed-point; system identification; VHDL; FPGA.

LISTA DE FIGURAS

Figura 1 – Fluxograma do algoritmo PSO.	19
Figura 2 – Mecanismo de atualização da velocidade do PSO.	20
Figura 3 – Representação de números negativos em Complemento de 2.	22
Figura 4 – Representação de número em ponto fixo.	23
Figura 5 – Circuito elétrico RLC.	25
Figura 6 – Ensaio experimental para obtenção do modelo matemático.	26
Figura 7 – Estrutura do estimador <i>online</i>	26
Figura 8 – Representações não paramétricas. Resposta ao impulso de (10).	27
Figura 9 – Diagrama de circuito de um conversor <i>boost</i>	28
Figura 10 – Fluxograma geral do algoritmo implementado.	30
Figura 11 – Bloco da função custo.	39
Figura 12 – Bloco de atualização da velocidade e posição das partículas.	41
Figura 13 – Bloco multiplicador.	42
Figura 14 – Mapeamento e fatiamento dos vetores de entrada conforme o sinal FLAG	43
Figura 15 – Máquina de Estados do bloco <i>PARTICLE_UPDATE</i>	44
Figura 16 – Sinais de entrada e saída gerados.	47
Figura 17 – Saída de referência e saída estimada - Ponto flutuante no <i>Matlab</i>	49
Figura 18 – Saída de referência e saída estimada - Ponto fixo no <i>Matlab</i>	50
Figura 19 – Formas de onda da saída estimada $\hat{y}[n]$ e valor da função custo (ao longo do processamento das 200 amostras), obtidas via simulação do bloco <i>COST_FUNCTION</i> no <i>ModelSim</i>	52
Figura 20 – Saída de referência e saída estimada - Implementação VHDL.	53

LISTA DE TABELAS

Tabela 1 – Representação do número 152 na base 10.	21
Tabela 2 – Representação do número 152 na base 2.	21
Tabela 3 – Sinais do bloco <i>COST_FUNCTION</i>	39
Tabela 4 – Sinais do bloco <i>PARTICLE_UPDATE</i>	41
Tabela 5 – Sinais do componente <i>multiplier</i>	43
Tabela 6 – Operações de cada estado da máquina de estados do bloco <i>PARTICLE_UPDATE</i>	44
Tabela 7 – Parâmetros do algoritmo PSO.	47
Tabela 8 – Resultados da estimação dos parâmetros utilizando o algoritmo PSO implementado em ponto flutuante no <i>Matlab</i>	48
Tabela 9 – Resultados da estimação dos parâmetros - Ponto fixo no <i>Matlab</i>	48
Tabela 10 – Resultados da validação do componente <i>multiplier</i>	51
Tabela 11 – Comparação entre os resultados ¹ da simulação em VHDL e da execução de referência em ponto fixo no <i>Matlab</i> para o bloco <i>PARTICLE_UPDATE</i> , iterações 50 a 53.	54
Tabela 12 – Resultados do bloco <i>PARTICLE_UPDATE</i> - VHDL.	54

LISTA DE ABREVIATURAS E SIGLAS

CC	Corrente Contínua
EA	Erro Absoluto
FPGA	<i>Field-Programmable Gate Array</i>
FT	Função de Transferência
LFSR	<i>Linear Feedback Shift Register</i>
MIF	<i>Memory Initialization File</i>
PRBS	<i>Pseudo-Random Binary Sequence</i>
PSO	<i>Particle Swarm Optimization</i>
RLC	Resistor-Indutor-Capacitor
TXT	<i>Plain Text File</i>
UFSC	Universidade Federal de Santa Catarina
VHDL	<i>VHSIC Hardware Description Language</i>
VHSIC	<i>Very High Speed Integrated Circuit</i>
ZOH	<i>Zero-Order Hold</i>

LISTA DE SÍMBOLOS

$\hat{y}[n]$	Saída estimada pelo modelo na amostra n
$\mathbf{v}_p(t)$	Vetor velocidade da partícula p na iteração t
t	Índice temporal ou número da iteração atual
w	Fator de inércia do algoritmo
c_1	Coefficiente de aceleração cognitiva
c_2	Coefficiente de aceleração social
r_1	Número aleatório uniformemente distribuído para a componente cognitiva
r_2	Número aleatório uniformemente distribuído para a componente social
$\mathbf{x}_p(t)$	Vetor posição da partícula p na iteração t
$\mathbf{p}_{best_p}$	Melhor posição individual encontrada pela partícula p
$\mathbf{g}_{best}$	Melhor posição global encontrada pelo enxame
R	Resistência de carga
L	Indutância
C	Capacitância
$u(t)$	Sinal de entrada do circuito RLC
$y(t)$	Sinal de saída do circuito RLC
$x[n]$	Entrada do sistema na amostra n
$y[n]$	Saída do sistema na amostra n
θ	Vetor de parâmetros estimados $\{\hat{b}_1, \hat{b}_2, \hat{a}_1, \hat{a}_2\}$
$\hat{b}_1$	Coefficiente estimado do numerador da FT discreta, posição 1
$\hat{b}_2$	Coefficiente estimado do numerador da FT discreta, posição 2
$\hat{a}_1$	Coefficiente estimado do denominador da FT discreta, posição 1
$\hat{a}_2$	Coefficiente estimado do denominador da FT discreta, posição 2
P	Número de partículas do enxame
$Runs$	Número de iterações do algoritmo
K	Dimensão do espaço de busca, número de parâmetros a estimar
$J(\theta)$	Função custo, somatório do erro quadrático
$Q_{m.n}$	Formato de ponto fixo com m bits inteiros e n bits fracionários

SUMÁRIO

1	INTRODUÇÃO	15
1.1	OBJETIVOS	16
1.1.1	Objetivo Geral	16
1.1.2	Objetivos Específicos	16
1.2	ORGANIZAÇÃO DO TRABALHO	16
2	FUNDAMENTAÇÃO TEÓRICA	18
2.1	OTIMIZAÇÃO POR ENXAME DE PARTÍCULAS	18
2.2	REPRESENTAÇÃO DE NÚMEROS BINÁRIOS	20
2.3	REPRESENTAÇÃO DE INTEIROS COM SINAL	21
2.4	REPRESENTAÇÃO COM PONTO FIXO	22
2.5	ARITMÉTICA DE PONTO FIXO	22
2.5.1	Adição e Subtração	22
2.5.2	Multiplicação	23
2.6	IDENTIFICAÇÃO DE SISTEMAS	24
2.6.1	Obtenção de Modelos Matemáticos para Sistemas	24
2.6.1.1	<i>Análise físico-matemática</i>	<i>24</i>
2.6.1.2	<i>Análise experimental</i>	<i>25</i>
2.6.2	Metodos Paramétricos e Não Paramétricos	27
3	METODOLOGIA	28
3.1	CONTEXTUALIZAÇÃO DA APLICAÇÃO	28
3.2	IMPLEMENTAÇÃO DO ALGORITMO PSO COM ARITMÉTICA DE PONTO FLUTUANTE	29
3.2.1	Inicialização do algoritmo	29
3.2.2	Avaliação da função custo	30
3.2.3	Atualização das partículas	31
3.2.4	Implementação da função custo	31
3.3	IMPLEMENTAÇÃO DO ALGORITMO PSO COM ARITMÉTICA DE PONTO FIXO	33
3.3.1	Operações aritméticas em ponto fixo	34
3.3.1.1	<i>Adição e Subtração</i>	<i>34</i>
3.3.1.2	<i>Multiplicação</i>	<i>35</i>
3.3.2	Implementação da função custo em ponto fixo	35
3.3.3	Atualização da velocidade e da posição	37
3.4	IMPLEMENTAÇÃO DOS BLOCOS DE FUNÇÃO CUSTO E <i>PARTI- CLE_UPDATE</i> EM VHDL	38
3.4.1	Bloco da função custo	38
3.4.2	Bloco de atualização da velocidade e posição das partículas .	40

3.4.2.1	<i>Bloco multiplicador</i>	42
3.4.2.2	<i>Atualização da velocidade e posição da partícula</i>	43
4	RESULTADOS	46
4.1	CENÁRIO DE TESTES	46
4.2	IMPLEMENTAÇÃO EM PONTO FLUTUANTE	47
4.3	IMPLEMENTAÇÃO EM PONTO FIXO	48
4.4	IMPLEMENTAÇÃO EM VHDL	50
4.4.1	Validação do componente <i>multiplier</i>	50
4.4.2	Validação do bloco <i>COST_FUNCTION</i>	51
4.4.3	Validação do bloco <i>PARTICLE_UPDATE</i>	53
5	CONCLUSÃO	55
	REFERÊNCIAS	57

1 INTRODUÇÃO

O crescente avanço dos sistemas de controle embarcado e de eletrônica de potência tem imposto exigências cada vez maiores em termos de precisão na modelagem e, ainda, na eficiência computacional no processamento. As aplicações de circuitos eletrônicos de potência abrangem equipamentos de conversão de alta potência, como linhas de transmissão de potência CC, aplicações de circuitos mais convencionais como fontes de alimentação para computadores, ferramentas elétricas portáteis, carregadores de bateria de celulares e bateria de automóveis híbridos (HART, 2012). A identificação de sistemas, que é uma técnica fundamental nesse contexto e com ampla área de aplicação, consiste na construção de modelos matemáticos de sistemas dinâmicos, para atender um determinado propósito, a partir de dados observados (LJUNG, 1999). Ao interagir com um sistema, faz-se necessário compreender como as suas variáveis interagem entre si, ou seja, construir um modelo que represente adequadamente o seu comportamento. No contexto de identificação de sistemas, a intenção é determinar os parâmetros que regem essas relações matemáticas, muitas vezes podendo ser abordada como um problema de otimização cujo objetivo primário é estimar parâmetros de um modelo candidato que melhor represente o comportamento do sistema real, estabelecendo uma base necessária para o projeto de estratégias de controle eficazes (COELHO; SANTOS COELHO, 2004).

Entre os diferentes métodos disponíveis para estimação de parâmetros, o algoritmo de Otimização por Enxame de Partículas se destaca como uma alternativa adotada em diversas aplicações, como robótica, energia e a otimização de sistemas complexos. Por ser um algoritmo estocástico baseado em populações e inspirado no comportamento coletivo de enxames na natureza, o PSO elimina a necessidade de propriedades matemáticas rígidas na função objetivo, como a diferenciabilidade ou separabilidade (HOUSSEIN *et al.*, 2021). Contudo, apesar de amplamente utilizado, o PSO envolve operações iterativas sobre populações de partículas, o que pode resultar em um elevado custo computacional quando implementado em sua forma convencional, tornando-se incompatível com plataformas embarcadas que exigem baixo consumo de recursos e operação em tempo real (DAMAJ *et al.*, 2020).

Nesse contexto, os dispositivos de lógica reconfigurável, especialmente os *Field-Programmable Gate Arrays* (FPGAs), são alternativas eficientes para execução de algoritmos de otimização (DA COSTA *et al.*, 2019). De maneira distinta das soluções fundamentadas exclusivamente em *software*, os FPGAs combinam flexibilidade de reprogramação com ampla capacidade de paralelização, acelerando algoritmos complexos. No entanto, a implementação do PSO em FPGAs estabelece um desafio adicional: esses dispositivos, especialmente os de menor tamanho, frequentemente não apresentam unidades de ponto flutuante dedicadas, tornando a aritmética de ponto fixo a abordagem mais adequada para esse contexto.

A utilização de aritmética de ponto fixo, portanto, permite superar o obstáculo imposto pela ausência de unidades de ponto flutuante em *hardware* embarcado, alinhando a capacidade de convergência do PSO com a necessidade de eficiência em área e velocidade de processamento, características essenciais para sistemas embarcados de baixo custo.

1.1 OBJETIVOS

Em sequência são descritos o objetivo geral e os objetivos específicos deste Trabalho de Conclusão de Curso.

1.1.1 Objetivo Geral

Diante do exposto, este trabalho tem como objetivo geral implementar o algoritmo PSO com aritmética de ponto fixo no formato $Q12.20$, aplicado à identificação de parâmetros de um conversor CC-CC do tipo *boost*, e desenvolver os principais blocos computacionais do algoritmo em linguagem de descrição de *hardware* VHDL, visando viabilizar sua futura síntese e execução em dispositivos FPGA.

1.1.2 Objetivos Específicos

- Implementar e validar o algoritmo PSO com aritmética de ponto flutuante no *Matlab*;
- Implementar e validar o algoritmo PSO com aritmética de ponto fixo no formato $Q12.20$ no *Matlab*;
- Definir e implementar em VHDL o bloco de cálculo da função custo;
- Definir e implementar em VHDL o bloco de atualização de velocidade e posição das partículas;
- Simular e validar o comportamento dos blocos implementados em VHDL no *ModelSim*;
- Comparar os resultados obtidos nas implementações em ponto fixo no *Matlab* e em VHDL com a referência em ponto flutuante.

1.2 ORGANIZAÇÃO DO TRABALHO

Este trabalho está estruturado em cinco capítulos. O Capítulo 1 apresenta a introdução, contextualizando a importância da identificação de sistemas e da implementação do algoritmo PSO em *hardware* reconfigurável, além de expor os objetivos geral e específicos que regem o desenvolvimento deste trabalho. O Capítulo 2 aborda a fundamentação teórica, descrevendo os principais conceitos relacionados ao algoritmo PSO, à representação e à aritmética de números em ponto fixo, e aos princípios de identificação de sistemas

necessários à compreensão do problema de estimação de parâmetros abordado. O Capítulo 3 apresenta a metodologia empregada no desenvolvimento do trabalho, iniciando pela contextualização da aplicação. Em seguida, é descrita a implementação do algoritmo PSO com aritmética de ponto flutuante no *Matlab*, utilizada como referência de comparação, seguida pela implementação em aritmética de ponto fixo no formato *Q12.20*, detalhando as principais operações aritméticas empregadas. Então, são apresentados os blocos de cálculo da função custo e de atualização de velocidade e posição das partículas, desenvolvidas em VHDL. O Capítulo 4 apresenta e discute os resultados obtidos a partir das implementações desenvolvidas, incluindo o cenário de testes utilizado, a comparação entre os parâmetros estimados e os parâmetros de referência da planta nas implementações em ponto flutuante e ponto fixo, e a validação funcional dos blocos em VHDL por meio de simulação no *ModelSim*. Por fim, o Capítulo 5 apresenta a conclusão deste trabalho, sintetizando as contribuições alcançadas e as perspectivas para trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta os principais conceitos necessários para o entendimento e desenvolvimento deste trabalho. Inicialmente, é apresentado o algoritmo de Otimização por Enxame de Partículas, descrevendo seu princípio de funcionamento e as equações que regem a atualização da posição e da velocidade das partículas. Em sequência, são abordados os conceitos relacionados à representação de números binários, à representação de inteiros com sinal por meio de complemento de 2 e à representação e aritmética de números em ponto fixo, fundamentais para compreensão da implementação do algoritmo em *hardware*. Por fim, são apresentados os princípios de identificação de sistemas, com destaque para as abordagens de obtenção de modelos matemáticos e a classificação dos métodos paramétricos e não paramétricos, relacionando esses conceitos ao problema de estimação de parâmetros tratado neste trabalho.

2.1 OTIMIZAÇÃO POR ENXAME DE PARTÍCULAS

A Otimização por Enxame de Partículas (PSO - *Particle Swarm Optimization*) é um algoritmo proposto por Kennedy e Eberhart em 1995, inspirado pelo comportamento social de animais, como bandos de pássaros e cardumes de peixes. Nesse método, soluções candidatas se movimentam pelo espaço de busca, sendo influenciadas tanto por sua melhor experiência individual quanto pela melhor experiência global do enxame, com o objetivo de minimizar a função custo (KENNEDY; EBERHART, 1995).

