UNIVERSIDADE FEDERAL DE OURO PRETO
ESCOLA DE MINAS
COLEGIADO DO CURSO DE ENGENHARIA DE CONTROLE
E AUTOMAÇÃO - CECAU

GUILHERME AUGUSTO LOPES SILVA

ANÁLISE DE ALGORITMOS EVOLUCIONÁRIOS E DE BUSCA LOCAL PARA O AJUSTE DO CONTROLADOR PID DE UM SISTEMA CÍBER-FÍSICO EM UMA ABORDAGEM HARDWARE-IN-THE-LOOP

MONOGRAFIA DE GRADUAÇÃO EM ENGENHARIA DE CONTROLE E AUTOMAÇÃO

Ouro Preto, 2020

GUILHERME AUGUSTO LOPES SILVA

ANÁLISE DE ALGORITMOS EVOLUCIONÁRIOS E DE BUSCA LOCAL PARA O AJUSTE DO CONTROLADOR PID DE UM SISTEMA CÍBER-FÍSICO EM UMA ABORDAGEM HARDWARE-IN-THE-LOOP

Monografia apresentada ao Curso de Engenharia de Controle e Automação da Universidade Federal de Ouro Preto como parte dos requisitos para a obtenção do Grau de Engenheiro de Controle e Automação.

Orientador: Prof. Dr. Eduardo José da Silva Luz

Coorientador: Prof. Dr. Alan Kardek Rêgo Segundo

Ouro Preto
Escola de Minas – UFOP
2020

SISBIN - SISTEMA DE BIBLIOTECAS E INFORMAÇÃO

S586a Silva, Guilherme Augusto Lopes.
Análise de algoritmos evolucionários e de busca local para o ajuste do controlador PID de um Sistema Cíber-Físico em uma abordagem Hardware-in-the-loop. [manuscrito] / Guilherme Augusto Lopes Silva. - 2020.
59 f.: il.: color., gráf., tab..

Orientador: Prof. Dr. Eduardo José da Silva Luz.
Coorientador: Prof. Dr. Alan Kardek Rêgo Segundo.
Monografia (Bacharelado). Universidade Federal de Ouro Preto. Escola de Minas. Graduação em Engenharia de Controle e Automação .

1. Teoria do controle. 2. Controladores PID. 3. Algoritmos. 4. Programação heurística - Busca em Vizinhaça Variável. I. Luz, Eduardo José da Silva. II. Rêgo Segundo, Alan Kardek. III. Universidade Federal de Ouro Preto. IV. Título.

CDU 681.5:004.02

Bibliotecário(a) Responsável: Sione Galvão Rodrigues - CRB6 / 2526

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE OURO PRETO
REITORIA
ESCOLA DE MINAS
DEPARTAMENTO DE ENGENHARIA CONTROLE E AUTOMACAO

FOLHA DE APROVAÇÃO

Guilherme Augusto Lopes Silva

ANÁLISE DE ALGORITMOS EVOLUCIONÁRIOS E DE BUSCA LOCAL PARA UM SISTEMA CÍBER-FÍSICO EM UMA ABORDAGEM HARDWARE-IN-THE-LOOP

Membros da banca

Eduardo José da Silva Luz - Doutor - UFOP
Alan Kardek Rêgo Segundo - Doutor - UFOP
Agnaldo José da Rocha Reis - Doutor - UFOP
Bruno Nazário Coelho - Doutor - UFOP

Versão final
Aprovado em 16 de novembro de 2020

De acordo

Professor Orientador

Documento assinado eletronicamente por **Alan Kardek Rego Segundo**, **PROFESSOR DE MAGISTERIO SUPERIOR**, em 16/11/2020, às 17:29, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

A autenticidade deste documento pode ser conferida no site http://sei.ufop.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0 , informando o código verificador **0102993** e o código CRC **58F4EA8C**.

Referência: Caso responda este documento, indicar expressamente o Processo nº 23109.008670/2020-13 SEI nº 0102993

R. Diogo de Vasconcelos, 122, - Bairro Pilar Ouro Preto/MG, CEP 35400-000
Telefone: 3135591533 - www.ufop.br

AGRADECIMENTOS

Primeiramente, agradeço a Deus por ter me acompanhado durante todo o caminho até aqui.

Aos meus pais Sivanil e Andrea por sempre estarem ao meu lado e pelo amor incondicional. Nada disso seria possível sem vocês.

Ao meu irmão Pedro, por ser meu melhor amigo, por sempre me incentivar e por não medir esforços para me ajudar.

Às minhas avós Zinha (in memorian) e Nenzinha, que me ensinaram muito sobre amor e carinho.

Aos meus tios e tias que sempre me apoiaram.

À Lívia por me ajudar nos momentos difíceis dessa jornada e por sempre me apoiar.

Aos amigos da automação, em especial, Irã, Matheus, Sofia e Wellington que foram fundamentais durante essa jornada.

Aos amigos do ITV, em especial Filipe, Maurício e Wagner pelo companheirismo e por contribuírem muito para o meu crescimento profissional.

Aos amigos do CSI-lab, em especial, Eduardo e Gladston que foram muito importantes para que eu chegasse até aqui e me interessasse pela área científica. Além disso, agradeço o empenho que eles dedicaram à elaboração deste trabalho.

Ao meu coorientador Alan pelo empenho para a elaboração deste trabalho.

“Nenhum talento supera a dedicação.” (Autor Desconhecido)

RESUMO

Ajustar o controlador Proporcional Integral Derivado, ou PID, em sistemas ciber-físicos é um grande desafio, uma vez que pode ser necessário que se tenha um conhecimento aprofundado não só do modelo matemático, mas também do modelo físico do equipamento sob estudo. Sendo assim, ajustar um controlador PID se torna um obstáculo ainda maior para quem não é especialista da área de controle. Nesse sentido, vários autores na literatura têm mostrado que algoritmos de otimização são eficientes para autoajustar as constantes do controlador PID, principalmente quando o modelo matemático do equipamento é desconhecido. Dentre os métodos de otimização, o algoritmo Busca por Vizinhança Variável (em inglês, *Variable Neighborhood Search* ou VNS) apresenta robustez e eficiência para resolver problemas de otimização. Porém, a literatura carece de trabalhos que mostrem sua eficiência para ajustar um controlador PID. Neste trabalho, investiga-se a eficiência de um VNS em relação à algoritmos evolutivos para sintonizar um controlador PID de um sistema ciber-físico real em um cenário hardware-in-the-loop. Experimentos revelaram que o VNS estabilizou a planta e a deixou mais estabilizada do que as melhores soluções encontradas pelos métodos evolucionários. Além disso, a abordagem proposta pode ser utilizada em sistemas reais ou comerciais, auxiliando na sintonia do controlador para novas mudanças de ambiente ou mesmo modificações de última hora em projetos.

Palavras-chaves: VNS. Algoritmos Evolutivos. Sistemas ciber-físico. controle Proporcional Integral Derivativo. Autoajuste. Hardware-in-the-loop.

ABSTRACT

Adjusting the Proportional Integral Derivative Controller, or PID, in cyber-physical systems is a great challenge, since it may be necessary to have an in-depth knowledge not only of the mathematical model, but also of the physical model of the equipment under study. Therefore, adjusting a PID controller becomes an even greater obstacle for those who are not specialists in the control area. In this sense, several authors in the literature have shown that optimization algorithms are efficient for self-adjusting the constants of the PID controller, especially when the mathematical model of the equipment is unknown. Among the optimization methods, the Variable Neighborhood Search algorithm (VNS) has robustness and efficiency to solve optimization problems. However, the literature lacks studies that show its efficiency to adjust a PID controller. In this work, we investigate the efficiency of a VNS in relation to evolutionary algorithms to tune a PID controller of a real cyber-physical system in a hardware-in-the-loop scenario. Experiments revealed that VNS stabilized the plant and left it more stabilized than the best solutions found by evolutionary methods. In addition, the proposed approach can be used in real or commercial systems, helping to tune the controller for new environment changes or even last-minute changes to projects.

Key-words: VNS. Evolutionary Algorithms. cyber-physical systems. Proportional Integral Derivative. Self-tuning. Hardware-in-the-loop.

LISTA DE ILUSTRAÇÕES

Figura 1 – Representação do controle PID. 17
Figura 2 – Exemplo de objetos 19
Figura 3 – Representações de um gene. 22
Figura 4 – Cruzamento. 23
Figura 5 – Fluxograma de um Algoritmo Genético genérico. 25
Figura 6 – Representações de um gene. 26
Figura 7 – Comportamento das partículas de um enxame. 27
Figura 8 – Fluxograma de um Algoritmo de Otimização por Enxame de Partículas. . . 29
Figura 9 – PID aplicado para estabilizar o birrotor. 32
Figura 10 – Forma geral do funcionamento do sistema. 32
Figura 11 – Base do sistema. 33
Figura 12 – Parafuso utilizado para fixação da base. 33
Figura 13 – Barra cilíndrica utiliza para fixar o birrotor na base. 33
Figura 14 – MPU 6050 utilizada. 34
Figura 15 – Controlador eletrônico de velocidade utilizado para acionar os motores. . . . 34
Figura 16 – Motor sem escova utilizado. 35
Figura 17 – Ligação dos fios de acionamento das bobinas dos motores para se alterar o sentido de rotação dos motores. 35
Figura 18 – Conjunto de hélices utilizadas no motor. 36
Figura 19 – Esquema do birrotor 37
Figura 20 – Header da classe GA. 38
Figura 21 – Header da classe I2C. 40
Figura 22 – Header da classe Individual. 41
Figura 23 – Header da classe Motor. 41
Figura 24 – Header da classe PID. 42
Figura 25 – Header da classe PSO. 44
Figura 26 – Header da classe Particle. 45
Figura 27 – Header da classe TiltSensor. 47
Figura 28 – Header da classe VNS. 48
Figura 29 – Média e desvio padrão ao longo das gerações dos melhores testes de GA e PSO 54
Figura 30 – Comparando a primeira solução com a solução gerada pelo VNS-PID 55

LISTA DE TABELAS

Tabela 1 – Resultados das cinco execuções do VNS-PID no birrotor 52
Tabela 2 – Resultados das cinco execuções do PSO-PID no birrotor 53
Tabela 3 – Resultados das cinco execuções do GA-PID no birrotor 53

LISTA DE ABREVIATURAS E SIGLAS

ASA	Adaptive Simulated Annealing
CC	Corrente Contínua
CPS	Cyber-Physical Systems
GA	Genetic Algorithm
I2C	Inter-Integrated Circuit
IGA	Improved Genetic Algorithm
PID	Proportional Integral Derivative
POO	Programação Orientada a Objetos
PSO	Particle Swarm Optimization
ROV	Remote Operated Vehicles
VNS	Variable Neighborhood Search

SUMÁRIO

1 INTRODUÇÃO 12

2 REVISÃO DA LITERATURA 15

2.1 Trabalhos Relacionados 15

2.2 Controlador PID 17

2.3 Programação Orientada a Objetos (POO) 18

2.4 Busca por Vizinhança Variável para a Calibração de um PID (VNS-PID) . . 19

2.5 Algoritmos Evolutivos 21

2.5.1 Algoritmo Genético para a calibração de um PID (GA-PID) 22

2.5.2 Otimização por Enxame de Partículas para a calibração de um PID (PSO-PID) 24

2.6 Filtro de Kalman 30

3 DESENVOLVIMENTO 31

3.1 Controlador PID para Sistemas Reais 31

3.2 Hardware do birrotor 32

3.3 Código elaborado para sintonizar o controlador PID do birrotor 36

3.4 Código do VNS-PID 49

3.5 Código GA-PID 51

3.6 Código PSO-PID 51

4 RESULTADOS 52

5 CONCLUSÃO 56

REFERÊNCIAS 57

1 INTRODUÇÃO

Nos últimos anos, o acesso à informação tornou-se mais fácil para pessoas de todo o mundo. Esse cenário permitiu que entusiastas em tecnologia não só compartilhassem seus projetos, mas também procurassem outros projetos desse tipo para aprender. Com base nisso, os projetos de sistemas ciber-físicos [1] foram amplamente compartilhados e adaptados. Um CPS é um sistema computacional e colaborativo, em que as suas atividades são monitoradas, coordenadas, controladas e integradas por meio de núcleos de comunicação e computação, que controlam uma planta física (LEE; SESHIA, 2016). Alguns exemplos de sistemas ciber-físicos são: robôs, drones e quaisquer estruturas eletrônicas com sensores, atuadores e comunicação em uma rede. Nesse sentido, é necessário que se tenha não só um conhecimento da modelagem matemática do sistema a ser construído, mas também de sua integração com um sistema embarcado.

Tendo isso em mente, a replicação de um CPS por parte de pessoas que não sejam especialistas pode ser complexa, visto que, geralmente, é gerado um modelo físico diferente daquele proposto em manuais, seja por customizações ou erro de projeto e, por isso, faz-se necessário uma nova modelagem matemática. Além disso, mesmo que a modelagem matemática seja idêntica, a mudança no ambiente em que o CPS está inserido pode afetar o seu comportamento. Outro fator que pode dificultar essa modelagem é a existência de anomalias no sistema, como uma região mais pesada que a outra, que pode ocorrer para o caso de um drone, por exemplo. Esses cenários podem ser obstáculos para quem não é especialista e até mesmo para um especialista, conforme descrito por Hernández-Alvarado et al. (2016), no qual os autores mostram a dificuldade de se ajustar manualmente um CPS. Neste trabalho, o CPS é um Veículo Operado Remotamente [2] usado em tarefas submarinas. Nesse cenário, as condições ambientais alteram as características da planta devido aos vazamentos de óleo, por exemplo. No ROV proposto pelos autores, os sensores da planta devem fornecer informações para o sistema embarcado, que, por meio delas, gera uma resposta de controle para os atuadores de acordo com o algoritmo de controle implementado. Essa abordagem é conhecida como *hardware-in-the-loop*.

Nesse sentido, um cenário *hardware-in-the-loop* consiste em uma técnica utilizada no desenvolvimento e ensaios de sistemas embarcados complexos em tempo real. Tendo isso em mente, sinais reais de um controlador são conectados a um sistema de teste que simula a realidade, fazendo com que o controlador "pense"que está no produto real. Com isso, pode-se percorrer milhares de cenários possíveis para avaliar adequadamente o controlador sem o custo e o tempo associados aos testes físicos reais. Nesse contexto, uma abordagem *hardware-in-the-loop* é apresentada por Stewart, Stone e Fleming (2004), na qual é projetado um controlador fuzzy direcionado para controlar o hardware e um otimizador que adapta as configurações da lógica

[1] (em inglês, *Cyber Physical Systems* ou CPS)
[2] (em inglês, *Remote Operated Vehicles* ou ROV)

fuzzy do controlador.