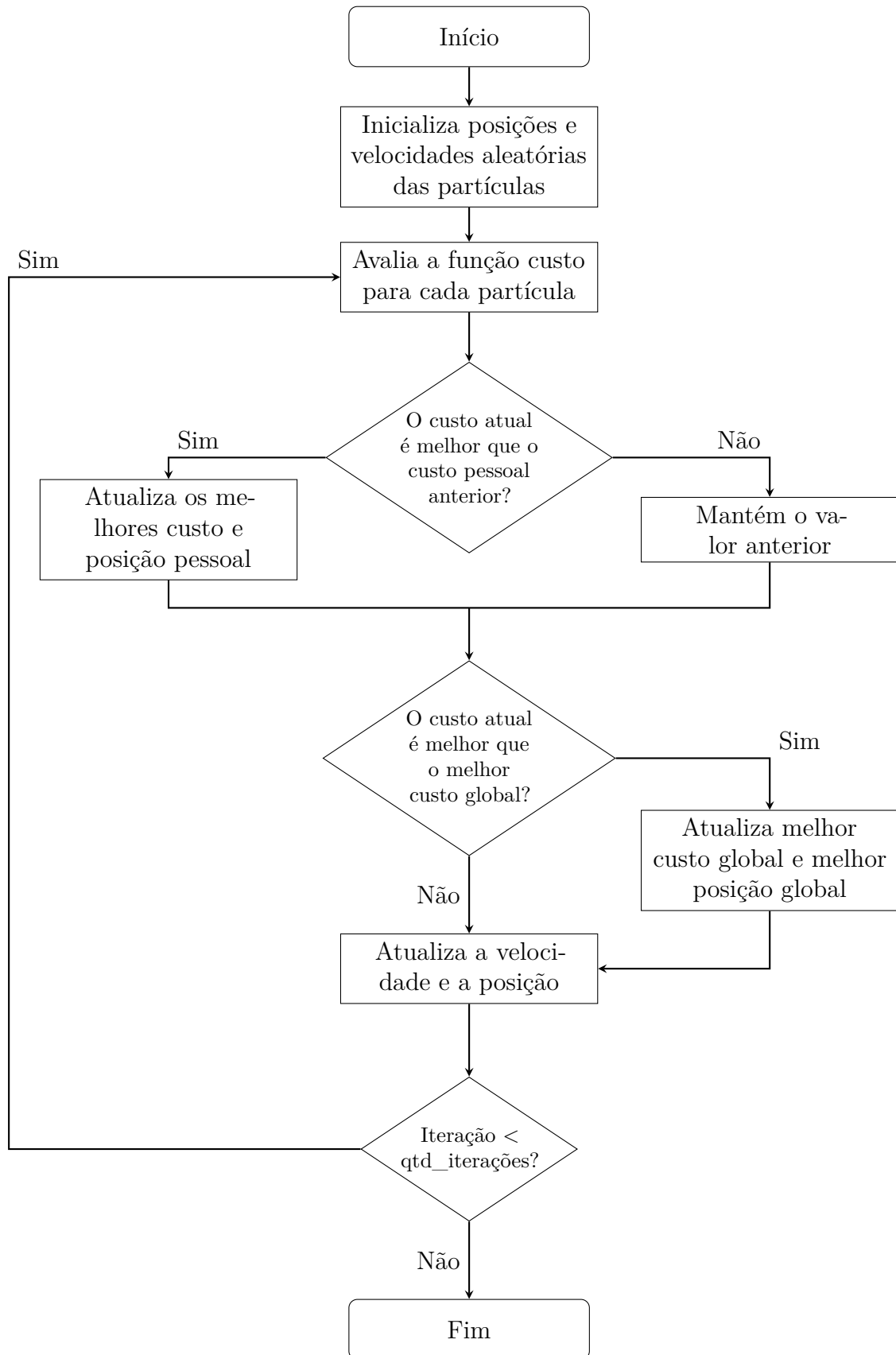
No PSO, cada partícula representa uma solução candidata do problema. O conjunto de partículas é o enxame, que se move iterativamente em um espaço de busca multidimensional, em que cada dimensão corresponde a uma variável do problema. A posição de uma partícula é uma possível solução, enquanto a velocidade define a direção e a magnitude do deslocamento no espaço de busca. O fluxograma do algoritmo é apresentado na Figura 1.

Durante o processo de otimização, as partículas atualizam as suas posições e velocidades com base em informações adquiridas nas iterações anteriores, à procura de regiões que apresentem melhores valores da função objetivo. Essas informações incluem a melhor posição individual encontrada até o momento ($\mathbf{p}_{best}$ - melhor pessoal) e a melhor posição global encontrada pelo enxame ($\mathbf{g}_{best}$ - melhor global). A atualização da velocidade é dada pela seguinte equação:

$$\mathbf{v}_p(t+1) = w \cdot \mathbf{v}_p(t) + c_1 \cdot r_1 \cdot (\mathbf{p}_{best_p} - \mathbf{x}_p(t)) + c_2 \cdot r_2 \cdot (\mathbf{g}_{best} - \mathbf{x}_p(t)) \quad (1)$$

em que, $\mathbf{v}_p(t)$ é a velocidade da partícula p na iteração t , enquanto w corresponde ao fator de inércia, responsável por regular a influência da velocidade anterior no movimento da partícula. Os parâmetros c_1 e c_2 são os coeficientes de aceleração associados, respectivamente, às componentes cognitiva e social do algoritmo, enquanto que r_1 e r_2 são números aleatórios uniformemente distribuídos no intervalo $[0, 1]$. A posição atual da partícula p

Figura 1 – Fluxograma do algoritmo PSO.



Fonte: Elaborado pela autora (2026), baseado em (DAMAJ *et al.*, 2020).

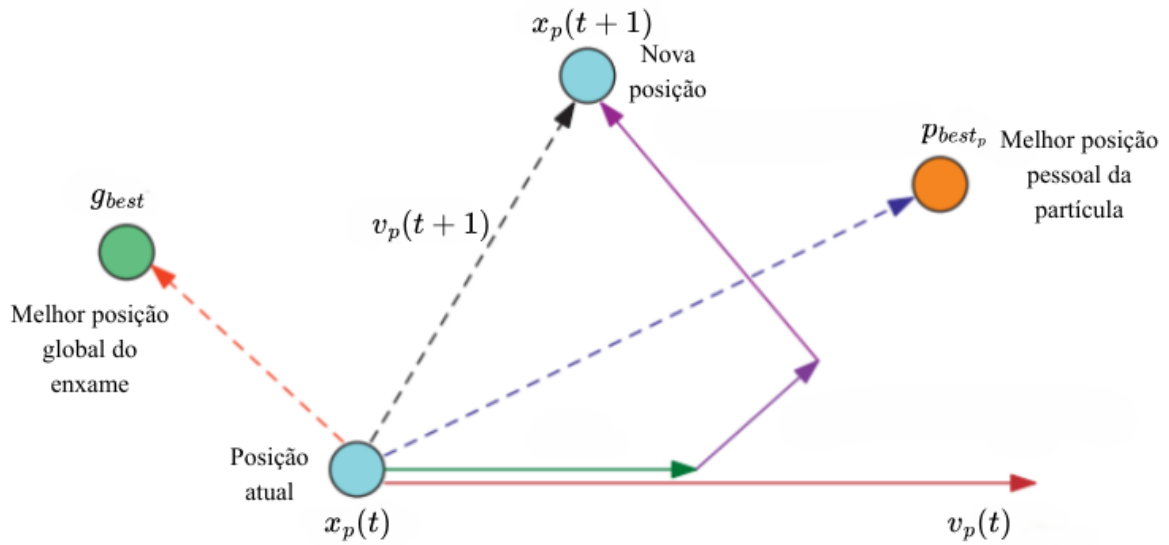
na iteração t é representada por $\mathbf{x}_p(t)$, enquanto $\mathbf{p}_{best_p}$ corresponde à melhor posição individual já encontrada por essa partícula ao longo do processo de busca. Por sua vez, $\mathbf{g}_{best}$ representa a melhor posição global obtida pelo enxame, considerando todas as partículas e iterações anteriores.

Em seguida à atualização da velocidade, a nova posição da partícula p é calculada pela Equação (2).

$$\mathbf{x}_p(t+1) = \mathbf{x}_p(t) + \mathbf{v}_p(t+1) \quad (2)$$

em que $\mathbf{x}_p(t+1)$ representa a nova posição da partícula, obtida pela soma da posição atual $\mathbf{x}_p(t)$ com a velocidade atualizada $\mathbf{v}_p(t+1)$. Dessa forma, a partícula se desloca no espaço de busca em direção a soluções que apresentem menor valor da função custo. Esse processo é repetido iterativamente até que um critério de parada seja satisfeito, como um número máximo de iterações. A Figura 2 ilustra o mecanismo de atualização da velocidade e sua influência no deslocamento das partículas ao longo da busca pela melhor solução.

Figura 2 – Mecanismo de atualização da velocidade do PSO.



Fonte: Adaptado de (REDDIPOGU; ELUMALAI, 2020).

2.2 REPRESENTAÇÃO DE NÚMEROS BINÁRIOS

Os números decimais utilizam o sistema de numeração de base dez, que é um sistema de numeração em que o número mais à direita referem-se a quantidade de unidades (10^0), à sua esquerda tem-se a quantidade das dezenas (10^1), em seguida as centenas (10^2) e assim por diante, como é apresentado na Tabela 1, onde a representação do número 152_{10} , pode ser representado como $0 \cdot 10^3 + 1 \cdot 10^2 + 5 \cdot 10^1 + 2 \cdot 10^0$ na base 10.

No entanto, em um sistema computadorizado baseado em eletrônica digital, os números são representados em uma base distinta, conhecida como base binária ou base

Tabela 1 – Representação do número 152 na base 10.

Decimal	0	1	5	2
Base 10	10^3	10^2	10^1	10^0

Fonte: Elaborado pela autora (2026).

dois, composta pelos dígitos 0 e 1. Ao trabalhar em base 2, cada dígito 1 adiciona uma determinada potência de 2 na construção do valor final, ou seja, um número binário é uma sequência de bits, em que cada bit será multiplicado pela potência de 2 correspondente. Por exemplo, o número 152_{10} (na base 10) pode ser representado na base 2 como 10011000_2 , conforme mostrado na Tabela 2, onde a quantidade representada pode ser obtida multiplicando-se cada dígito 0 ou 1 pelo seu correspondente peso ou valor, ou seja, $1 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0$.

Tabela 2 – Representação do número 152 na base 2.

Bits	1	0	0	1	1	0	0	0
Potência	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Valor	128	0	0	16	8	0	0	0

Fonte: Elaborado pela autora (2026).

2.3 REPRESENTAÇÃO DE INTEIROS COM SINAL

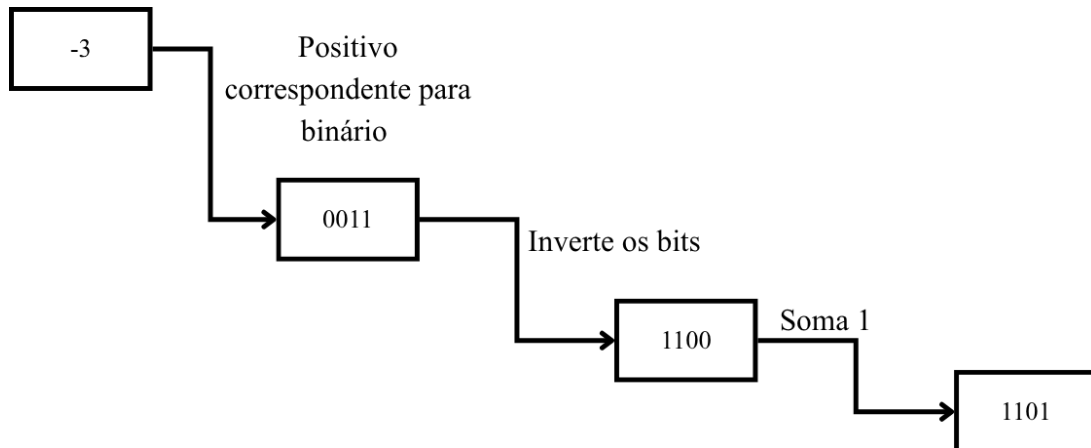
Uma das maneiras mais utilizadas para a representação de números inteiros com sinal (isto é, positivos ou negativos) na base 2 é o formato conhecido como Complemento de 2. Nesse formato, para diferenciar números positivos e negativos é utilizado o bit mais significativo para representar um peso negativo com valor -2^{N-1} , onde N corresponde ao número de bits utilizado na representação.

A representação de números positivos em complemento de 2 permanece igual a representação binária padrão, com o bit mais significativo igual à zero. Por exemplo, o valor 127_{10} , pode ser representado com 8 bits em complemento de 2 da seguinte forma: 0111111_2 , ou seja, $0 \cdot (-2^7) + 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 127_{10}$.

Na representação de números inteiros negativos em complemento de 2, o bit mais significativo (que possui o único peso negativo) é sempre 1. Por exemplo, o valor -1_{10} , pode ser representado com 8 bits em complemento de 2 da seguinte forma: 1111111_2 , ou seja, $1 \cdot (-2^7) + 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = -1_{10}$.

Uma forma prática de obter a representação binária em complemento de 2 para um número negativo é primeiramente, obter a representação binária do número positivo correspondente, em seguida inverter todos os bits do seu número e, por fim, adicionar uma unidade ao resultado, como é apresentado na Figura 3. Para o formato de complemento de 2, considerando N bits, é possível representar valores no intervalo -2^{N-1} até $2^{N-1} - 1$.

Figura 3 – Representação de números negativos em Complemento de 2.



Fonte: Elaborado pela autora (2026).

2.4 REPRESENTAÇÃO COM PONTO FIXO

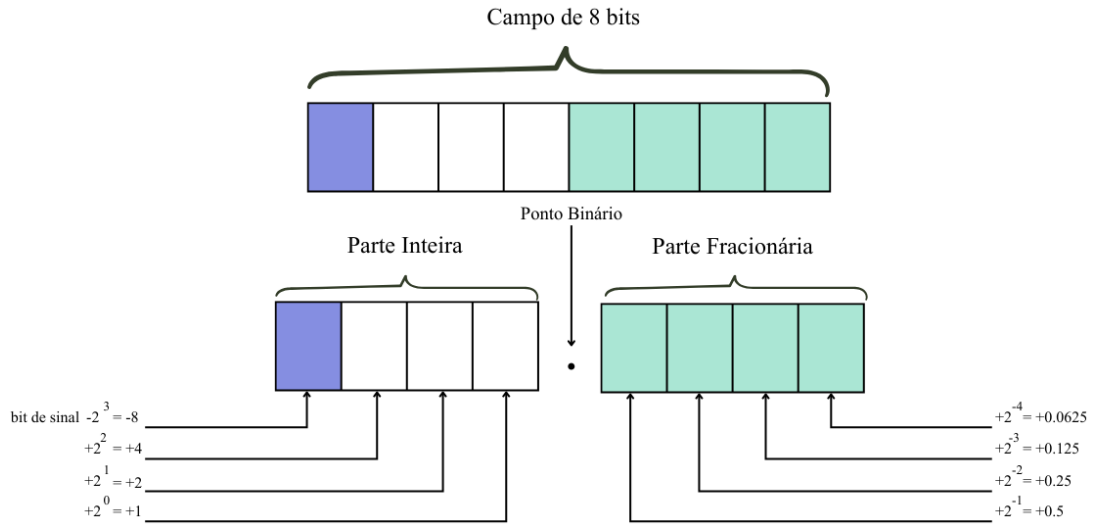
Uma das possíveis formas de representar números binários com parte fracionária é a representação em ponto fixo. Um número em ponto fixo reserva uma quantidade fixa de bits para representar a parte inteira e outra quantidade para a parte fracionária. Ou seja, um número binário em ponto fixo pode ser interpretado como se houvesse um ponto, de posição fixa (definida pela projetista), separando os bits das partes inteiras e fracionárias. A Figura 4 ilustra a representação de um número de 8 bits em ponto fixo no formato Q4.4, o qual é composto por 1 bit de sinal, 3 bits para a parte inteira e 4 bits para a parte fracionária. Conforme mostrado na Figura 4, cada bit da parte inteira representa uma quantidade dada pelo fator 2^b , onde b refere-se a posição do bit, iniciando em 0 (para o caso de complemento de 2, a última posição terá peso -2^b). Cada bit da parte fracionária também representa a quantidade dada pelo fator 2^b , porém agora com b decrescente e iniciando em -1 . Por exemplo, no formato Q4.4, o menor número positivo que pode ser representado é $0000.0001_2 = 2^{-4}_{10} = 0,0625_{10}$. Assim, utilizando Q4.4, o menor valor negativo que pode ser representado é $1000.0000_2 = -2^3_{10} = -8_{10}$ e o maior valor positivo é $0111.1111_2 = 7,9375_{10}$, com resolução de $0000.0001_2 = 0,0625_{10}$ entre valores consecutivos.

2.5 ARITMÉTICA DE PONTO FIXO

2.5.1 Adição e Subtração

Para realizar a soma de números de ponto fixo é necessário o alinhamento entre os pontos binários dos operandos. A adição é feita da mesma maneira da adição binária usual e, na subtração, emprega-se com complemento de 2, ou seja, o número subtraído é representado como negativo e a operação entre eles passa ser a adição (PYEATT; UGHETTA, 2020).

Figura 4 – Representação de número em ponto fixo.



Fonte: Adaptado de (SILVA, 2026).

$$\begin{array}{r}
 0010.1100 \quad (2.75) \\
 +0010.1000 \quad (2.5) \\
 \hline
 0101.0100 \quad (5.25)
 \end{array} \tag{3}$$

$$\begin{array}{r}
 0010.1100 \quad (2.75) \\
 +1101.1000 \quad (-2.5) \\
 \hline
 0000.0100 \quad (0.25)
 \end{array} \tag{4}$$

2.5.2 Multiplicação

Na multiplicação de um número de ponto fixo $QN.F$ com outro número $QN.F$ resulta em um número $Q(N+N).(F+F)$, isto é, com $N+N$ bits na parte inteira e $F+F$ bits na parte fracionária. Ou seja, na multiplicação de números de ponto fixo, o tamanho da parcela fracionária resultante é igual a soma da quantidade de bits fracionários dos operandos. Em (5) é possível observar um exemplo de multiplicação entre números binários de formato Q4.4.

$$\begin{array}{r}
0010.1100 \quad (2.75) \\
\times 0010.1000 \quad (2.5) \\
\hline
00000000 \\
0000000+ \\
0000000 + + \\
00101100 + + + \\
0000000 + + + + \\
00101100 + + + + + \\
0000000 + + + + + + \\
0000000 + + + + + + + \\
\hline
0000110.11100000 \quad (6.875)
\end{array} \tag{5}$$

Para que o resultado da multiplicação seja representado com a mesma quantidade de bits dos operandos, realiza-se um ajuste por meio de deslocamento de bits. Para isso, deslocam-se F bits para a direita, sendo F a quantidade original de bits da parte fracionária. Em seguida, realiza-se o truncamento da parte inteira pela quantidade de N bits. Por exemplo:

$$\begin{array}{r}
0000110.11100000 \quad (6.875) \\
- - - - - > \text{deslocamento de 4 bits} \\
00000000110.1110 \\
\hline
0110.1110 \quad (6.857)
\end{array} \tag{6}$$

2.6 IDENTIFICAÇÃO DE SISTEMAS

2.6.1 Obtenção de Modelos Matemáticos para Sistemas

Usualmente os procedimentos envolvidos na obtenção de modelos matemáticos para representação de sistemas são classificados em análise físico-matemática e análise experimental (COELHO; SANTOS COELHO, 2004).

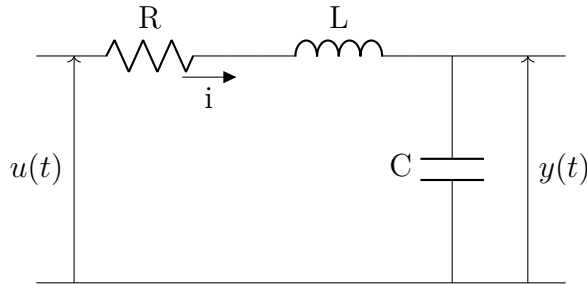
2.6.1.1 Análise físico-matemática

Essa abordagem, fundamenta-se nas leis da física que descrevem um determinado sistema, como os princípios de conservação de massa, energia e movimento. Um exemplo é a Figura 5 que apresenta a ilustração de um modelo físico de um circuito elétrico RLC.

Considerando a lei das tensões de *Kirchhoff*, é obtida a equação que descreve a dinâmica do sistema elétrico:

$$u(t) = Ri(t) + L \frac{di(t)}{dt} + \frac{1}{C} \int i(t) dt \tag{7}$$

Figura 5 – Circuito elétrico RLC.



Fonte: Adaptado de (COELHO; SANTOS COELHO, 2004).

com,

$$i(t) = C \frac{dy(t)}{dt} \quad (8)$$

em que R é a resistência, L representa a indutância, C caracteriza a capacitância, $u(t)$ denota a tensão de entrada e $y(t)$ é a tensão de saída. A representação matemática do sistema em termos da transformada de *Laplace* é dada por

$$\frac{Y(s)}{U(s)} = \frac{1}{LCs^2 + RCs + 1}. \quad (9)$$

2.6.1.2 Análise experimental

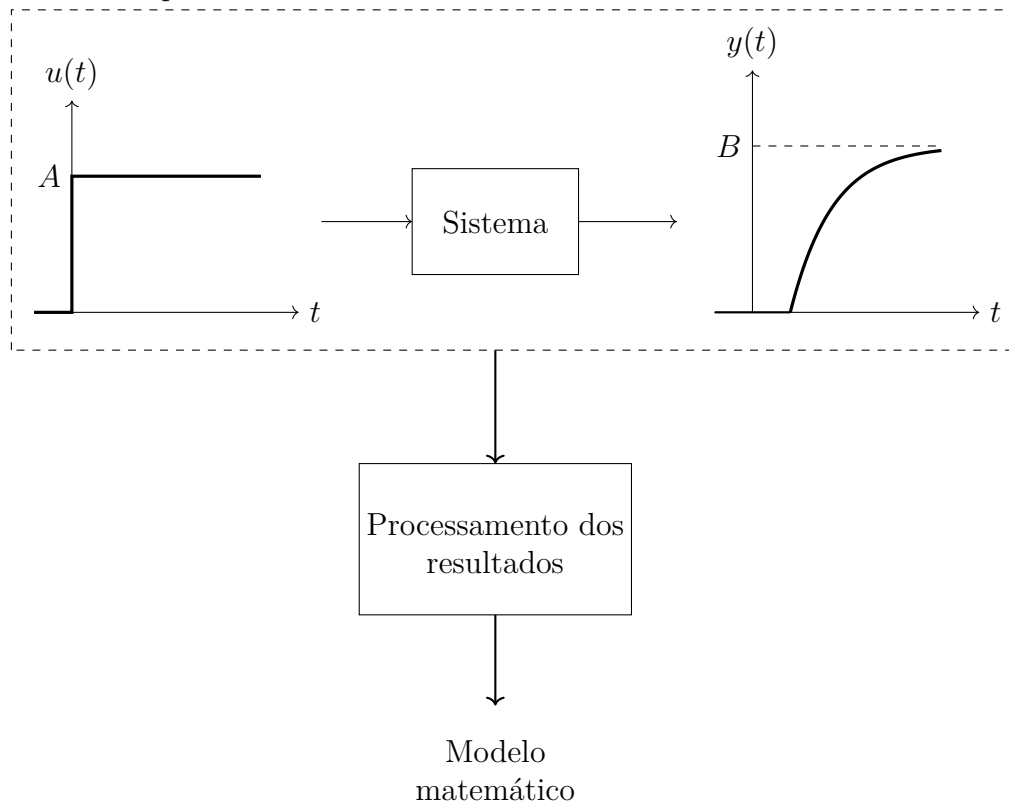
Outra forma de obtenção de modelos matemáticos é a partir de dados observados (por exemplo, entrada e saída), em conformidade com critérios predeterminados (JOHANSSON, 1993). Por vezes, tal processo é abordado como um problema de otimização, que utiliza algumas medidas para ajustar modelos candidatos à representação de um processo real. A escolha de modelos matemáticos e o ajuste de parâmetros são influenciados por diferentes elementos, como o conhecimento prévio do sistema e suas propriedades (por exemplo, sua complexidade ou ordem), a seleção da medida do erro a ser minimizado e a presença de ruídos (COELHO; SANTOS COELHO, 2004).

Nesse contexto, a identificação de sistemas por meio de dados experimentais, pode-se dividir em duas categorias:

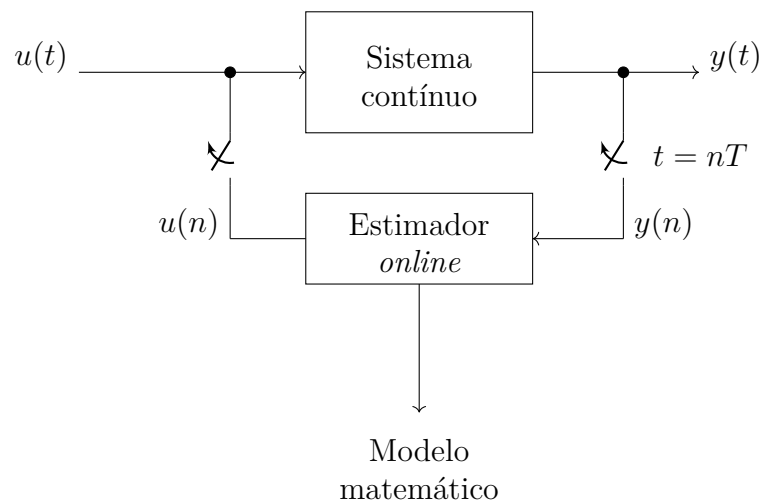
- Métodos Off-line: primeiramente, é realizado o ensaio experimental adquirindo medidas de entrada e saída do sistema e, após essa etapa, é obtido o modelo matemático para representar o sistema (Figura 6).
- Métodos On-line: os parâmetros do modelo matemático são estimados e atualizados conforme novas medições das entradas e saídas são obtidas, como mostrado na Figura 7.

Figura 6 – Ensaio experimental para obtenção do modelo matemático.

Ensaio experimental



Fonte: Adaptado do material de aula do Prof. Dr. Marcos Vinicius Matsuo.

Figura 7 – Estrutura do estimador *online*.

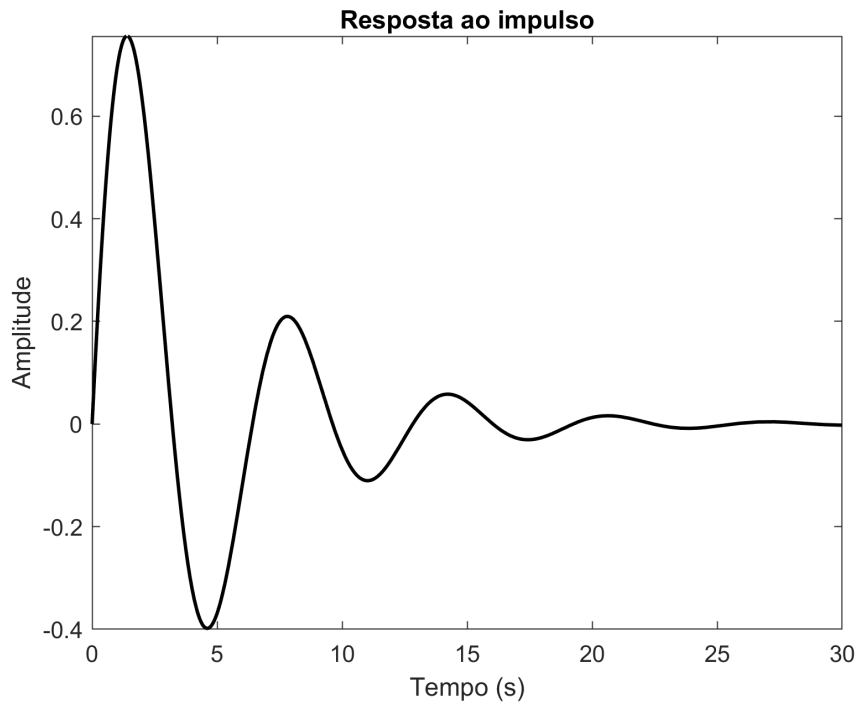
Fonte: Adaptado do material de aula do Prof. Dr. Marcos Vinicius Matsuo.

2.6.2 Métodos Paramétricos e Não Paramétricos

Os métodos de identificação de sistemas, baseados em análise experimental, podem ainda também ser paramétricos e não paramétricos. Nos métodos paramétricos, primeiramente, define-se um modelo paramétrico para representação de um sistemas (por exemplo, função de transferência de segunda ordem). Na sequência, os dados medidos são processados (de forma *offline* ou *online*) para estimação dos parâmetros do modelo previamente definido (AGUIRRE, 2007). Por outro lado, nos métodos não paramétricos, são estimados modelos não paramétricos para representação do sistema, por exemplo, resposta ao impulso ou resposta em frequência. A título de ilustração, o modelo apresentado na Equação (10) é paramétrico, sendo seus parâmetros 1, 0.4, 1. A resposta ao impulso $h(t) = \mathcal{L}^{-1}\{H(s)\}$ é apresentado na Figura 8, sendo um exemplo de modelo não paramétrico.

$$H(s) = \frac{1}{s^2 + 0.4s + 1} \quad (10)$$

Figura 8 – Representações não paramétricas. Resposta ao impulso de (10).



Fonte: Adaptado de (AGUIRRE, 2007).

Vale ressaltar que a informação contida na resposta ao impulso (mostrada na Figura 8) é a mesma que a função de transferência (Equação (10)), mesmo não contendo nenhum parâmetro.

3 METODOLOGIA

Neste capítulo é apresentado o desenvolvimento do trabalho proposto, cujo foco consiste na implementação do algoritmo PSO utilizando aritmética de ponto-fixe no *software Matlab* no contexto de identificação de sistemas. Inicialmente, é realizada a contextualização da aplicação. Em seguida, é apresentada a implementação do PSO utilizando aritmética de ponto flutuante no *Matlab*. Posteriormente, é descrita a implementação do algoritmo utilizando aritmética de ponto fixo, apresentando as principais operações aritméticas empregadas. Por fim, são apresentados os blocos da função custo e de atualização das velocidades e posições das partículas implementadas em VHDL.

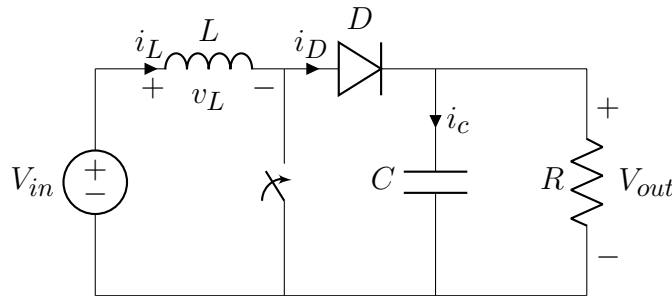
3.1 CONTEXTUALIZAÇÃO DA APLICAÇÃO

Neste trabalho, o algoritmo PSO é utilizado para estimar os coeficientes de uma função de transferência (FT) de segunda ordem dada por:

$$H(z) = \frac{b_1 z^{-1} + b_2 z^{-2}}{1 + a_1 z^{-1} + a_2 z^{-2}} = \frac{Y(z)}{X(z)} \quad (11)$$

Especificamente, a função de transferência acima é utilizada para modelar um conversor CC-CC do tipo *boost* (planta), cujo diagrama de circuito é apresentado na Figura 9. Para geração dos sinais utilizados no processo de identificação, considera-se o modelo de pequenos sinais do conversor. A função de transferência contínua considerada que representa a dinâmica do sistema é apresentada na Equação (12), a qual é discretizada utilizando o método ZOH (*Zero-Order Hold*) para obtenção do modelo discreto equivalente na forma da Equação (11).

Figura 9 – Diagrama de circuito de um conversor *boost*.



Fonte: Adaptado de (HART, 2012).

$$H(s) = \frac{Y(s)}{X(s)} = \frac{V_{in}}{(1-D)^2} \frac{Ls + R(1-D)^2}{RLCs^2 + Ls + R(1-D)^2} \quad (12)$$

Por meio de simulação numérica, são geradas amostras da entrada $x[n]$ e da saída $y[n]$ da planta, as quais são utilizadas no processo de identificação. Como função custo

(considerada no algoritmo PSO), é adotado o somatório do erro quadrático entre amostras $y[n]$ medidas da saída da planta (obtidas via simulação) e da saída estimada $\hat{y}[n]$, ou seja,

$$J(\theta) = \sum_{n=1}^{n=N} (y[n] - \hat{y}[n])^2 \quad (13)$$

onde $\theta = \{\hat{b}_1, \hat{b}_2, \hat{a}_1, \hat{a}_2\}$ é o conjunto de parâmetros estimados e a saída estimada $\hat{y}[n]$ é computada como

$$\hat{y}[n] = \hat{b}_1 x[n-1] + \hat{b}_2 x[n-2] - \hat{a}_1 \hat{y}[n-1] - \hat{a}_2 \hat{y}[n-2] \quad (14)$$

Para estimação do vetor θ que minimiza a função custo, é utilizado o PSO, onde cada partícula representa um vetor de parâmetros candidatos e se move pelo espaço de busca com base em sua própria experiência e na melhor solução encontrada pelo enxame até o aquele momento.