Devido à complexidade do processo de calibração de uma planta física, diversas abordagens heurísticas foram propostas para o ajuste de um controlador Proporcional Integral Derivativo (PID). O método mais usado na indústria é o método de Ziegler-Nichols (ZIEGLER; NICHOLS, 1942), o qual é baseado na curva de reação do sistema. No entanto, essa abordagem consome um tempo considerável para realizar a sintonia de um controlador PID, visto que envolve ensaios em malha aberta para se obter os parâmetros do sistema e de sintonia do controlador. Por esse motivo, o processo de calibração do PID seria trabalhoso para sistemas que precisem de calibração contínua devido a mudanças sistemáticas em seu ambiente. Além disso, o método Ziegler-Nichols não garante a calibração ideal para todos os casos e, ocasionalmente, requer uma ferramenta auxiliar para ajustar o controlador (OGATA; YANG, 2002; CASTRO et al., 2019; LIMA et al., 2019).

Ang, Chong e Li (2005) apresentam uma visão geral das tecnologias computacionais que envolvem a calibração de um controlador PID. De acordo com os autores, diversos estudos foram feitos aplicando-se diferentes técnicas de otimização com o objetivo de facilitar e/ou otimizar o ajuste dos ganhos de um controlador PID. Dentre as técnicas, foram destacadas: Redes Neurais (SHU; PI, 2000; ZHAO et al., 2015; FANG; ZHUO; LEE, 2010), Lógica Fuzzy (VISIOLI, 2001; FEREIDOUNI; MASOUM; MOGHBEL, 2015; CASTELLANOS; BALLESTEROS, 2019), Algoritmo Genético [3] (KROHLING; REY, 2001; AMARAL; TANSCHEIT; PACHECO, 2018; KUMARI et al., 2018) e Otimização por Enxame de Partículas [4] (GIRIRAJKUMAR; JAYARAJ; KISHAN, 2010; MUKHTAR; TAYAL; SINGH, 2019).

Dentre os métodos explorados na literatura para otimização de valores reais, aqueles baseados em Pesquisa de Vizinhança Variável [5] apresentaram soluções muito próximas ao valor ótimo com um custo computacional moderado em diversos contextos (HANSEN; MLADENOVIĆ, 2001; BRIMBERG et al., 2019; PEI et al., 2019). Fortes et al. (2018) propõe o uso do VNS para sintonizar os parâmetros de dois controladores de amortecimento suplementares: um estabilizador de sistema de energia [6] e um controlador de fluxo de potência entre malhas - amortecimento de oscilação de potência [7] em um cenário ou IPFC-POD) com o objetivo de amortecer sinais locais e os modos de oscilação entre áreas presentes nos dois sistemas de potência estudados. O VNS foi comparado com um método multi-início [8]. Os resultados mostraram que não só o VNS foi mais eficiente, mas também gerou soluções com níveis mais altos de amortecimento. Dessa forma, com base nesses resultados propõe-se neste trabalho a investigação da viabilidade de se aplicar o VNS para a sintonia de um controlador PID aplicado a sistemas reais instáveis, como drones, em um cenário de um CPS. Além disso, como dito anteriormente,

[3] (em inglês, *Genetic Algorithm* ou GA)
[4] (em inglês, *Particle Swarm Optimization* ou PSO)
[5] (em inglês, *Variable Neighborhood Search* ou VNS)
[6] (em inglês, *Power System Stabilizer* ou PSS)
[7] (em inglês, *Interline Power Flow Controller–Power Oscillation Damping*
[8] (em inglês, *multi-start method*)

o uso de Algoritmos Evolucionários como GA e PSO são frequentemente empregados para a sintonia de um controlador PID.

Com isso, os objetivos deste trabalho são:

- Propor um método eficaz para sintonia de um controlador PID, utilizando VNS;
- Comparar o método proposto com os seguintes algoritmos evolutivos: GA e PSO;
- Elaborar uma função objetivo eficaz para o contexto deste trabalho.

Com base nos objetivos e considerando o cenário exposto, as contribuições deste trabalho podem ser sumarizadas como:

- Proposta de um método, baseado em VNS, para ajustar o PID de uma planta real (birrotor);
- Análise da proposta em um sistema *hardware-in-the-loop*;
- Proposta de uma função objetivo (ou aptidão) eficiente para o problema em questão;
- Comparação da proposta baseada em VNS contra uma alternativa baseada em algoritmos evolucionários (GA e PSO);

O trabalho está organizado da seguinte maneira. A Seção 2 apresenta os trabalhos relacionados a esta proposta e apresenta uma visão geral sobre VNS e os Algoritmos Evolucionários utilizados neste trabalho. A Seção 3 descreve o problema tratado neste trabalho e a construção do protótipo utilizado. Além disso, essa seção detalha os experimentos das abordagens propostas para otimização das constantes PID. Uma discussão sobre os experimentos e seus resultados é apresentada na Seção 4. Finalmente, a última seção, Seção 5, apresenta a conclusão e os trabalhos futuros.

2 REVISÃO DA LITERATURA

2.1 Trabalhos Relacionados

Em relação às técnicas de otimização e calibração de um controlador PID, vários autores contribuíram efetivamente para esse problema.

Zhang et al. (2009) empregou um GA auto-ajustado para otimizar os parâmetros PID, o que melhora a eficiência da pesquisa global. Seus resultados mostraram o seu potencial para o uso colaborativo com os controladores existentes para ajuste de PID.

Angel, Viola e Vega (2019) apresentaram uma abordagem *hardware-in-the-loop* para o controle da velocidade de um sistema motor-gerador. Eles usaram um controlador PID e um Algoritmo Genético (PID-GA) para encontrar as melhores constantes PID para um tempo específico de ultrapassagem e acomodação. Além disso, os autores compararam seu modelo com um controlador PID ajustado pela técnica Internal Model Control (PID-IMC). Os resultados mostraram que o PID-GA tem melhor desempenho de rastreamento do que o PID-IMC.

Um ajuste automático das constantes PID usando o Adaptive Simulated Annealing (ASA) é aplicado por Fraga-Gonzalez et al. (2017). Os autores compararam alguns métodos diferentes de ASA com algumas outras técnicas, como o método de Ziegler-Nichols. Eles usaram 20 sistemas de benchmark comumente encontrados no setor industrial para testar cada abordagem. Os testes realizados em quatro cenários do benchmark demonstraram a viabilidade do ASA no ajuste de controladores PID para plantas com distúrbios constantes. O algoritmo proposto relacionado ao ASA apresentou melhores resultados em relação ao método Ziegler-Nichols.

Um controlador PID baseado em lógica fuzzy para otimizar os ganhos de um controlador PID de um sistema elétrico de potência é apresentado por Osinski, Leandro e Oliveira (2019). Os autores utilizaram um modelo matemático baseado em informações estruturais de uma usina hidrelétrica brasileira como base de testes. O algoritmo apresentado foi comparado com um controlador PID convencional implementado na usina hidrelétrica de estudo. Os resultados mostraram que o controlador fuzzy apresentou respostas não só mais suaves, mas também menos oscilantes.

Rosales et al. (2018) apresenta um controle PID neural adaptativo projetado para sistemas configurados em tempo discreto. Esse controlador proporciona uma implementação mais eficiente, ao passo que evita o problema de implementação de controladores projetados para tempo contínuo em sistemas digitais. O controlador ajusta os ganhos PID de forma automática. Dessa forma, qualquer pessoa sem conhecimento prévio da planta é capaz de ajustar tais constantes. Os experimentos foram feitos em um quadrotor e mostraram a eficácia desse método em relação a um PID de ganho fixo clássico, tendo em vista que o desempenho do primeiro é melhor em

relação ao segundo.

Um GA melhorado (em inglês, *Improved Genetic Algorithm*, ou IGA) baseado em um controlador PID é apresentado por Feng et al. (2018). A população inicial do IGA é gerada próxima de uma solução encontrada pelo método de Ziegler-Nichols. O objetivo do algoritmo proposto é melhorar o desempenho do rastreamento de trajetória de uma escavadeira robótica. Os autores compararam o IGA com dois outros métodos: Algoritmo genético padrão (em inglês, *Standard genetic algorithm*, ou SGA) e Ziegler-Nichols. Os resultados mostraram que o IGA é efetivo na melhora da precisão do rastreamento. Além disso, o IGA obteve não só o maior controle de precisão, mas também o menor erro de rastreamento dentre os métodos analisados.

Um controle PID robusto e efetivo utilizando PSO para otimizar seus ganhos é implementado para um Reator de água pressurizada (em inglês, *Pressurized water reactor* ou PWR) por Mousakazemi e Ayoobian (2019). Os experimentos mostraram que o PSO não precisou conhecer o mecanismo do sistema dinâmico para obter uma resposta ideal. Além disso, as simulações feitas revelaram a robustez do algoritmo proposto, especialmente contra as incertezas paramétricas.

Mukhtar, Tayal e Singh (2019) apresentam um PSO para ajustar um controlador PID (PSO-PID) de um tanque de nível de líquido de processo é proposto. Os autores compararam a metodologia elaborada com um PID convencional. Ambos os métodos foram aplicados no MATLAB/Simulink. Os experimentos mostraram que a resposta gerada pelo PID convencional obteve maior tempo de estabilização e sobressinal do que o PSO-PID.

Zahir et al. (2018) propõe um GA para sintonizar as constantes PID de um Motor DC sem escova. Uma comparação é feita entre o método proposto e métodos clássicos para o ajuste de um controlador PID como Zigler-Nichols. A função de aptidão do algoritmo proposto é influenciada de acordo com os valores de: tempo de subida, tempo de acomodação, sobressinal e erro em regime permanente do indivíduo analisado. Os experimentos revelaram que a metodologia apresentada pelo autor obteve resultados melhores considerando os seguintes parâmetros: tempo de subida, tempo de acomodação e sobressinal. No entanto, seu erro em regime permanente foi o maior dentre todos os métodos clássicos analisados.

Zeng et al. (2015) propõe um controlador PID de ordem fracionária (FOPID) com base em um algoritmo aprimorado de otimização multiobjetivo para um sistema de regulador de tensão automático. Eles demonstraram a eficácia do controlador MOEO-FOPID proposto em termos de precisão e robustez em comparação ao FOPID baseado em NSGA-II, controladores FOPID baseados em algoritmos evolutivos de objetivo único, controladores PID baseados em MOEO e NSGA-II.

2.2 Controlador PID

O controlador PID (Figura 9) é um sistema de controle em malha fechada que fornece uma correção da resposta de uma determinada entrada. Essa correção é baseada em três termos: o proporcional (K_p), o integral (K_i) e o derivativo (K_d). O ajuste do PID é a ação de equilibrar essas três constantes com a intenção de melhorar a resposta do sistema. A Equação 2.1 mostra o modelo matemático calculado em cada loop de um controlador PID:

$$u(t) = K_p e(t) + K_i \sum_{k=0}^{t} e(t)\Delta t + K_d \frac{\Delta e(t)}{\Delta t} \tag{2.1}$$

em que $u(t)$ é a saída da função PID, $e(t)$ é o erro entre a referência desejada e a saída do sistema, $\Delta e(t)$ é a variação do erro no intervalo de amostragem e Δt é o tempo de amostragem do sistema.

A Equação 2.1 é apresentada esquematicamente na Figura. 1. O controlador PID fornece uma resposta ajustada, $u(t)$, para a planta, a fim de minimizar o erro entre a referência desejada, $x(t)$, e a saída do sistema (variável de processo medida), $y(t)$.

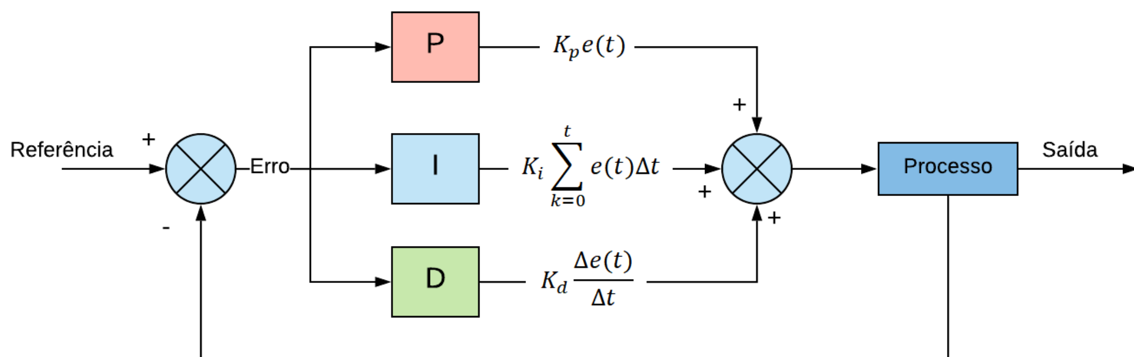


Figura 1 – Representação do controle PID.

Alguns sistemas apresentam algumas características específicas, como: sobressinal e erro em regime permanente. O sobressinal pode ser definido como a maior diferença entre a posição do birrotor e sua resposta em regime permanente e o erro em regime permanente é o erro entre o sistema e sua referência após a estabilização.

Tendo isso em mente, o impacto da alteração de uma das constantes do controlador PID pode alterar uma dessas características específicas, alterando o comportamento do sistema (ANG; CHONG; LI, 2005). Por exemplo, o aumento do ganho proporcional (K_p) diminui o tempo de subida, pois a resposta do sistema se torna mais forte e mais rápida a um estímulo externo, uma vez que há um aumento da potência dos atuadores. No entanto, o sobressinal aumenta e o sistema tende a ficar mais instável. Portanto, em um sistema do tipo 0 (sem integrador na função de transferência em malha aberta), o ganho proporcional não poderá levar o erro em regime

permanente para zero por si só. Por esse motivo, geralmente, ele é utilizado junto a um termo Integral.

O ganho integral tem uma influência maior na resposta do sistema quando o erro é pequeno e o sistema está próximo da referência. O termo K_i soma o erro ao longo do tempo, para levá-lo a zero. Esse cenário força uma resposta mais abrupta ao longo do tempo, uma vez que o erro é diferente de zero. Por esse motivo, à medida que K_i aumenta, uma pequena diminuição no tempo de subida é observada, o erro em regime permanente diminui e o sobressinal aumenta.