3.2 IMPLEMENTAÇÃO DO ALGORITMO PSO COM ARITMÉTICA DE PONTO FLUTUANTE

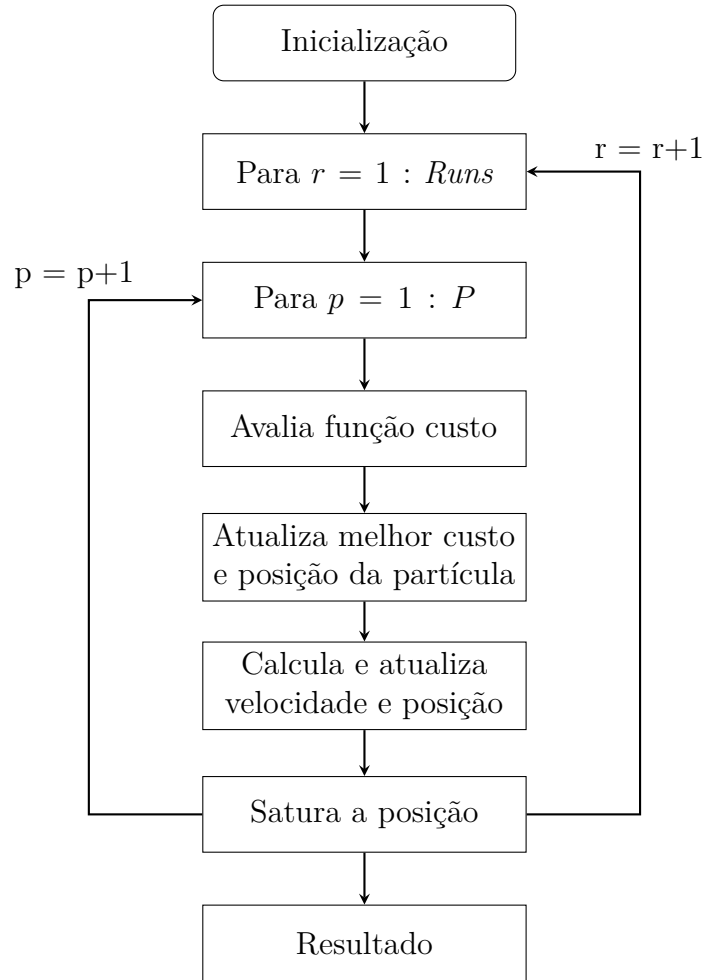
A implementação em ponto flutuante tem como objetivo estabelecer uma referência de desempenho para o algoritmo PSO aplicado à identificação da planta descrita na Seção 3.1, servindo como base de comparação para a implementação em ponto fixo apresentada nas seções seguintes.

A organização geral do algoritmo é ilustrada no fluxograma da Figura 10. A primeira etapa do algoritmo corresponde à inicialização das partículas e dos parâmetros do PSO. Em sequência, para cada iteração r e para cada partícula p da população P , é realizado o cálculo da função custo, responsável por avaliar a adequação da solução candidata ao sistema de interesse. Com base no custo calculado, são atualizadas as melhores posições individuais e globais, seguidas da etapa de atualização da velocidade e posição. Por fim, as posições são saturadas nos limites do espaço de busca, garantindo que os parâmetros estimados permaneçam dentro dos intervalos estabelecidos.

3.2.1 Inicialização do algoritmo

Na etapa de inicialização, são definidos os parâmetros do algoritmo: número de iterações $Runs$, tamanho do enxame P , quantidade de parâmetros a serem estimados K , que no contexto deste trabalho corresponde a quantidade de coeficientes que devem ser estimados (isto é, $\hat{b}_1$, $\hat{b}_2$, $\hat{a}_1$ e $\hat{a}_2$) da função de transferência definida na Equação (11). Adicionalmente, também são definidos os valores dos componentes de ponderação adotados, especificamente, w para a inércia, c_1 para o componente cognitivo e c_2 para o componente social. O espaço de busca é delimitado por vetores definidos com base em conhecimento prévio sobre os coeficientes da planta. As posições e velocidades iniciais de cada partícula são sorteadas aleatoriamente dentro desse intervalo.

Figura 10 – Fluxograma geral do algoritmo implementado.



Fonte: Elaborado pela autora (2026).

3.2.2 Avaliação da função custo

A função custo $J(\theta)$ é avaliada para cada partícula p por meio da função `f_custo`. Essa função simula o modelo discreto amostra a amostra, utilizando *buffers* de entrada e saída para armazenar valores anteriores dos sinais. A saída estimada $\hat{y}[n]$ é calculada iterativamente conforme a Equação (14), enquanto o erro quadrático médio entre a saída estimada e a saída medida é acumulado ao longo da simulação. A avaliação da função custo para cada partícula é realizada conforme o Código 3.1.

Código 3.1 – Avaliação da função custo.

```

1 current_cost = f_custo(current_position(:,p), x, y);

```

Caso o custo calculado seja inferior ao melhor custo individual registrado previamente, as variáveis associadas à melhor posição pessoal da partícula são atualizadas. Da mesma maneira, caso o custo obtido também seja inferior ao melhor custo global do enxame, a melhor posição global é atualizada como é apresentado no Código 3.2.

Código 3.2 – Avaliação do custo.

```

1 if current_cost < best_personal_cost(p)
2     best_personal_cost(p) = current_cost;
3     best_personal_position(:,p) = current_position(:,p);
4
5     if current_cost < best_global_cost
6         best_global_cost = current_cost;
7         best_global_position = current_position(:,p);
8     end
9 end

```

3.2.3 Atualização das partículas

Conforme apresentado na Seção 2.1, a atualização da velocidade das partículas é realizada considerando os componentes de inércia, cognitivo e social do algoritmo PSO:

Código 3.3 – Cálculo da velocidade.

```

1 current_velocity(:,p) = w*current_velocity(:,p) +
2     c1*r1.*(best_personal_position(:,p) - ...
3     current_position(:,p)) + ...
4     c2*r2.*(best_global_position - current_position(:,p));

```

Em sequência, a posição das partículas é atualizada pela soma entre a posição atual e a velocidade calculada:

Código 3.4 – Cálculo da posição.

```

1 current_position(:,p) = ...
2     current_position(:,p) + current_velocity(:,p);

```

Após a atualização, é executada a saturação dos parâmetros para que os coeficientes permaneçam dentro da região de busca previamente estabelecida. Ao impedir que as partículas extrapolem esses limites, essa restrição evita a exploração de regiões divergentes, acelerando a convergência da solução.

3.2.4 Implementação da função custo

A fim de avaliar o desempenho do conjunto de parâmetros ao longo do processo de otimização, é implementada a função custo, apresentada no Código 3.5. O primeiro procedimento realizado é a extração do vetor de parâmetros os coeficientes estimados $\hat{b}_1$, $\hat{b}_2$, $\hat{a}_1$ e $\hat{a}_2$, correspondentes a função de transferência candidata. Em sequência, são inicializados os vetores *xbuffer* e *ybuffer*, responsáveis por armazenar as amostras passadas dos sinais de entrada e saída necessárias para o cálculo recursivo da Equação (13).

A cada iteração, uma nova amostra de entrada é inserida no *buffer* de entrada (*xbuffer*), enquanto as amostras anteriores são deslocadas para as posições subsequentes. A saída estimada é então calculada utilizando os coeficientes da partícula avaliada no momento e os valores armazenados nos *buffers*. Essa operação é a implementação da equação das diferenças do sistema discreto.

Após o cálculo da saída estimada, o *buffer* de saída é atualizado para que as amostras mais recentes fiquem disponíveis na iteração seguinte. É, então, calculado o erro entre a saída medida (simulada) da planta e a saída estimada pelo modelo. O erro ao quadrado é acumulado na variável de custo J , conforme apresentado na Equação (13).

Dessa forma, partículas das quais os coeficientes geram saídas estimadas mais próximas da resposta real da planta apresentam menores valores de custo e possuem maior influência na direção de busca do enxame nas próximas iterações.

Código 3.5 – Implementação da função `f_custo`.

```

1 function [J, yestimado] = f_custo(parametros, xmedido, ymedido)
2
3     N = numel(ymedido);
4     J = 0;
5     yestimado = zeros(N,1);
6
7     b1 = parametros(1);
8     b2 = parametros(2);
9     a1 = parametros(3);
10    a2 = parametros(4);
11
12    xbuffer = zeros(3,1);
13    ybuffer = zeros(2,1);
14
15    for n = 1:N
16
17        xbuffer(2:end) = xbuffer(1:end-1);
18        xbuffer(1) = xmedido(n);
19
20        yestimado(n) = [0 b1 b2]*xbuffer - [a1 a2]*ybuffer;
21
22        ybuffer(2:end) = ybuffer(1:end-1);
23        ybuffer(1) = yestimado(n);
24
25        J = J + (yestimado(n)-ymedido(n))^2;
26    end
27 end

```

3.3 IMPLEMENTAÇÃO DO ALGORITMO PSO COM ARITMÉTICA DE PONTO FIXO

Nesta seção, é discutida a implementação do algoritmo PSO, apresentado originalmente na Seção 3.2, porém agora adotando aritmética de ponto fixo. Essa abordagem facilita a posterior adaptação do algoritmo para arquiteturas *hardware* que não possuem circuitos para realização de operações aritméticas com ponto flutuante, tais como alguns modelos de microcontroladores e FPGAs. Enquanto a aritmética de ponto flutuante ajusta dinamicamente a posição do ponto binário, a aritmética de ponto fixo utiliza uma representação inteira com escala fixa, na qual a posição do ponto binário é definida previamente pelo projetista e mantida constante ao longo de toda a execução.

Neste trabalho, adotou-se o formato de ponto fixo com sinal $Qm.n$, no qual m denota o número de bits da parte inteira, incluindo o bit de sinal, e n representa a quantidade de bits fracionários. Fez-se uso de números inteiros de 32 bits sob o formato $Q12.20$, sendo 20 bits destinados para parte fracionária e os 12 bits restantes para a parcela inteira, incluindo o bit de sinal. Empregando 12 bits à parte inteira, no formato de complemento de 2, o intervalo representável é $[-2048, 2047]$. Por sua vez, os 20 bits fracionários dispõem de uma resolução de $2^{-20} \approx 9.5367 \cdot 10^{-7}$. Dessa maneira, o menor número positivo representável é 2^{-20} , o limite inferior negativo é $-2048(100000000000.00000000000000000000_2)$ e o maior valor positivo é aproximadamente $2047.9999(011111111111.11111111111111111111_2)$. Essa estrutura oferece uma faixa dinâmica ampla e reduz o risco de saturação nos valores intermediários das operações de multiplicação e acumulação ao longo da execução do algoritmo.

Apesar dessa robustez, a falta de tratamento para *overflow* e saturação pode comprometer a convergência do PSO. O *overflow* em complemento de 2 distorce os valores devido ao seu comportamento cíclico (como a inversão indesejada de sinal), propagando erros que levam o sistema à divergência. A saturação mitiga esse problema ao limitar os resultados aos extremos da faixa representável, embora introduza uma leve perda de precisão ao truncar valores que extrapolam os limites, ela evita erros de grande magnitude. Para garantir a estabilidade numérica e a fidelidade em relação à referência em ponto flutuante, este trabalho adotou o formato $Q12.20$ e aplicou a operação de saturação imediatamente após cada cálculo aritmético.

Com o objetivo de viabilizar a simulação (no *Matlab*) do algoritmo com aritmética de ponto fixo, primeiramente é realizada a conversão de todos os dados e parâmetros de ponto flutuante para ponto fixo (no formato $Q12.20$). Isso é feito pela multiplicação do valor original pelo fator de escala 2^{20} e o truncamento para inteiro de 32 bits com sinal, conforme a Equação (15).

$$x_{\text{int}} = \text{int32}(x \cdot 2^{20}) \quad (15)$$

x representa o dado original em ponto flutuante, $\text{int32}(\cdot)$ denota a conversão para o tipo de dado 32 bits com sinal e x_{int} consiste no valor inteiro de 32 bits resultante. Dessa

maneira, tanto as amostras da entrada $x[n]$ e saída $y[n]$ da planta, quanto os parâmetros do algoritmo (w , c_1 , c_2) e os limiares do espaço de busca são representados como inteiros de 32 bits antes do início do processo de otimização. A partir deste momento todas as operações são realizadas em aritmética de ponto fixo.

A operação de mapeamento inversa, utilizada para converter de ponto fixo para ponto flutuante, é calculada pela divisão do valor inteiro pelo fator de escala 2^{20} , obtendo o valor de ponto flutuante correspondente. Essa conversão é utilizada apenas para verificação e apresentação dos resultados, por exemplo, para comparar os coeficientes estimados obtidos pelo algoritmo com os valores esperados em ponto flutuante.

3.3.1 Operações aritméticas em ponto fixo

Para que os resultados das operações aritméticas não ultrapassem os limites representáveis no formato de 32 bits com sinal, após cada operação, os resultados obtidos são limitados às extremidades representáveis, evitando o *overflow*. O Algoritmo 1 apresenta o procedimento de saturação, que é utilizado por todas as operações aritméticas implementadas.

Algoritmo 1 QSAT - Saturação para 32 bits

```

1: % x : int64
2: % y : valor saturado em int32
3: function Q_SAT( $x$ )
4:    $MAX32 \leftarrow \text{int64}(\text{intmax}('int32'))$            % Limite superior do tipo int32
5:    $MIN32 \leftarrow \text{int64}(\text{intmin}('int32'))$          % Limite inferior do tipo int32
6:   if  $x$  é maior que  $MAX32$  then
7:      $x \leftarrow MAX32$ 
8:   end if
9:   if  $x$  é menor que  $MIN32$  then
10:     $x \leftarrow MIN32$ 
11:  end if
12:   $y \leftarrow \text{int32}(x)$ 
13:  return  $y$ 
14: end function

```

3.3.1.1 Adição e Subtração

A adição e a subtração entre dois operandos de 32 bits são executadas após a conversão dos operandos para 64 bits, preservando o resultado da ocorrência de *overflow* e o bit de sinal. O resultado da operação é então saturado, gerando um valor de 32 bits, conforme descrito nos Algoritmos 2 e 3.

Algoritmo 2 Adição em ponto fixo

```

1: % a, b: inteiros de 32 bits em formato  $Qm.n$ 
2: % y: inteiro de 32 bits em formato  $Qm.n$ 
3: function QADD( $a, b$ )
4:    $temp \leftarrow \text{int64}(a) + \text{int64}(b)$ 
5:    $y \leftarrow \text{Q\_SAT}(temp)$ 
6:   return  $y$ 
7: end function

```

Algoritmo 3 Subtração em ponto fixo

```

1: % a, b: inteiros de 32 bits em formato  $Qm.n$ 
2: % y: inteiro de 32 bits em formato  $Qm.n$ 
3: function QSUB( $a, b$ )
4:    $temp \leftarrow \text{int64}(a) - \text{int64}(b)$ 
5:    $y \leftarrow \text{Q\_SAT}(temp)$ 
6:   return  $y$ 
7: end function

```

3.3.1.2 Multiplicação

Conforme discutido na Seção 2.5.2, a multiplicação em ponto fixo produz um resultado com o dobro de bits fracionários, sendo necessários procedimentos adicionais em comparação à adição e à subtração. Para retomar a forma original, é necessário deslocar o produto intermediário n posições à direita, ou seja, realizar um deslocamento aritmético à direita. Nesse procedimento, os bits são deslocados para a direita e o bit mais significativo, o bit de sinal em complemento de dois, é replicado nas posições liberadas à esquerda, preservando assim o sinal do número. No entanto, antes de realizar o deslocamento aritmético, o produto intermediário é arredondado por meio da adição de 2^{n-1} , uma vez que a operação de deslocamento apenas trunca o resultado. Caso esse arredondamento não seja realizado, pode ocorrer perda de informações associadas à parte fracionária descartada. O Algoritmo 4 apresenta o procedimento de multiplicação em ponto fixo.

O modo de operação da multiplicação é selecionado por um parâmetro (*flag*) que difere o produto entre um escalar e um vetor do produto elemento a elemento entre dois vetores. O primeiro (*flag* = 0), refere-se às multiplicações pelos coeficientes w , c_1 e c_2 , já o segundo (*flag* = 1) é necessário para o cálculo dos componentes cognitivo e social da velocidade, nos quais tanto o coeficiente aleatório quanto a diferença de posições são vetores de dimensão K .

3.3.2 Implementação da função custo em ponto fixo

A função custo em ponto fixo possui o mesmo comportamento da função custo descrita na Seção 3.2, substituindo as operações em ponto flutuante por suas equivalentes em ponto fixo. A saída estimada definida na Equação (14) é calculada amostra a amostra

Algoritmo 4 Multiplicação em ponto fixo

```

1: % a, b: inteiros de 32 bits em formato  $Qm.n$  (escalares ou vetores de dimensão  $K$ )
2: % y: inteiro (ou vetor) de 32 bits em formato  $Qm.n$ 
3: function QMUL( $a, b, n, flag$ )
4:   if  $flag = 1$  then
5:      $temp \leftarrow \text{int64}(a) \odot \text{int64}(b)$       % produto elemento a elemento, em 64 bits
6:   else
7:      $temp \leftarrow \text{int64}(a) \times \text{int64}(b)$       % produto escalar-vetor, em 64 bits
8:   end if
9:    $temp \leftarrow temp + 2^{n-1}$                   % arredondamento
10:   $temp \leftarrow temp \gg n$                   % deslocamento aritmético à direita de  $n$  posições
11:   $y \leftarrow \text{Q\_SAT}(temp)$                   % saturação para 32 bits
12:  return  $y$ 
13: end function

```

por meio de *buffers* de entrada e saída de 32 bits. Assim como na implementação anterior, a cada amostra n , os *buffers* são atualizados por deslocamento e a saída estimada é calculada por meio das funções apresentadas anteriormente. O loop principal da implementação é apresentado no Código 3.6.