Uma vantagem do controle integral é que ele introduz um termo s no denominador, aumentando o tipo do sistema em uma unidade. O tipo do sistema é dado pelo número de integradores puros presentes no denominador. Caso o sistema seja do tipo 0, o controlador integral elimina o erro estacionário que deveria ocorrer para uma entrada em degrau, se fosse utilizado unicamente o tipo de controlador proporcional. Porém, uma desvantagem deste controlador é a introdução de um polo na origem. Se nenhum zero for introduzido, a diferença entre o número de polos e zeros é aumentada, diminuindo os ângulos das assíntotas dos lugares das raízes. Desse modo, os ângulos das assíntotas apontam mais em direção ao semiplano direito do plano S, reduzindo a estabilidade relativa do sistema. Por isso, a ação integral geralmente não é utilizada de maneira isolada.

Por fim, o ganho derivativo (K_d) exerce grande influência quando o erro apresenta uma variação rápida. Seu aumento pode causar uma diminuição no tempo de subida e no sobressinal. Entretanto, a influência do K_d no erro em regime permanente é quase inexistente quando o sistema está estabilizado, uma vez que a variação do sinal de erro ($\Delta e(t)/\Delta$) é próxima de zero e, portanto, a contribuição da parcela derivativa é muito pequena.

2.3 Programação Orientada a Objetos (POO)

A programação Orientada a Objetos é um dos paradigmas da computação baseado no conceito de modelo de objetos. O seu objetivo é representar as situações do mundo real na forma de objetos que interagem entre si. Para isso, esse modelo engloba conceitos como: abstração, encapsulamento e classificação (FARINELLI, 2007).

Em POO, um objeto é a representação de um elemento do mundo real. Com isso, todo objeto possui: um conjunto de características próprias que o descrevem (atributos) e ações que ele pode realizar (métodos). Um gato, por exemplo, pode ser representado como um objeto. Seus atributos seriam: nome, idade, cor dos pelos, cor dos olhos e peso. Já seus métodos seriam: correr, pular, comer, etc. Percebe-se então, que todo gato possui os mesmos métodos, mas com atributos diferentes. Tendo isso em mente, vários objetos podem apresentar propriedades semelhantes. Sendo assim, tem-se o conceito de classe, que representa o conjunto de características e comportamentos comuns de um objeto. No exemplo citado acima, pode-se considerar que todos os gatos possuem a mesma classe, mas com valores de atributos diferentes.

Dessa forma, o objeto é uma instância de uma classe, o qual apresenta valores de atributos próprios, como se pode observar na Figura 2.

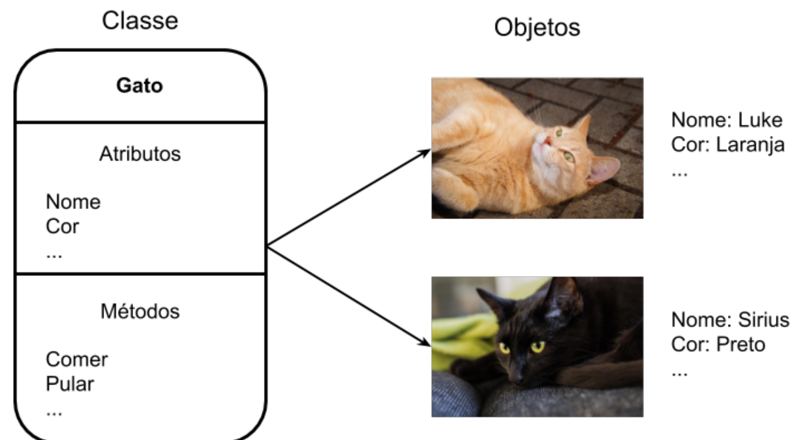


Figura 2 – Exemplo de objetos

Os atributos e métodos de uma classe podem ter vários estados, dependendo da linguagem de programação utilizada. Os dois principais estados, os quais são utilizados neste trabalho são:

- *public*: os atributos e métodos da classe podem ser acessados fora dessa;
- *private*: os atributos e métodos da classe não podem ser acessados fora dessa;

Quando um atributo é *private*, utiliza-se funções *get* e *set* para acessar e alterar o seu valor, respectivamente. A primeira é uma função que retorna o valor do atributo a ela associado. A segunda atribui o valor de sua variável passada por parâmetro ao atributo que se deseja alterar dentro da classe.

2.4 Busca por Vizinhança Variável para a Calibração de um PID (VNS-PID)

Nesta seção, é apresentada a abordagem proposta, chamada VNS-PID, que aplica o VNS ao problema de ajuste dos ganhos PID. O principal objetivo do VNS-PID é sintonizar os parâmetros do controlador PID do birrotor.

O VNS é uma meta-heurística que se baseia em mudanças sistemáticas na vizinhança (MLADENOVIĆ; HANSEN, 1997). Diferentes vizinhanças são exploradas assumindo que um ótimo local em uma delas não é, necessariamente, um ótimo local em outra.

O Algoritmo 1 apresenta uma implementação básica de um VNS para minimização, em que s é uma solução viável para o problema, k é o número de buscas locais sem melhorias, $f(.)$ é uma função que atribui uma pontuação a uma solução candidata e m é o número máximo de buscas locais sem melhoria. A função $generateSolution$ é responsável por gerar uma solução válida para o problema do PID. A função $randomNeighbor$ gera um vizinho válido de uma determinada solução. Por fim, a função $localSearch$ pesquisa a vizinhança de uma solução.

Algorithm 1: Algoritmo de um VNS básico.

1: **Procedure:**
2: $s \leftarrow generateSolution()$
3: $k \leftarrow 1$
4: **while** $k \leq m$ **do**
5: $s' \leftarrow randomNeighbor(s)$
6: $s^* \leftarrow localSearch(s')$
7: **if** $f(s^*) < f(s)$ **then**
8: $s \leftarrow s^*$
9: $k \leftarrow 1$
10: **else**
11: $k \leftarrow k + 1$
12: **return**: s

O VNS-PID concentra-se em maximizar a pontuação atribuída a uma solução. Essa pontuação é dada por meio da soma dos valores retornados por uma função de aptidão. A solução consiste em uma combinação dos valores das três constantes PID: K_p, K_i e K_d.

A pontuação de uma solução, em um determinado momento t, é calculada conforme apresentado na Equação 2.2 e a pontuação final é dada pela Equação 2.3:

$$g(t) = \left| \frac{x(t) - y(t)}{A - x(t)} \right| \tag{2.2}$$

em que, A é o ângulo máximo que o birrotor pode alcançar.

$$f(s) = 100 - \frac{\sum_{t=0}^{T} g(t)}{T} \tag{2.3}$$

em que, $g(t)$ é a pontuação no momento t e T é o tempo de duração do experimento.

O intervalo da função de pontuação final é de zero a 100. A pontuação máxima indica que o birrotor está estabilizado desde o início, enquanto a pontuação mínima indica que o birrotor não saiu de sua posição de origem. Neste trabalho, o cálculo da função de pontuação final é iniciado desde o início da avaliação da solução, portanto, a solução candidata é penalizada durante o período em que o sistema está iniciando. Essa é uma maneira de permitir que o sistema continue sempre evoluindo. Com isso, é impossível atingir a pontuação máxima, tendo em vista que o sistema inicia com um dos motores na posição inferior e precisa de um período para atingir a referência, que é de 180 graus. Decidiu-se começar nessa posição para garantir que todas as soluções comecem nas mesmas condições.

O Algoritmo 2 apresenta o pseudo-algoritmo VNS-PID, que se assemelha a uma implementação básica do VNS. No entanto, o processo de busca local utiliza o conceito de *first improvement*, ou seja, quando é encontrada uma solução melhor que a atual: (i) o algoritmo interrompe a busca local atual, (ii) o valor de k no Algoritmo 2 é definido como 1, (iii) o intervalo da busca local é dividido por dois e (iv) a próxima iteração é iniciada. Decidiu-se por essa

abordagem devido ao alto custo relacionado à avaliação de uma única solução e ao estreitamento da vizinhança avaliada, o que a aproxima de um máximo local. Além disso, adicionamos a variável i, a qual é responsável por restringir o número máximo de iterações. Dessa forma, os critérios de parada para a abordagem proposta são o número máximo de pesquisas locais sem aprimoramento da solução (k) e o número máximo de pesquisas locais a serem executadas (i). Se qualquer um dos dois critérios for atendido, o algoritmo encerra sua execução.

A estrutura de vizinhança usada neste trabalho consiste na adição de um valor aleatório diferente (negativo ou positivo) para cada constante PID. Uma nova solução é considerada uma solução viável se todas as constantes respeitarem as restrições. Se uma restrição for violada, a constante é arredondada para o valor mais próximo no intervalo (máximo ou mínimo).

Algorithm 2: Algoritmo VNS-PID.

```
1:  Procedure: Neighborhoods N^1, ..., N^m.
2:  s ← generateSolution()
3:  k ← 1
4:  range ← [−0.4 0.4]
5:  while k ≤ m do
6:      s' ← randomNeighbor(s, N^k, range)
7:      s* ← localSearch(s', range' = range/2)
8:      if f(s*) > f(s) then
9:          s ← s*
10:         k ← 1
11:         range ← range'
12:     else
13:         k ← k + 1
14:         range ← [−0.4 0.4]
15: return: s
```

2.5 Algoritmos Evolutivos

Algoritmos evolutivos são heurísticas de otimização baseados em comportamentos observados na natureza, como o processo de evolução e seleção natural, para o caso do GA, e o comportamento social e cooperativo observado no voo das aves, para o caso do PSO. Essas heurísticas são ditas evolutivas porque otimizam de forma iterativa. Diante disso, um grupo de soluções aleatórias é criado e é alterado, durantes as iterações, de acordo com os operadores evolutivos de cada algoritmo. As soluções com melhores desempenho nos critérios de seleção são disseminadas pelas gerações, e outras que não se encaixam desaparecem no decorrer das gerações. Dessa maneira, o conjunto de soluções converge para as que melhor atendem ao critério escolhido.

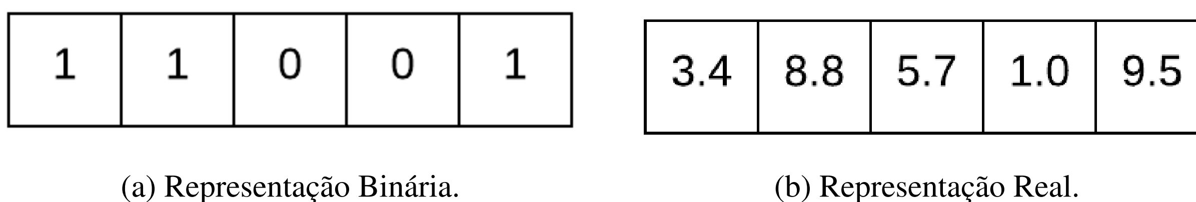
Como dito anteriormente, os algoritmos evolutivos usados neste trabalho são: GA e PSO.

Para ambos os algoritmos, GA e PSO, a representação de uma solução e a função de aptidão são etapas críticas para o seu processo de evolução, tendo em vista que uma solução

representada de forma equivocada (presença de características desnecessárias para o sistema e limites de valores errôneos) aliado a uma função que calcula a aptidão, de uma forma ineficiente, podem fazer com que o algoritmo não ache soluções ótimas para o problema. Neste trabalho, os parâmetros de uma dada solução é formado pelo conjunto de constantes PID: K_p, K_i, K_d e sua aptidão. No caso deste trabalho, a função de aptidão está relacionada ao período em que o birrotor permanece fora de sua referência. A função de aptidão é a mesma do VNS apresentada na Seção 2.4.

2.5.1 Algoritmo Genético para a calibração de um PID (GA-PID)

O Algoritmo Genético (WHITLEY, 1994; DAVIS, 1991) foi desenvolvido com base na teoria de seleção natural de Darwin. Portanto, ele simula a sobrevivência e reprodução de indivíduos de uma população em um ambiente, onde os mais aptos tendem a propagar suas características para as gerações futuras mais frequentemente. Para o contexto de um GA, um indivíduo representa uma solução possível para o problema, a população representa um conjunto de indivíduos e o ambiente representa o problema a ser otimizado. Assim como no contexto real, cada indivíduo possui genes que definem suas características e a forma como se comportam no ambiente; esses genes constituem a resposta para o problema sob análise. Sua representação pode ser binária (Figura 3 a) ou por números reais (Figura 3 b). A utilização de uma delas depende do tipo de problema; por exemplo, para problemas de roteamento como o cálculo da rota mais rápida a ser seguida para se deslocar de uma cidade A até uma cidade B, é recomendável que a representação seja binária, tendo em vista que é necessário guardar as cidades que farão parte da rota. Nesse contexto, cada posição representa uma cidade, em que posições com o valor 0 não fazem parte da rota e posições com o valor 1 fazem parte. Já a representação por números reais é mais adequada para a solução de equações e para problemas como o deste trabalho: otimizar constantes de um controlador, em que cada posição representa o valor de uma constante.



(a) Representação Binária.

(b) Representação Real.

Figura 3 – Representações de um gene.

Como dito anteriormente, os genes também caracterizam a forma como o indivíduo se comportará no ambiente em que está inserido. Para saber o quão bom é este comportamento faz-se uma avaliação, a qual é feita com base em uma função que descreve o ambiente (função objetivo). O valor gerado pela aplicação dos genes na função objetivo é chamado de aptidão. Esse valor é utilizado para classificar a população. Para o caso deste trabalho, a função objetivo é dada pela Equação 2.3.

O cálculo da aptidão é um dos processos mais importantes de um GA, pois permite a execução das demais etapas desse algoritmo e a aplicação dos mecanismos genéticos como: (1) seleção, (2) cruzamento, (3) mutação e (4) elitismo.

A seleção é o mecanismo responsável por selecionar os indivíduos que farão o cruzamento e gerarão um filho. Existem vários tipos de seleção, que variam de acordo com a aplicação e necessidades do problema. As duas principais formas de seleção são:

- Torneio: consiste na seleção do melhor indivíduo (o que tiver maior *aptidão*) entre dois indivíduos diferentes, escolhidos de forma aleatória. Essa abordagem aumenta a probabilidade de se propagar as melhores características para as gerações futuras;
- Evento aleatório: consiste na seleção aleatória de um indivíduo. Essa abordagem torna o algoritmo mais aleatório e permite uma busca mais extensa do espaço de busca. Por outro lado, a velocidade como o algoritmo converge diminui.