Código 3.6 – Loop principal da função f_custo_fp .

```

1 for  $n = 1:N$ 
2   % atualiza buffer da entrada
3    $xbuffer(3) = xbuffer(2);$ 
4    $xbuffer(2) = xbuffer(1);$ 
5    $xbuffer(1) = xin(n);$ 
6
7   % calcula a saída do sistema
8    $aux1 = \text{QMUL}(b1, xbuffer(2), \text{FRACTION\_BITS}, 0);$ 
9    $aux2 = \text{QMUL}(b2, xbuffer(3), \text{FRACTION\_BITS}, 0);$ 
10   $aux3 = \text{QMUL}(a1, ybuffer(1), \text{FRACTION\_BITS}, 0);$ 
11   $aux4 = \text{QMUL}(a2, ybuffer(2), \text{FRACTION\_BITS}, 0);$ 
12   $yestimado(n) = \text{QSUB}(\text{QSUB}(\text{QADD}(aux1, aux2), aux3), aux4);$ 
13
14  % atualiza buffer da saída
15   $ybuffer(2) = ybuffer(1);$ 
16   $ybuffer(1) = yestimado(n);$ 
17
18  % atualiza custo
19   $aux5 = \text{QMUL}(\text{QSUB}(yestimado(n), yin(n)), \text{QSUB}(yestimado(n), yin(n)), \text{FRACTION\_BITS}, 0);$ 
20   $J = \text{QADD}(J, aux5);$ 
21 end

```

3.3.3 Atualização da velocidade e da posição

A atualização da velocidade e da posição das partículas segue as mesmas equações da implementação anterior, alterando apenas o modo de cálculo, que passa a utilizar operações em ponto fixo. Para facilitar futura implementação em *hardware* e garantir a correta utilização das funções de operações, o cálculo do vetor de velocidades para cada partícula p é decomposto em variáveis auxiliares, essa separação divide a equação de cálculo da velocidade em três componentes principais ($aux1$, $aux2$ e $aux3$). Além disso, os coeficientes aleatórios r_1 e r_2 são gerados como inteiros no intervalo $[0; 2^{20}]$, representando, no formato $Qm.n$ ($Q12.20$), valores uniformes no intervalo $[0; 1]$, como é apresentado no Código 3.7.

Código 3.7 – Cálculo dos componentes aleatórios.

```
1 r1 = int32(randi(2^FRACTION_BITS, K, 1));
2 r2 = int32(randi(2^FRACTION_BITS, K, 1));
```

O primeiro termo auxiliar calculado corresponde ao termo de inércia ($aux1$), responsável pela influência da direção anterior no movimento da partícula. Obtido pelo produto escalar entre o fator de inércia w e o vetor de velocidade atual ($current_velocity$), cuja dimensão é K , a multiplicação em ponto fixo é realizada com $flag = 0$, conforme Código 3.8.

Código 3.8 – Cálculo da variável auxiliar $aux1$.

```
1 aux1 = QMUL(w, current_velocity(:,p), FRACTION_BITS, 0);
```

O componente auxiliar $aux2$ corresponde à componente cognitiva, que representa a tendência da partícula retornar à melhor posição que ela já encontrou no processo de busca. O cálculo é realizado em duas etapas, primeiro, o coeficiente aleatório $r1$ é multiplicado elemento a elemento ($flag = 1$) pela diferença entre a melhor posição pessoal ($best_personal_position$) e a posição atual ($current_position$), armazenando no termo intermediário $aux20$. Então, o termo intermediário é multiplicado pelo termo cognitivo $c1$, obtendo assim o segundo termo da Equação (1), conforme o Código 3.9.

Código 3.9 – Cálculo da variável auxiliar $aux2$.

```
1 aux20 = QMUL(r1, QSUB(best_personal_position(:,p), ...
2                     current_position(:,p)), FRACTION_BITS, 1);
3 aux2 = QMUL(c1, aux20, FRACTION_BITS, 0);
```

O terceiro termo auxiliar ($aux3$) representa o componente social, que é a influência da melhor posição global ($best_global_position$) obtida pelo enxame sobre o movimento da partícula. O cálculo segue a mesma estrutura da componente cognitiva, o coeficiente aleatório $r2$ é multiplicado elemento a elemento pela diferença entre a melhor posição global, com $flag = 1$, sendo armazenado no termo intermediário $aux30$. Em sequência,

é obtido o terceiro termo da Equação (1), multiplicando *aux30* pelo coeficiente social *c2* com *flag* = 0, conforme 3.10.

Código 3.10 – Cálculo da variável auxiliar *aux3*.

```

1 aux30 = QMUL(r2, QSUB(best_global_position, ...
2     current_position(:,p)), FRACTION_BITS,1);
3 aux3 = QMUL(c2, aux30, FRACTION_BITS,0);

```

Com os três termos calculados, a velocidade da partícula é atualizada pela soma dos termos como é apresentado no Código 3.11.

Código 3.11 – Atualização da velocidade.

```

1 current_velocity(:,p) = QADD(aux1, QADD(aux2, aux3));

```

Conforme a Equação (2), a posição de cada partícula é atualizada pela soma da posição atual com a nova velocidade, conforme o Código 3.12. Por fim, a posição é saturada nos limiares do espaço de busca, garantindo que os parâmetros estimados permaneçam dentro dos valores previamente definidos.

Código 3.12 – Atualização da posição.

```

1 current_position(:,p) = QADD(current_position(:,p), ...
2     current_velocity(:,p));

```

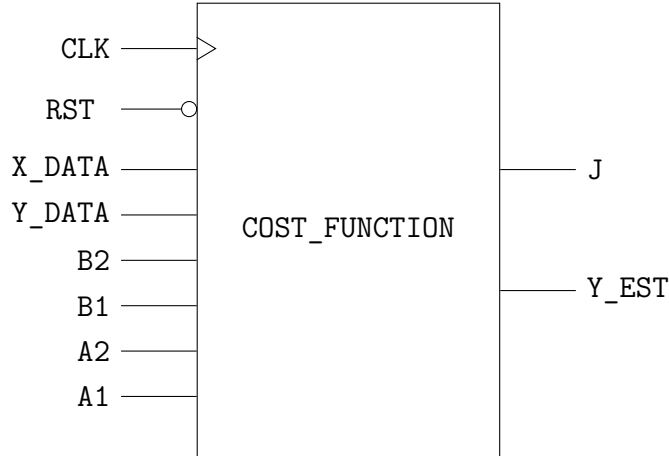
3.4 IMPLEMENTAÇÃO DOS BLOCOS DE FUNÇÃO CUSTO E *PARTICLE_UPDATE* EM VHDL

Após implementar o algoritmo PSO em ponto fixo no *Matlab*, os principais blocos do algoritmo são agora implementados em VHDL. Esse desenvolvimento, permite que tais blocos sejam sintetizados em FPGA, visando paralelizar e acelerar a execução do algoritmo. Vale ressaltar que este trabalho não tem por objetivo implementar todo o algoritmo PSO em VHDL, mas desenvolver os blocos responsáveis pelo cálculo da função custo (*COST_FUNCTION*) e pela atualização da velocidade e posição das partículas (*PARTICLE_UPDATE*), responsáveis pelo maior custo computacional do algoritmo. O desenvolvimento desses blocos é realizado de forma que a versão em VHDL reproduza exatamente o comportamento da implementação em ponto fixo, discutida nas seções anterior.

3.4.1 Bloco da função custo

O bloco de cálculo do custo *COST_FUNCTION* recebe, a cada ciclo de *clock*, uma amostra de entrada $x[n]$ e uma amostra de saída referência $y[n]$, além dos quatro coeficientes estimados ($B2$, $B1$, $A2$ e $A1$) relacionados à posição atual da partícula. Na Tabela 3 é apresentada a listagem e descrição dos sinais utilizados e na Figura 11 é exibido o bloco com os sinais de entrada e saída.

Figura 11 – Bloco da função custo.



Fonte: Elaborado pela autora (2026).

Tabela 3 – Sinais do bloco *COST_FUNCTION*.

Sinal	Tipo	Descrição
CLK	Entrada	Sinal de clock
RESET	Entrada	Reset assíncrono ativo em nível baixo
X_DATA	Entrada	Amostra de entrada $x[n]$ em $Q12.20$
Y_DATA	Entrada	Amostra de saída $y[n]$ de referência em $Q12.20$
B2	Entrada	Coefficiente $\hat{b}_2$ em $Q12.20$
B1	Entrada	Coefficiente $\hat{b}_1$ em $Q12.20$
A2	Entrada	Coefficiente $\hat{a}_2$ em $Q12.20$
A1	Entrada	Coefficiente $\hat{a}_1$ em $Q12.20$
J	Saída	Custo acumulado J em $Q12.20$
Y_EST	Saída	Saída estimada $\hat{y}[n]$ em $Q12.20$

Fonte: Elaborado pela Autora (2026).

O comportamento do bloco implementado em linguagem de *hardware* é caracterizado por um processo sensível ao *clock* e ao *reset*. Quando o *reset* é ativado em nível baixo ($RESET = '0'$), os *buffers* e registradores internos são zerados. A cada borda de subida do *clock*, o bloco executa as operações descritas no Algoritmo 5.

O bloco efetua quatro multiplicações por ciclo de *clock*, relacionadas com o cálculo da saída estimada $\hat{y}[n]$, conforme (14), mais uma multiplicação para o cálculo do erro quadrático. Ao ocorrer a borda de subida do *clock*, o *buffer* de entrada é atualizado por deslocamento, de modo que a amostra recebida em X_DATA é armazenada na posição $x[n]$, enquanto os valores anteriores são deslocados para as posições $x[n - 1]$ e $x[n - 2]$. Esse deslocamento em cadeia garante que, a cada ciclo, as amostras necessárias para o cálculo da Equação (14) estejam disponíveis nos registradores intermediários.

Em sequência, a saída estimada $\hat{y}[n]$ é calculada, com a utilização de termos auxiliares. Como essas multiplicações são executadas sobre operandos de 32 bits, para evitar *overflow*, cada produto é internamente calculado em 64 bits e reescalado para 32 bits

Algoritmo 5 Comportamento do bloco *COST_FUNCTION*

```

1: if RESET = 0 then
2:   Zerar todos os buffers, registradores e saídas
3: else if borda de subida do clock then
4:   Atualiza buffers de entrada:
5:      $x[n-2] \leftarrow x[n-1]$ 
6:      $x[n-1] \leftarrow x[n]$ 
7:      $x[n] \leftarrow X\_DATA$ 
8:   Calcula saída estimada:
9:      $aux1 \leftarrow ((\hat{b}_1 \times x[n-1]) \gg 20)$ 
10:     $aux2 \leftarrow (\hat{b}_2 \times x[n-2]) \gg 20$ 
11:     $aux3 \leftarrow (\hat{a}_1 \times \hat{y}[n-1]) \gg 20$ 
12:     $aux4 \leftarrow (\hat{a}_2 \times \hat{y}[n-2]) \gg 20$ 
13:     $\hat{y}[n] \leftarrow \text{resize}(aux1 + aux2 - aux3 - aux4, 32)$ 
14:   Atualiza buffer de saída:
15:      $\hat{y}[n-2] \leftarrow \hat{y}[n-1]$ 
16:      $\hat{y}[n-1] \leftarrow \hat{y}[n]$ 
17:   Atualiza o custo:
18:      $\text{diff} \leftarrow \hat{y}[n] - Y\_DATA$ 
19:      $J_{acumulado} \leftarrow J_{acumulado} + \text{resize}((\text{diff} \times \text{diff}) \gg 20, 32)$ 
20:   Atualiza saídas:
21:      $Y\_EST \leftarrow \hat{y}[n]$ 
22:      $J \leftarrow J_{acumulado}$ 
23: end if

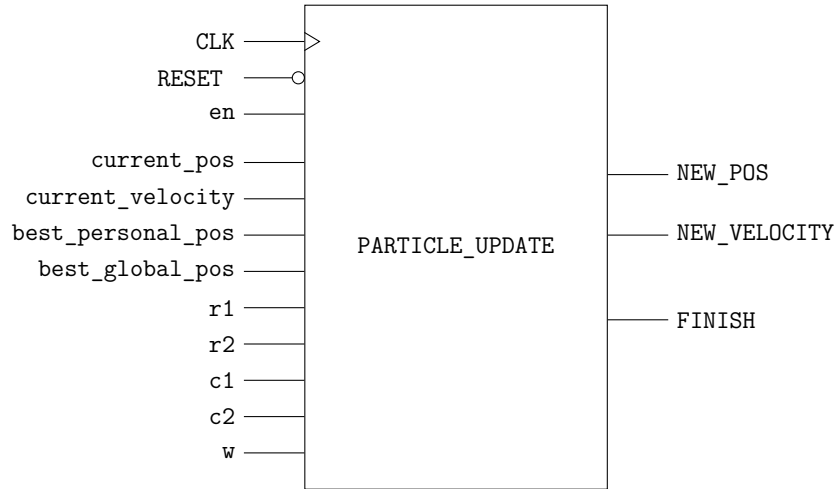
```

pelo deslocamento aritmético à direita de n posições, que na implementação considerada neste trabalho corresponde a 20 bits, conforme descrito na Seção 2.5. Os resultados são somados e subtraídos, obtendo $\hat{y}[n]$, que é truncado em 32 bits pela função **resize**. Após o cálculo da saída estimada, o *buffer* de saída é atualizado de forma análoga ao de entrada, e o erro ($\text{diff} = \hat{y}[n] - Y_DATA$) é calculado, elevado ao quadrado, reescalado e acumulado ao custo $J_{acumulado}$.

3.4.2 Bloco de atualização da velocidade e posição das partículas

O bloco *PARTICLE_UPDATE* implementa, em *hardware*, as equações de atualização de velocidade e posição descritas nas Equações (1) e (2). Ao contrário do bloco *COST_FUNCTION*, no qual as multiplicações são realizadas sobre escalares a cada ciclo de *clock*, o cálculo da equação de velocidade contém operações sobre vetores de dimensão $K = 4$, o que gera restrições importantes à implementação em *hardware*. A Figura 12 apresenta o bloco com os sinais de entrada e saída e na Tabela 4 são descritos os sinais adotados.

Figura 12 – Bloco de atualização da velocidade e posição das partículas.



Fonte: Elaborado pela autora (2026).

Tabela 4 – Sinais do bloco *PARTICLE_UPDATE*.

Sinal	Tipo	Descrição
CLK	Entrada	Sinal de <i>clock</i>
RESET	Entrada	Reset assíncrono ativo em nível baixo
en	Entrada	Habilita o início do processamento da partícula
current_pos	Entrada	Posição atual da partícula em $Q_{12.20}$
current_velocity	Entrada	Velocidade atual da partícula em $Q_{12.20}$
best_personal_pos	Entrada	Melhor posição pessoal em $Q_{12.20}$
best_global_pos	Entrada	Melhor posição global em $Q_{12.20}$
r1, r2	Entrada	Coefficientes aleatórios em $Q_{12.20}$
c1, c2	Entrada	Coefficientes cognitivo e social em $Q_{12.20}$
w	Entrada	Fator de inércia em $Q_{12.20}$
new_pos	Saída	Nova posição da partícula em $Q_{12.20}$
new_velocity	Saída	Nova velocidade da partícula em $Q_{12.20}$
finish	Saída	Indica a conclusão do processamento da partícula

Fonte: Elaborado pela autora (2026).

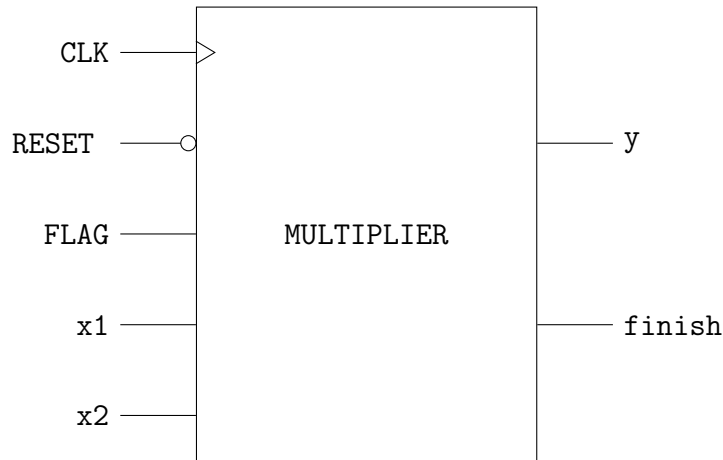
Para executar as multiplicações sobre vetores, foi implementado um componente auxiliar denominado *multiplier*, responsável por calcular os produtos do bloco *PARTICLE_UPDATE*. Os blocos foram desenvolvidos possuindo como *hardware* alvo a placa disponível na UFSC (Universidade Federal de Santa Catarina), a *Altera DE2-115 Board*, que contém o dispositivo *FPGA Altera Cyclone IV 4CE115*, o qual dispõe de 266 multiplicadores embarcados (18×18 bits) (ALTERA CORPORATION, 2010). Apesar dessa quantidade comportar multiplicadores de 32 bits instanciados em paralelo, optou-se para o bloco *PARTICLE_UPDATE* pela implementação de um único componente *multiplier* reutilizado de forma sequencial. Para justificar essa escolha, é possível calcular o número, teórico, máximo de partículas que poderiam ser processadas em paralelo caso o componente *multiplier* não fosse utilizado sequencialmente. Considerando apenas o bloco de atualização da partícula, caso todas as multiplicações desse bloco fossem implementadas

de maneira totalmente paralela, cada uma das cinco operações vetoriais demandaria $K = 4$ multiplicadores de 32 bits, totalizando $5 \times K = 20$ multiplicadores por partícula. Como cada multiplicador consome quatro blocos de 18 bits, para cada partícula seriam utilizados 80 blocos multiplicadores. Dividindo os 266 blocos disponíveis no dispositivo alvo por essa necessidade ($\frac{266}{80} \approx 3$), apenas três partículas poderiam ser processadas simultaneamente em paralelo, sem considerar os recursos demandados pelo restante do algoritmo. Logo, a escolha de implementação de um *multiplier*, reduz o consumo de blocos multiplicadores para apenas quatro blocos de 18 bits.

3.4.2.1 Bloco multiplicador

O componente *multiplier*, cujos sinais são apresentados na Figura 13 e descritos na Tabela 5, recebe dois vetores de entrada, **x1** e **x2**, cada um com K elementos de **DATA_WIDTH** bits concatenados, e um sinal **FLAG**, que define o modo de operação. A Figura 14 ilustra o mapeamento e o fatiamento desses vetores conforme o valor da **FLAG**. Quando **FLAG** = '0', o primeiro operando (**aux1**) assume o valor dos últimos 32 bits de **x1**, ou seja, no elemento da posição zero, independentemente do valor do contador interno, caracterizando uma multiplicação entre escalar e vetor. Quando **FLAG** = '1', o operando **aux1** é indexado pelo contador **cout**, de modo que a cada ciclo de *clock* é selecionado uma fatia distinta, de 32 bits, de **x1**, ou seja, uma multiplicação elemento a elemento entre dois vetores. Nos dois modos de operação, o operando **aux2** é indexado pelo contador, visto que **x2** corresponde ao vetor na operação escalar-vetor e ao segundo vetor na operação elemento a elemento. Essa diferenciação é análoga ao parâmetro **flag** utilizado na função *QMUL* na implementação em *Matlab*.

Figura 13 – Bloco multiplicador.



Fonte: Elaborado pela autora (2026).

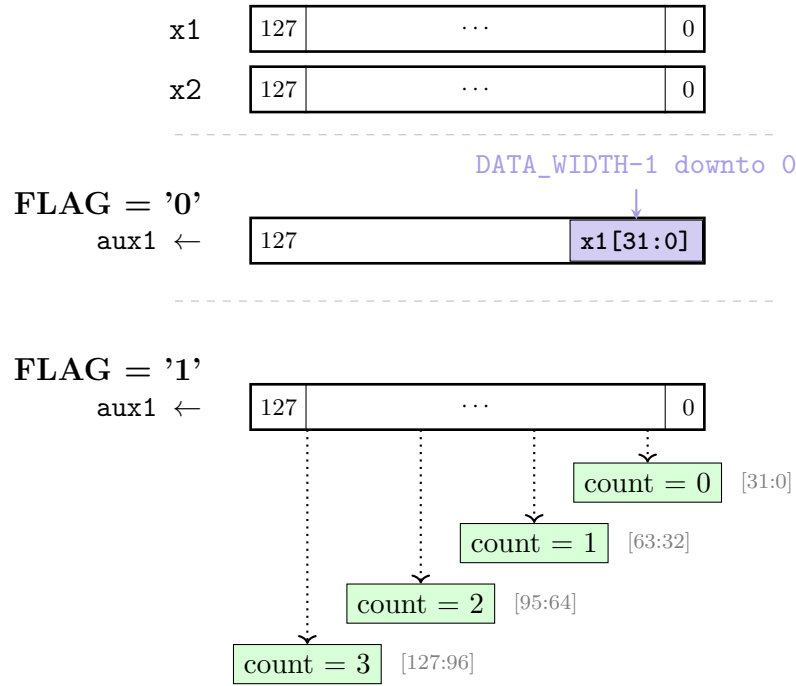
A cada ciclo de *clock*, o bloco calcula o produto do elemento selecionado, reescala o resultado pelo deslocamento aritmético à direita de **FRACTION_BITS** posições e incrementa

Tabela 5 – Sinais do componente *multiplier*.

Sinal	Tipo	Descrição
CLK	Entrada	Sinal de clock
RESET	Entrada	Reset assíncrono ativo em nível baixo
FLAG	Entrada	Seleciona o modo de operação: '0' para produto escalar-vetor, '1' para produto elemento a elemento
x1	Entrada	Vetor de K elementos em $Q_{12.20}$, concatenados em um único barramento
x2	Entrada	Vetor de K elementos em $Q_{12.20}$, concatenados em um único barramento
y	Saída	Vetor resultante do produto em $Q_{12.20}$, concatenado em um único barramento
finish	Saída	Indica a conclusão do processamento dos K elementos

Fonte: Elaborado pela autora (2026).

Figura 14 – Mapeamento e fatiamento dos vetores de entrada conforme o sinal FLAG.



Fonte: Elaborado pela autora (2026).

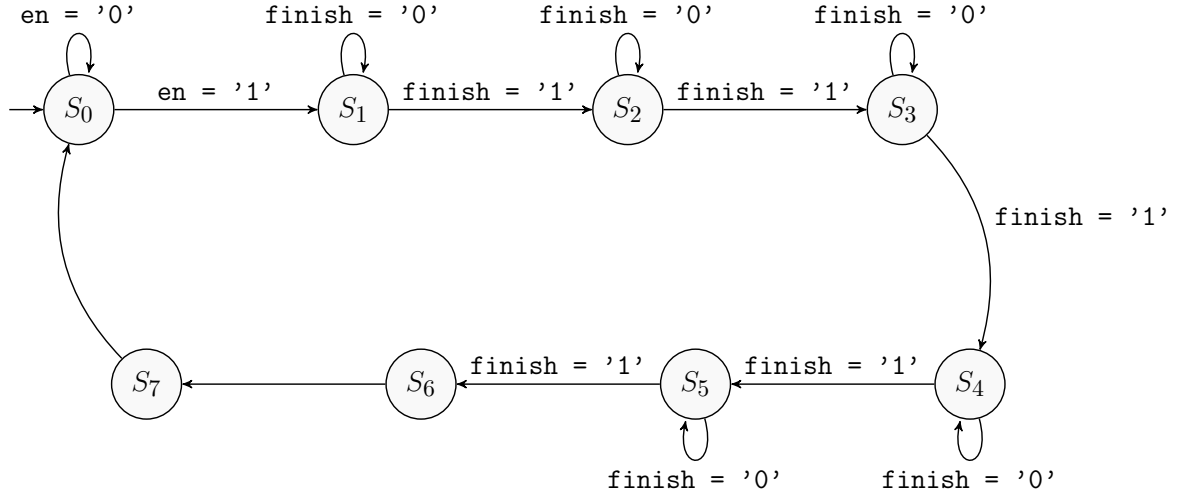
o contador interno. Ao atingir o K -ésimo elemento, o sinal **finish** é ativado, indicando que o vetor resultante está disponível na saída **y**, e o contador é reiniciado.

3.4.2.2 Atualização da velocidade e posição da partícula

O comportamento do bloco *PARTICLE_UPDATE* é controlado por uma máquina de estados finitos composta por oito estados, apresentada na Figura 15. No estado S_0 , o bloco permanece em espera enquanto o sinal de *enable* (**en**) estiver em nível baixo, ao ser ativado, o estado muda, avançando para S_1 . Nos estados S_1 a S_5 , o bloco configura as entradas do componente *multiplier* e permanece no estado enquanto o sinal de término

(*finish*) estiver em nível baixo, avançando ao estado seguinte somente quando o produto correspondente for finalizado. A Tabela 6 apresenta a operação realizada em cada um desses estados.

Figura 15 – Máquina de Estados do bloco *PARTICLE_UPDATE*.



Fonte: Elaborado pela autora (2026).

Tabela 6 – Operações de cada estado da máquina de estados do bloco *PARTICLE_UPDATE*.

Estado	Operação
S_1	Calcula a componente de inércia $aux1 = w \times \text{currenty_velocity}$, com FLAG = '0'.
S_2	Calcula a diferença (best_personal_position – currenty_position) e o produto elemento a elemento $aux20 = \mathbf{r1} \odot (\text{best_personal_position} - \text{currenty_position})$, com FLAG = '1'.
S_3	Escala o resultado intermediário pelo coeficiente cognitivo: $aux2 = c1 \times aux20$, com FLAG = '0'.
S_4	Calcula a diferença (best_global_position – current_position) e o produto elemento a elemento $aux30 = \mathbf{r2} \odot (\text{best_global_position} - \text{current_position})$, com FLAG = '1'.
S_5	Escala o resultado intermediário pelo coeficiente social: $aux3 = c2 \times aux30$, com FLAG = '0'.
S_6	Soma as três componentes para obter a nova velocidade: current_velocity = $aux1 + aux2 + aux3$.
S_7	Atualiza a posição pela soma com a nova velocidade: current_position = current_position + current_velocity ; ativa o sinal finish e retorna ao estado S_0 .

Fonte: Elaborado pela autora (2026).

De maneira distinta dos estados anteriores, as transições de S_6 para S_7 e de S_7 para S_0 acontecem de forma incondicional, dado que a soma das componentes de velocidade e a atualização da posição são operações realizadas diretamente sobre os sinais já calculados, sem a necessidade do componente *multiplier*, logo completadas em um único ciclo de

clock. Nesta implementação, a saturação nos limiares do espaço de busca não é aplicada dentro do bloco `PARTICLE_UPDATE`.

4 RESULTADOS

Este capítulo apresenta e discute os resultados obtidos a partir dos testes realizados para o algoritmo PSO implementado neste trabalho no contexto de identificação de sistemas. Inicialmente, é descrito o cenário de simulação considerado, incluindo a planta simulada e o processo de obtenção dos dados de entrada e saída. Em seguida, são apresentados os resultados obtidos de cada implementação, em ponto flutuante e em ponto fixo no *matlab*, e dos blocos em VHDL.

4.1 CENÁRIO DE TESTES

Para a validação das implementações desenvolvidas, foi utilizado um conjunto de dados sintéticos gerados a partir da simulação de um conversor CC-CC do tipo *boost*, descrito na Seção 3.1. Os parâmetros do conversor utilizados na obtenção dos dados foram resistência de carga $R = 10\ \Omega$, capacitância $C = 470\ \mu\text{F}$, indutância $L = 0.75\ \text{mH}$, tensão de entrada $V_{in} = 5\ \text{V}$ e razão cíclica $D = 0.5$.

A função de transferência de pequenos sinais do conversor, dada pela Equação (12), foi discretizada pelo método ZOH com frequência de amostragem $f_s = 20\ \text{kHz}$, resultando na função de transferência discreta de segunda ordem dada por

$$H(z) = \frac{-0.1939z^{-1} + 0.2292z^{-2}}{1 - 1.988z^{-1} + 0.9894z^{-2}}. \quad (16)$$

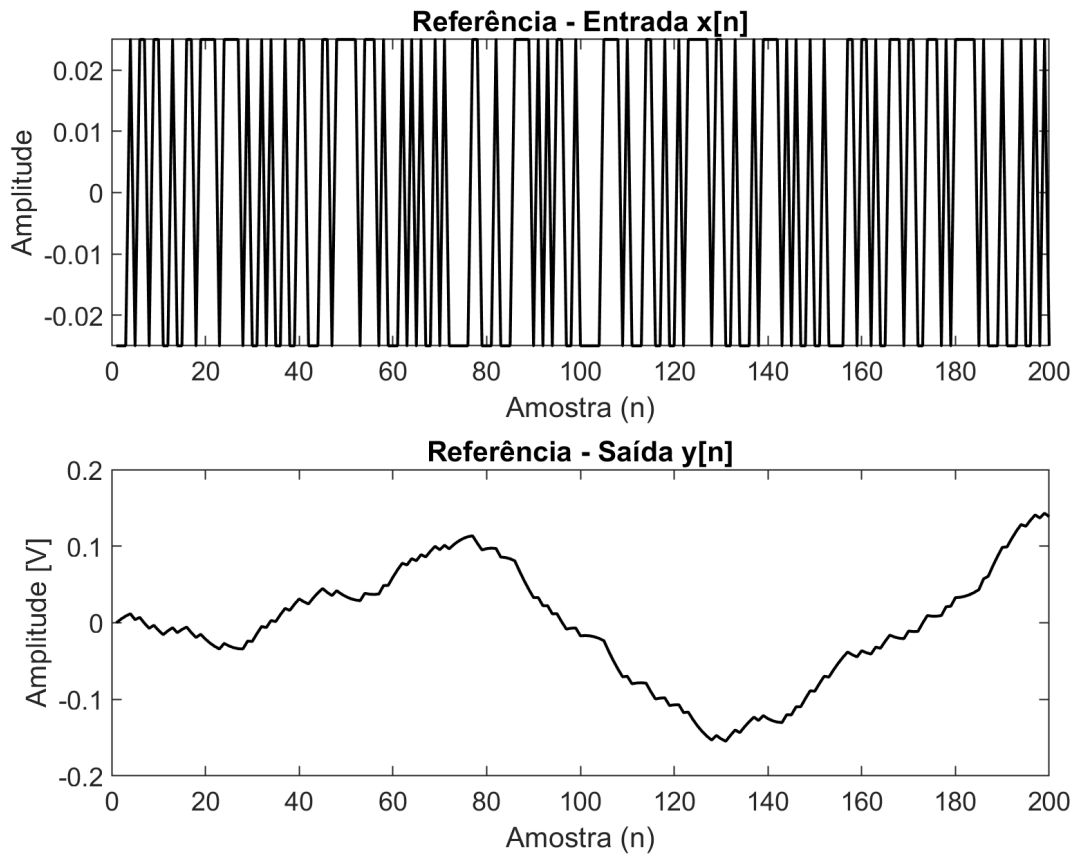
Os coeficientes $b_1 = -0,1939$, $b_2 = 0,2292$, $a_1 = -1.9888$ e $a_2 = 0.9894$ são os parâmetros que devem ser estimados pelo algoritmo PSO, sendo os valores teóricos adotados como referência para avaliar a adequação da estimação obtida pelo algoritmo implementado.

O sinal de entrada $x[n]$ empregado para excitar o sistema foi gerado por meio de uma sequência binária pseudoaleatória [PRBS (*Pseudo-Random Binary Sequence*)] de ordem 11, com amplitude de 0.05 em torno do ponto de operação. A resposta do sistema $y[n]$ foi obtida utilizando o sinal de entrada $x[n]$ na equação de diferenças relacionada com a função de transferência discreta (16), obtendo-se $N = 200$ amostras. A Figura 16 apresenta o comportamento da entrada e da saída utilizadas no algoritmo implementado para estimação dos parâmetros b_1 , b_2 , a_1 e a_2 .

Para utilizar os mesmos dados nas implementações em ponto flutuante, ponto fixo e em VHDL, as amostras de entrada e saída, assim como os parâmetros de referência da planta, foram convertidos para o formato Q12.20 e para implementação em VHDL exportados em arquivos de texto no formato *.MIF* (*Memory Initialization File*), contendo a representação binária de cada amostra. Isso permite que o mesmo conjunto de dados seja utilizado de forma equivalente nas três implementações, possibilitando comparar diretamente os resultados obtidos.

Os parâmetros do algoritmo PSO utilizados nas implementações em *Matlab* foram mantidos constantes, conforme apresentado na Tabela 7.

Figura 16 – Sinais de entrada e saída gerados.



Fonte: Elaborado pela autora (2026).

Tabela 7 – Parâmetros do algoritmo PSO.

Parâmetro	Valor
Número de iterações ($Runs$)	200
Número de partículas (P)	100
Dimensão do espaço de busca (K)	4
Fator de inércia (w)	0.7
Coefficiente cognitivo (c_1)	1.4
Coefficiente social (c_2)	1.4
Limiar superior ($threshold_high = [b_1, b_2, a_1, a_2]$)	$[0, 0.25, -1.8, 1]$
Limiar inferior ($threshold_low = [b_1, b_2, a_1, a_2]$)	$[-0.2, 0.2, -2, 0.8]$

Fonte: Elaborado pela autora (2026).