O cruzamento é o mecanismo genético responsável por gerar um novo indivíduo a partir de dois outros provenientes da seleção. O novo indivíduo terá seu material genético derivado dos pais selecionados. A forma como ocorre essa mistura é de acordo com a representação dos indivíduos. Para o contexto deste trabalho, em que as características do indivíduo são números reais, o cruzamento (S_g) ocorre de acordo com a Equação 2.4:

$$S_g = \alpha S_1 + (1 - \alpha) S_2 \tag{2.4}$$

em que S_1 é uma das constantes PID do primeiro indivíduo selecionado, α é um número aleatório entre 0,0 e 1,0 e S_2 é uma das constantes PID do outro indivíduo.

A Figura 4 ilustra a forma como é feito o cruzamento neste trabalho.

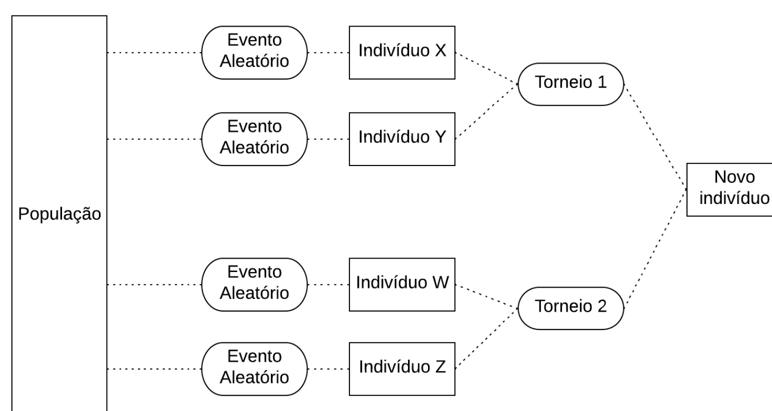


Figura 4 – Cruzamento.

Após o cruzamento, pode-se efetuar a mutação no indivíduo gerado. A mutação é um mecanismo de variabilidade genética, que permite alterar uma característica, específica, de

um indivíduo de forma aleatória e ocasional. Esse mecanismo evita que o algoritmo fique estacionado em um máximo local, visto que é possível explorar outras regiões do espaço de busca. Assim como ocorre com o cruzamento, a forma como ocorre a mutação varia de acordo com a representação do indivíduo. Para o contexto deste trabalho, o operador de mutação pode ser definido pela Equação 2.5:

$$S_m = S \times \delta \tag{2.5}$$

em que S é uma das constantes PID do indivíduo selecionado e δ é um número aleatório entre 0,0 e 1,0.

Após a mutação, faz-se o elitismo. Esse mecanismo é muito importante para a preservação das melhores soluções encontradas, visto que ele seleciona os melhores pais para compor a próxima população. A porcentagem de pais varia de acordo com cada problema. Ressalta-se que um valor elevado, pode comprometer a busca do GA, haja visto que a probabilidade da população ficar presa em um máximo local é alta. Por outro lado, um valor baixo pode retardar a convergência da população e a perda de indivíduos promissores. Neste trabalho, o elitismo consiste em juntar todos os pais e filhos e selecionar os melhores dentre eles. A Figura 5 ilustra os passos de um GA genérico descritos anteriormente.

O Algorítimo 3 apresenta um pseudocódigo do GA utilizado neste trabalho. O algoritmo gera um número pré-definido de indivíduos, por meio da função $generatePopulation()$ e então os avalia por meio da função $evaluateIndividual()$. Se a probabilidade for menor que a definida para cruzamento, é feito o torneio duas vezes, a fim de selecionar dois pais. O torneio é feito pela função $tournament()$, que pega dois indivíduos de forma aleatória e retorna o melhor entre eles. O torneio é feito até que os pais sejam indivíduos diferentes. Feito isso, a função $crossover$ gera um novo indivíduo originado da mistura genética dos pais. Se a probabilidade for menor que a probabilidade de mutação, o filho sofre uma mutação por meio da função $mutation()$. Essa função adiciona de forma aleatória um valor entre 0 e 1 em cada uma das características genéticas do filho (constantes PID). Após a etapa de cruzamento, o filho é avaliado. Com isso, ao final da geração, tem-se o dobro da população inicial. Porém, apenas metade dessa população fará parte da população inicial da próxima geração, a qual consistirá nos melhores indivíduos. Esse processo é chamado de elitismo e é feito pela função $elitism()$. Após a conclusão de todas as gerações, o algoritmo retorna o melhor indivíduo.

2.5.2 Otimização por Enxame de Partículas para a calibração de um PID (PSO-PID)

O Algoritmo de Otimização por Enxame de Partículas (SHI et al., 2001; SHI; EBERHART, 1999) foi desenvolvido inspirado na dinâmica comportamental e social de um bando de pássaros à procura de um alvo, o qual pode ser um local para pouso, alimento ou proteção contra predadores, por exemplo. Nesse contexto, um pássaro representa uma solução e é chamado de partícula e um

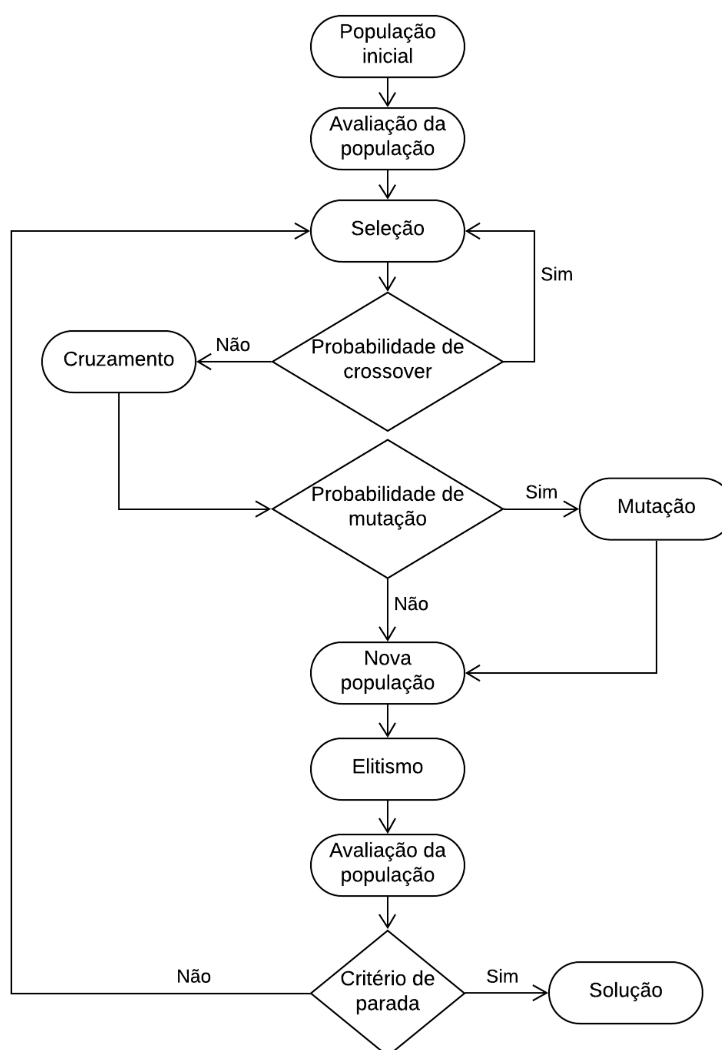


Figura 5 – Fluxograma de um Algoritmo Genético genérico.

bando representa um conjunto de soluções, chamado de enxame. O objetivo do PSO é encontrar uma solução ótima em um espaço de busca por meio do compartilhamento de informações entre as partículas de um enxame determinando qual trajetória que cada partícula deverá seguir no espaço de busca.