4.2 IMPLEMENTAÇÃO EM PONTO FLUTUANTE

A implementação em ponto flutuante, detalhada na Seção 3.2, foi executada utilizando os dados de simulação descritos anteriormente. A Tabela 8 apresenta a comparação entre os parâmetros reais da planta e os estimados pelo algoritmo, assim como o valor final da função custo (J) e o tempo de execução do algoritmo. Vale ressaltar que, apesar

de utilizar os mesmos dados de entrada $x[n]$ e saída $y[n]$, em razão da natureza estocástica do algoritmo, que utiliza números aleatórios uniformemente distribuídos para geração das posições iniciais de cada partícula e para os coeficientes r_1 e r_2 [utilizados na Equação (1)], podem ocorrer pequenas variações nos resultados a cada execução.

Tabela 8 – Resultados da estimação dos parâmetros utilizando o algoritmo PSO implementado em ponto flutuante no *Matlab*.

Parâmetro	Valor referência	Valor estimado	Erro absoluto
b_1	-0.1939	-0.1646	0.0293
b_2	0.2292	0.2000	0.0292
a_1	-1.9877	-1.9877	0.0000
a_2	0.9894	0.9894	0.0000
Melhor Custo J (best_global_cost)		0.00083	
Tempo de execução		4.15 segundos	

Fonte: Elaborado pela autora (2026).

A Figura 17 apresenta a comparação entre a saída medida $y[n]$ e a saída estimada $\hat{y}[n]$, calculada com os parâmetros identificados pela melhor partícula do enxame ao final das 200 iterações.

Observa-se que os resultados obtidos nessa implementação são próximos aos valores de referência, sendo o maior Erro Absoluto (EA) o do parâmetro b_1 que foi de $EA = 0.0293$. Nota-se da Figura 17 que a curva da saída estimada $\hat{y}[n]$ acompanha a curva da saída $y[n]$ (referência), obtendo um bom resultado para fins de comparação posterior com a implementação em ponto fixo, mostrado na próxima seção.

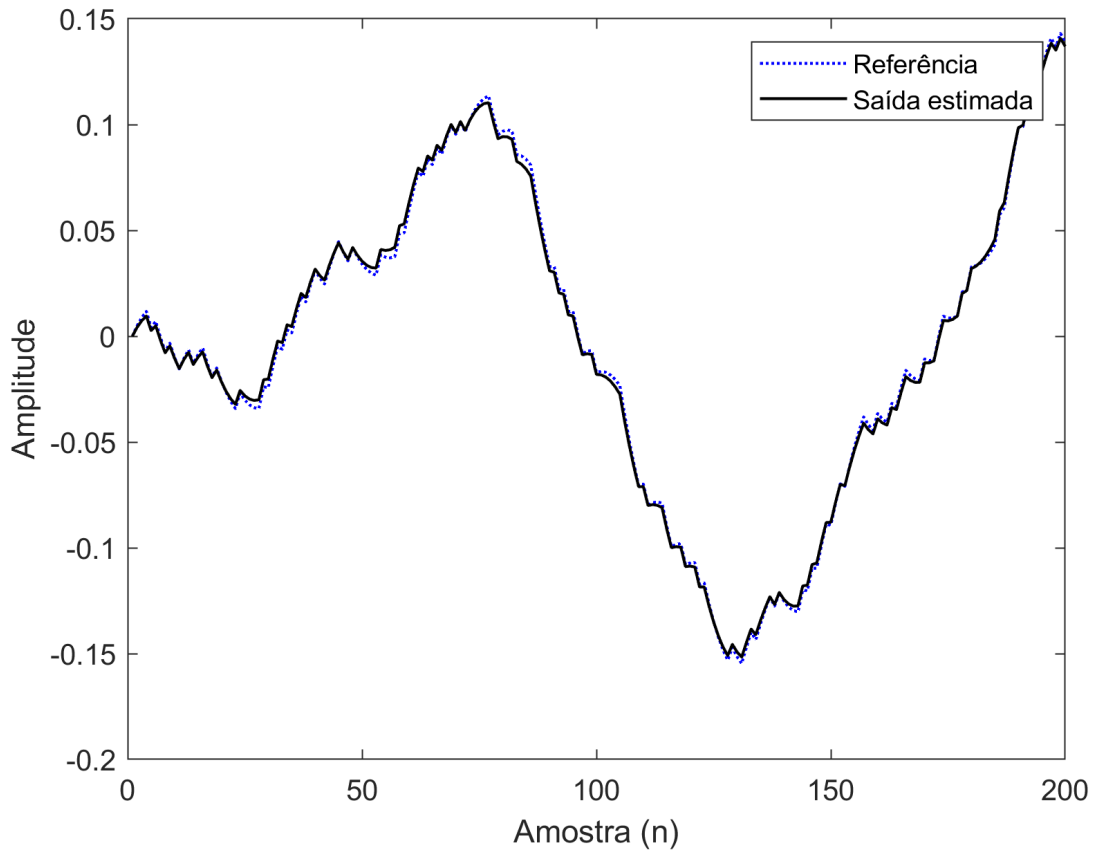
4.3 IMPLEMENTAÇÃO EM PONTO FIXO

A implementação em ponto fixo no *Matlab*, descrita na Seção 3.3, utilizando o formato Q12.20, foi executada com os mesmos dados e mesmos parâmetros do algoritmo PSO empregados na implementação em ponto flutuante. A Tabela 9 apresenta os resultados obtidos.

Tabela 9 – Resultados da estimação dos parâmetros - Ponto fixo no *Matlab*.

Parâmetro	Valor referência	Valor estimado	Erro absoluto
b_1	-0.1939	-0.1807	0.0132
b_2	0.2292	0.2245	0.0047
a_1	-1.9877	-1.9805	0.0072
a_2	0.9894	0.9821	0.0073
Melhor Custo J (best_global_cost)		0.0142	
Tempo de execução		93.39 segundos	

Fonte: Elaborado pela autora (2026).

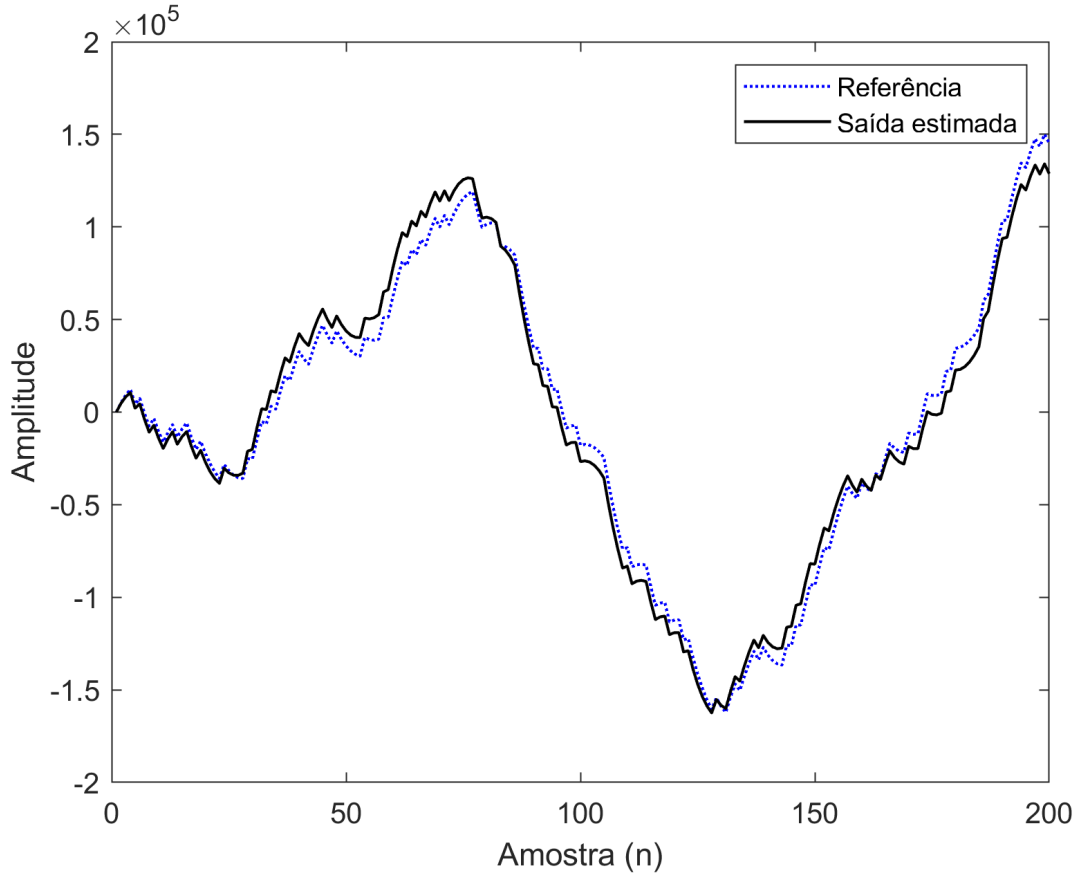
Figura 17 – Saída de referência e saída estimada - Ponto flutuante no *Matlab*.

Fonte: Elaborado pela autora (2026).

A Figura 18 apresenta as saídas de referência $y[n]$ e estimada $\hat{y}[n]$ resultante da implementação em ponto fixo, com os valores convertidos para ponto flutuante por meio da função descrita na Seção 3.3 para fins de visualização.

Observando os resultados das Tabelas 8 e 9, verifica-se que a implementação em ponto fixo obteve erros absolutos de mesma ordem de grandeza que a implementação em ponto flutuante, com erro máximo de 0.0132 no parâmetro b_1 , inferior ao erro máximo de 0.0293 observado na implementação em ponto flutuante. Esse resultado indica que o formato escolhido ($Q12.20$) foi adequado e o algoritmo conseguiu convergir para uma solução próxima aos valores de referência da planta.

Em relação ao valor do melhor custo (*best_global_cost*), nota-se que ambas as implementações obtiveram valores de custos baixos (0.00083 em ponto flutuante e 0.0142 em ponto fixo), refletindo a boa adequação da melhor solução do enxame em ambos os casos. Ademais, o tempo de execução do algoritmo em ponto flutuante é mais de vinte vezes menor do que o tempo observado na implementação em ponto fixo. Isso ocorreu, pois a implementação em ponto fixo implementada não está otimizada para execução plataforma

Figura 18 – Saída de referência e saída estimada - Ponto fixo no *Matlab*.

Fonte: Elaborado pela autora (2026).

utilizada (isto é, computador *desktop* com *Matlab*). De todo modo, a implementação em ponto fixo desenvolvida pode ser facilmente portada para ser executada com bom desempenho em plataformas computacionais embarcadas, que possuam apenas *hardware* para computação operações aritméticas em ponto fixo.

4.4 IMPLEMENTAÇÃO EM VHDL

Nesta etapa do trabalho foram implementados e validados individualmente os blocos *COST_FUNCTION* e *PARTICLE_UPDATE*, descritos na Seção 3.4. Dessa forma, os resultados das implementações em VHDL correspondem à validação funcional de cada bloco isoladamente, por meio de simulação no *software ModelSim*.

4.4.1 Validação do componente *multiplier*

O componente *multiplier*, responsável pelas operações de multiplicação executadas no bloco *PARTICLE_UPDATE*, foi validado por meio de um *testbench* (*tb_multiplier*),

no qual foram aplicados dois conjuntos de estímulos correspondentes aos dois modos de operação do sinal **FLAG**. No primeiro estímulo, com **FLAG** = '1', foram fornecidos os vetores $\mathbf{x}_1 = [4.0, 3.0, 2.0, 1.5]$ e $\mathbf{x}_2 = [2.0, 2.0, 2.0, 2.0]$, avaliando o produto elemento a elemento entre dois vetores. No segundo estímulo, com **FLAG** = '0', foi fornecido o escalar $x_1 = 1.5$, posicionado no elemento de índice zero de $\mathbf{x}_1$, e o vetor $\mathbf{x}_2 = [4.0, 3.0, 2.0, 1.0]$, avaliando o produto escalar-vetor. A Tabela 10 apresenta os valores esperados em ponto fixo, calculados pela Equação (15), e os valores obtidos na simulação no *ModelSim*.

Tabela 10 – Resultados da validação do componente *multiplier*.

Modo	Elemento	Valor esperado	Valor simulado
FLAG = '1'	$y[0] = 1.5 \times 2.0$	3145728 (3.0)	3145728
	$y[1] = 2.0 \times 2.0$	4194304 (4.0)	4194304
	$y[2] = 3.0 \times 2.0$	6291456 (6.0)	6291456
	$y[3] = 4.0 \times 2.0$	8388608 (8.0)	8388608
FLAG = '0'	$y[0] = 1.5 \times 1.0$	1572864 (1.5)	1572864
	$y[1] = 1.5 \times 2.0$	3145728 (3.0)	3145728
	$y[2] = 1.5 \times 3.0$	4718592 (4.5)	4718592
	$y[3] = 1.5 \times 4.0$	6291456 (6.0)	6291456

Fonte: Elaborado pela autora (2026).

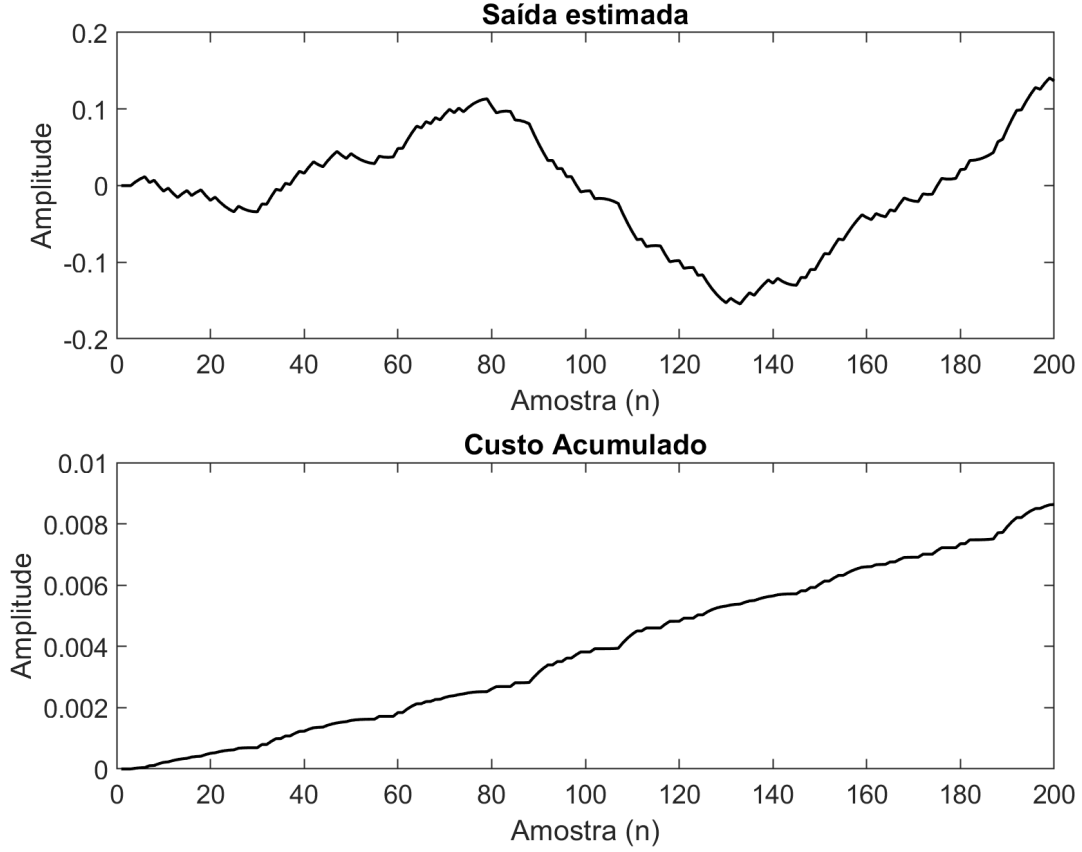
Os resultados obtidos confirmam que o componente *multiplier* reproduz corretamente os dois modos de multiplicação descritos na Seção 3.4, no modo de multiplicação escalar-vetor, o operando $x1$ permanece fixo no elemento de índice zero ao longo dos quatro ciclos de processamento, enquanto no modo de multiplicação elemento a elemento ambos os operandos são indexados pelo mesmo contador interno a cada ciclo. Em ambos estímulos de entrada, o sinal *finish* foi ativado exatamente após $K = 4$ ciclos de *clock*, conforme o comportamento planejado.

4.4.2 Validação do bloco *COST_FUNCTION*

O bloco *COST_FUNCTION* foi validado por meio de um *testbench* (*tb_cost_function*), no qual o bloco foi conectado a um componente *DATA_MEMORY*, responsável por fornecer sequencialmente as amostras de entrada $x[n]$ e saída $y[n]$ a partir dos dados salvos em arquivos do tipo .MIF, conforme descrito anteriormente. Os parâmetros $\hat{b}_1$, $\hat{b}_2$, $\hat{a}_1$ e $\hat{a}_2$ fornecidos ao bloco correspondem aos parâmetros de referência da planta em questão, convertidos para o formato *Q12.20*, permitindo avaliar se a saída estimada $\hat{y}[n]$ produzida pelo bloco reproduz adequadamente a saída medida $y[n]$ ao longo das $N = 200$ amostras.

A cada ciclo de *clock*, as saídas do bloco foram registradas em arquivos de texto (.TXT) e posteriormente importados para o *Matlab* para conversão em ponto flutuante, facilitando a visualização dos resultados gráficos. A Figura 19 apresenta as formas de onda obtidas na simulação no *ModelSim*, evidenciando a evolução da estimação da saída e do acúmulo do custo ao longo das 200 amostras processadas pelo bloco.

Figura 19 – Formas de onda da saída estimada $\hat{y}[n]$ e valor da função custo (ao longo do processamento das 200 amostras), obtidas via simulação do bloco *COST_FUNCTION* no *ModelSim*.

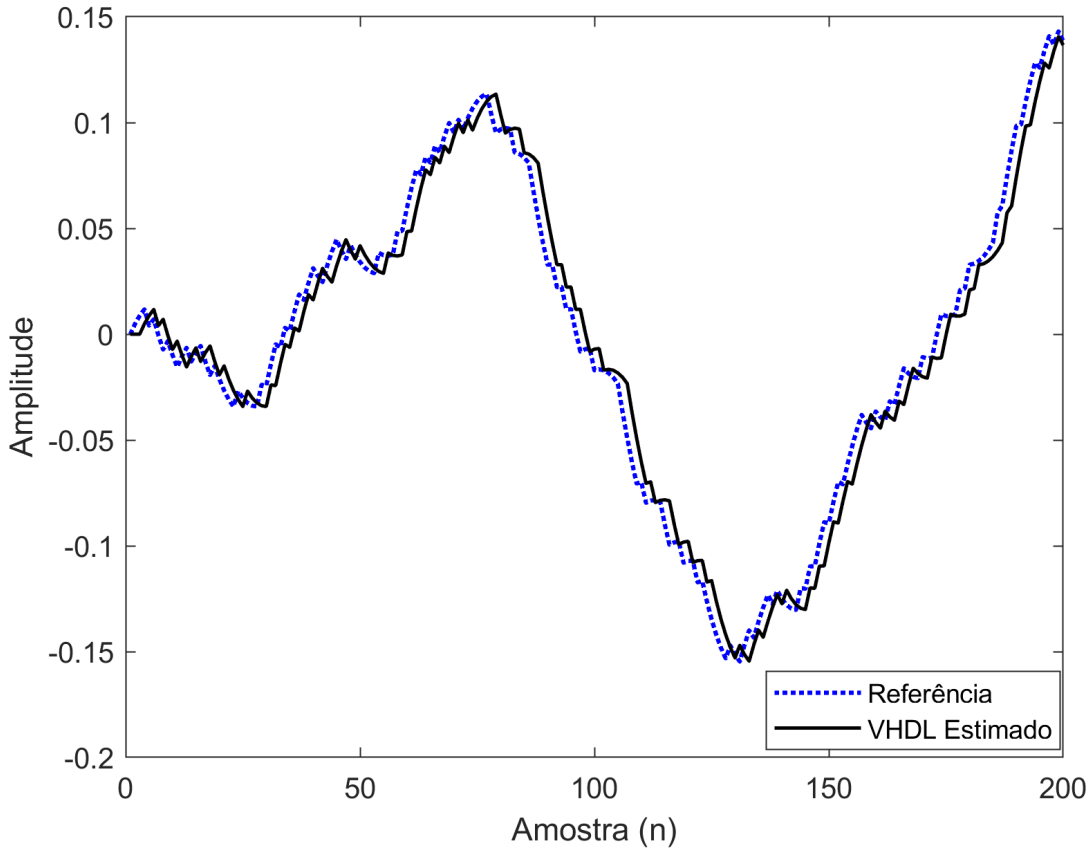


Fonte: Elaborado pela autora (2026).

Nota-se que o formato de onda da saída estimada $\hat{y}[n]$ obtida é consistente ao esperado (ou seja, está de acordo com as formas de onda obtidas nas implementações em ponto flutuante e em ponto fixo), além disso, o custo acumulado J cresce ao longo das amostras, que é o comportamento esperado, em que o erro quadrático de cada amostra é somado ao valor anterior.

Para verificar a fidelidade da saída obtida em relação à referência, a Figura 20 apresenta a saída estimada, obtida por meio do *testbench*, comparada ao sinal de referência. Observa-se que a saída estimada apresenta um atraso de duas amostras em relação à saída de referência $y[n]$. Esse comportamento é esperado do modelo utilizado (Equação (14)), que depende das amostras passadas $x[n - 1]$ e $x[n - 2]$, somado a um ciclo extra de latência introduzido pela atualização síncrona dos registradores do *buffer* (*buffer1*, *buffer2*, *buffer3*), nesta implementação só é refletida no próximo ciclo de *clock*.

Figura 20 – Saída de referência e saída estimada - Implementação VHDL.



Fonte: Elaborado pela autora (2026).

4.4.3 Validação do bloco *PARTICLE_UPDATE*

O bloco *PARTICLE_UPDATE* foi validado por meio de um *testbench*, no qual foram fornecidos, ao longo de quatro iterações sucessivas, valores de posição, velocidade, melhor posição pessoal e melhor posição global de uma partícula extraída de uma execução do algoritmo PSO em ponto fixo no *Matlab* com partícula $P = 1$ e gerador de números aleatórios baseado em um registrador de deslocamento de 20 bits com realimentação linear (LFSR - *Linear Feedback Shift Register*). Essa abordagem de teste permite reproduzir a mesma sequência de coeficientes aleatórios r_1 e r_2 tanto na simulação em VHDL quanto na execução de referência em *Matlab*, eliminando a aleatoriedade como fonte de diferença entre as duas implementações.

Os estímulos aplicados correspondem aos valores de entrada e saída registrados nas iterações 50, 51, 52 e 53, no *Matlab*, de uma execução completa do algoritmo, no qual $w = 0.7$ e $c_1 = c_2 = 1.4$ permanecem constantes. A Tabela 11 apresenta os valores intermediários (aux_1 , aux_{20} , aux_2 , aux_{30} , aux_3) e finais ($current_velocity$ e $current_position$) produzidos na simulação em VHDL para as quatro iterações avaliadas.

Tabela 11 – Comparação entre os resultados¹ da simulação em VHDL e da execução de referência em ponto fixo no *Matlab* para o bloco *PARTICLE_UPDATE*, iterações 50 a 53.

Elemento	Iteração 50				Iteração 51			
	$k = 0$	$k = 1$	$k = 2$	$k = 3$	$k = 0$	$k = 1$	$k = 2$	$k = 3$
<i>aux</i> ₁	356	-558	-1520	-376	24	-315	-813	-1168
<i>aux</i> ₂₀	55	24	294	-399	135	507	1501	654
<i>aux</i> ₂	76	33	411	-559	188	709	2101	915
<i>aux</i> ₃₀	-283	54	-37	-523	-301	329	262	478
<i>aux</i> ₃	-397	75	-52	-733	-422	460	366	669
<i>current_velocity</i>	35	-450	-1161	-1668	-210	854	1654	416
<i>current_position</i>	-202960	239782	-2085269	1036514	-203170	240636	-2083615	1036930

Elemento	Iteração 52				Iteração 53			
	$k = 0$	$k = 1$	$k = 2$	$k = 3$	$k = 0$	$k = 1$	$k = 2$	$k = 3$
<i>aux</i> ₁	-147	597	1157	291	-236	198	595	599
<i>aux</i> ₂₀	0	0	0	0	158	-269	-756	-665
<i>aux</i> ₂	0	0	0	0	221	-377	-1059	-931
<i>aux</i> ₃₀	-135	-223	-218	404	97	-64	-301	-128
<i>aux</i> ₃	-189	-313	-306	565	135	-90	-422	-180
<i>current_velocity</i>	-336	284	851	856	120	-269	-886	-512
<i>current_position</i>	-203506	240920	-2082764	1037786	-203386	240651	-2083650	1037274

Fonte: Elaborado pela autora (2026).

Os valores adquiridos na simulação em VHDL são iguais aos valores calculados pela referência no *Matlab*, para todos os elementos visualizados e nas quatro iterações avaliadas, confirmando que o bloco *PARTICLE_UPDATE* reproduz, em *hardware*, de maneira adequada a sequência de operações aritméticas descritas previamente. Esse resultado valida a estratégia escolhida de desenvolvimento de uma máquina de estados com a utilização sequencial do componente *multiplier* nos cálculos desse bloco.

Ademais, convertendo o resultado obtido (*current_position*) para ponto flutuante em todas as iterações, são obtidos os mesmos resultados, isso ocorre pois, ao longo das quatro iterações a posição da partícula permanece em uma vizinhança muito próxima aos parâmetros de referência da planta, comportamento esperado de uma partícula em estágio avançado de convergência do algoritmo PSO, conforme apresentado na Tabela 12.

Tabela 12 – Resultados do bloco *PARTICLE_UPDATE* - VHDL.

Parâmetro	Valor referência	Valor estimado	Erro absoluto
b_1	-0.1939	-0.1939	0.0000
b_2	0.2292	0.2292	0.0000
a_1	-1.9877	-1.9877	0.0000
a_2	0.9894	0.9894	0.0000

Fonte: Elaborado pela autora (2026).

¹ Os valores obtidos na simulação em VHDL são idênticos aos calculados pela referência no *Matlab* para todos os dados apresentados.

5 CONCLUSÃO

Este trabalho teve como objetivo geral implementar o algoritmo de Otimização por Enxame de Partículas (PSO) com aritmética de ponto fixo no formato $Q12.20$, aplicado à identificação de parâmetros de um conversor CC-CC do tipo *boost*, e desenvolver os principais blocos computacionais do algoritmo em linguagem de descrição de *hardware* VHDL. A partir dos resultados apresentados no Capítulo 4, verifica-se que os objetivos específicos propostos foram atendidos e que o objetivo geral foi alcançado.

A implementação do algoritmo PSO em aritmética de ponto flutuante no *Matlab* serviu como referência para comparação, demonstrando boa capacidade de convergência e estimação dos parâmetros da função de transferência da planta, com erros absolutos pequenos em relação aos valores de referência. A implementação subsequente em aritmética de ponto fixo, utilizando o formato $Q12.20$, apresentou resultados de qualidade compatível, com erros de estimação de mesma ordem de grandeza, indicando que a quantização introduzida pela representação em ponto fixo não comprometeu significativamente a capacidade do algoritmo convergir para uma solução próxima à referência da planta.

Outra contribuição deste trabalho, é a validação dos blocos *COST_FUNCTION* e *PARTICLE_UPDATE*, desenvolvidas em VHDL. Os resultados obtidos por meio de simulação no *ModelSim* demonstraram que ambos os blocos reproduzem, em *hardware*, os valores calculados pela implementação de referência em ponto fixo no *Matlab*, tanto para os valores intermediários das operações aritméticas quanto para os resultados finais de saída estimada, custo acumulado e velocidade e posição das partículas. Essa equivalência funcional evidencia que a estratégia de implementação adotada foi capaz de reproduzir em nível de descrição de *hardware*, o comportamento aritmético validado em *software*.

Como limitação, destaca-se que a integração completa do algoritmo PSO em *hardware* não foi realizada, tendo sido implementados e validados apenas os blocos individuais. A definição do bloco de controle responsável por gerenciar o laço de iterações sobre o enxame de partículas, a comparação e atualização das melhores posições pessoal e global, e a geração dos coeficientes aleatórios em *hardware*, constitui uma etapa necessária para a obtenção de um sistema PSO operando integralmente em FPGA.

Para trabalhos futuros, sugere-se:

- Implementar o bloco de controle responsável pelo laço de iterações e pela atualização das melhores posições, integrando os blocos já validados em um sistema PSO completamente em VHDL;
- Avaliar uma abordagem híbrida, na qual o *softcore* de microprocessador Nios II, sintetizado no próprio FPGA por meio do *Platform Designer* da Intel (Altera), seja responsável pelo controle do algoritmo em *software*, enquanto os blocos *COST_FUNCTION* e *PARTICLE_UPDATE* atuam como periféricos dedicados em *hardware*;

- Sintetizar o sistema resultante em um dispositivo FPGA físico, avaliando a utilização dos recursos lógicos;
- Investigar formatos de ponto fixo com maior faixa de representação para a variável de custo acumulado.

Conclui-se, portanto, que a aritmética de ponto fixo no formato $Q12.20$ constitui uma abordagem viável para a implementação do algoritmo PSO em sistemas embarcados, e que os blocos desenvolvidos e validados em VHDL representam um avanço em direção à execução completa do algoritmo em dispositivos FPGA. Por fim, o desenvolvimento realizado neste trabalho, incluindo as implementações em *Matlab* e em VHDL estão disponíveis no repositório [GitHub](#) do projeto.

REFERÊNCIAS

- AGUIRRE, Luis Antonio. **Introdução à Identificação de Sistemas: Técnicas Lineares e Não-Lineares Aplicadas a Sistemas Reais**. 3. ed. rev. e ampl. Belo Horizonte: Editora UFMG, 2007.
- ALTERA CORPORATION. **Cyclone IV Device Handbook, Volume 1, Chapter 4: Embedded Multipliers**. [S.l.: s.n.], 2010.
<https://docs.altera.com/v/u/docs/654776/cyclone-iv-device-handbook-volume-1-chapter-4-embedded-multipliers>. Acessado em: 14 de junho de 2026.
- COELHO, Antônio Augusto Rodrigues; SANTOS COELHO, Leandro dos. **Identificação de Sistemas Dinâmicos Lineares**. 1. ed. Florianópolis: Ed. da UFSC, 2004. P. 181.
- DA COSTA, Alexandre L. X.; SILVA, Caroline A. D.; TORQUATO, Matheus F.; FERNANDES, Marcelo A. C. Parallel Implementation of Particle Swarm Optimization on FPGA. **IEEE Transactions on Circuits and Systems II: Express Briefs**, v. 66, n. 11, p. 1875–1879, 2019.
- DAMAJ, Issam; ELSHAFEI, Mohamed; EL-ABD, Mohammed; AYDIN, Mehmet Emin. An analytical framework for high-speed hardware particle swarm optimization. **Microprocessors and Microsystems**, v. 72, p. 102949, 2020.
- HART, Daniel W. **Eletrônica de Potência: Análise e Projetos de Circuitos**. 1. ed. Porto Alegre: AMGH Editora Ltda., 2012.
- HOUSSEIN, Essam H.; GAD, Ahmed G.; HUSSAIN, Kashif; SUGANTHAN, Ponnuthurai Nagaratnam. Major Advances in Particle Swarm Optimization: Theory, Analysis, and Application. **Swarm and Evolutionary Computation**, v. 63, p. 100868, 2021. ISSN 2210-6502.
- JOHANSSON, Rolf. **System Modeling and Identification**. Englewood Cliffs, New Jersey: Prentice-Hall International, 1993. ISBN 0-13-145889-2.
- KENNEDY, J.; EBERHART, R. Particle swarm optimization. *In: PROCEEDINGS of ICNN'95 - International Conference on Neural Networks*. [S.l.: s.n.], 1995. v. 4, 1942–1948 vol.4.

LJUNG, Lennart. **System Identification: Theory for the User**. 2. ed. Upper Saddle River: Prentice Hall PTR, 1999. ISBN 0-13-656695-2.

PYEATT, Larry D.; UGHETTA, William. Chapter 8 - Non-integral mathematics. *In*: ARM 64-Bit Assembly Language. [S.l.]: Newnes, 2020. P. 239–292. ISBN 978-0-12-819221-4.

REDDIPOGU, Joshua Sunder David; ELUMALAI, Vinodh Kumar. Hardware in the Loop Testing of Adaptive Inertia Weight PSO-Tuned LQR Applied to Vehicle Suspension Control. **Journal of Control Science and Engineering**, Hindawi, v. 2020, p. 1–16, 2020. Acesso em: 9 maio 2026.

SILVA, Erick Jhone Rodrigues da. **Problemas de Precisão com Decimais no Desenvolvimento de Software**. Digital Innovation One. Abr. 2026. Disponível em: <https://www.dio.me/articles/problemas-de-precisao-com-decimais-no-desenvolvimento-de-software-f9ea316be9d0>. Acesso em: 27 jun. 2026.