A forma como é feito o compartilhamento de informações entre o enxame é chamada de topologia. Há dois tipos de topologia:

- Topologia Global: As partículas possuem informações das partículas de todo o enxame. Essa topologia está representada na Figura 6 a.
- Topologia Local: As partículas possuem informações apenas de suas vizinhas. Essa topologia está representada na Figura 6 b.

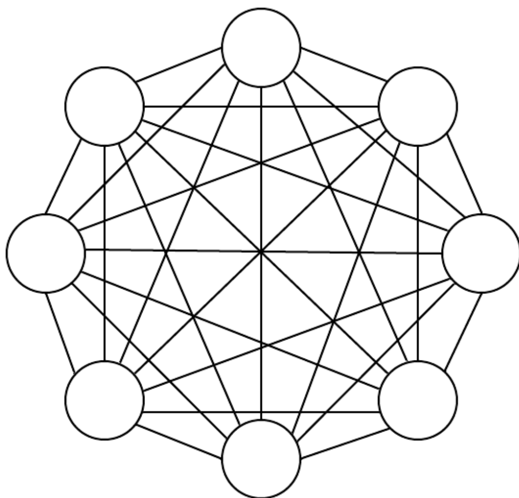
A topologia a ser escolhida depende muito do tipo de problema e cada uma possui seus

Algorithm 3: Algoritmo GA-PID

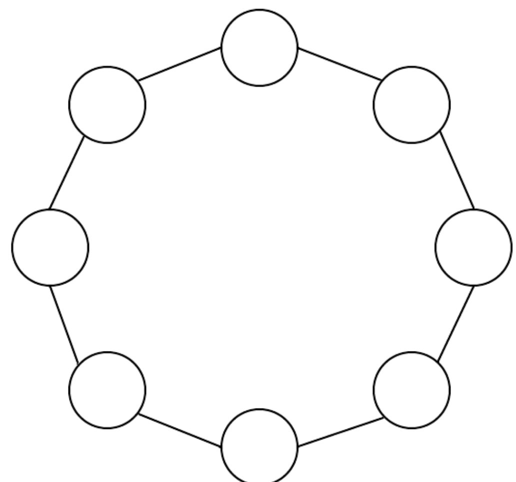
-

```
1:  s[] ← generatePopulation();
2:  evaluateIndividual();
3:  for i = 0; i < M; i++ do
4:      while j < N do
5:          probability ← random(0, 1);
6:          if probability < 0.95 then
7:              j++;
8:              s' ← tournament(s[]);
9:              while s' != s'' do
10:                 s'' ← tournament(s[]);
11:             s* ← crossover(s', s'');
12:             probability ← random(0, 1);
13:             if probability < 0.05 then
14:                 s* ← mutation(s*);
15:             evaluateIndividual();
16:     s[] ← elitism();
17: return: s[0]
```

prós e contras. Se por um lado a topologia global torna a convergência mais rápida, por outro, ela facilita a permanência das soluções em um máximo local, visto que todas as partículas seguirão a melhor posição global do enxame. Sendo assim, pode acontecer desse *gBest* estar preso e prender todo o enxame nesse mesmo local. Já a topologia local fornece o benefício de diversificar as partículas, permitindo uma exploração maior do espaço de busca. Porém, sua convergência é muito mais lenta.



(a) Topologia Global.



(b) Topologia Local.

Figura 6 – Representações de um gene.

Portanto, ao contrário do GA, o ambiente do PSO é cooperativo, ao invés de competitivo. Cada partícula tem uma posição e se move em uma velocidade particular, que se baseia não só em

sua própria experiência (termo cognitivo), mas também na experiência do enxame (termo social). Além disso, há um peso inercial (termo inercial) que influencia todo o enxame. O termo cognitivo é chamado *pBest* e representa a melhor posição que a partícula atingiu. O termo social, por outro lado, é chamado de *gBest* e representa a melhor posição do enxame atingiu, caso a topologia seja global; caso contrário, o *gBest* vai ser a melhor posição que um dos vizinhos da partícula alcançou. Por fim, o termo inercial é uma constante que equilibra a exploração global e local. Para valores altos, facilita-se a exploração global enquanto que para valores baixos facilita-se a exploração local (COSTA; BIAZI; VÍTOR, 2010). A Figura 7 apresenta o comportamento das partículas de um enxame em relação a cada um de seus termos: social, inercial e cognitivo.

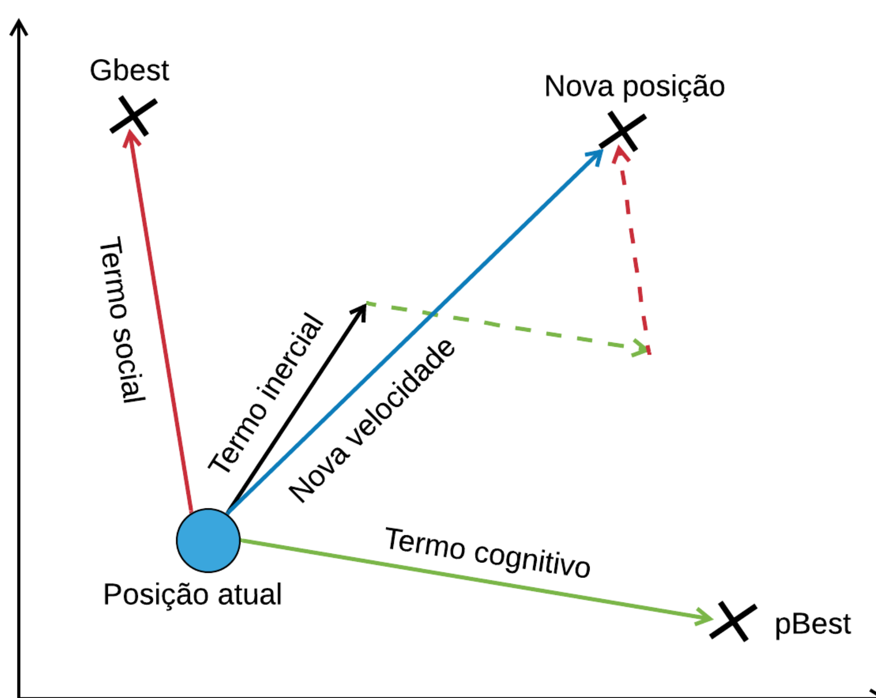


Figura 7 – Comportamento das partículas de um enxame.

Tendo isso em mente, a velocidade de uma partícula é influenciada por três principais fatores:

1. Velocidade anterior;
2. Distância entre sua posição atual e seu *pBest*;
3. Distância entre sua posição atual e o *gBest*.

A velocidade de uma partícula pode ser calculada conforme apresentado na Equação 2.6.

$$v_{ij}^{k+1} = w \cdot v_{ij}^{k} + c_1 \cdot r_{1j} \cdot (p_{ij}^{k} - x_{ij}^{k}) + c_2 \cdot r_{2j} \cdot (g_{j}^{k} - x_{ij}^{k})$$

em que v_{ij}^{k+1} é a velocidade da partícula i na geração $k+1$ da constante PID j. w é o peso inercial da partícula, v_{ij}^{k} é a velocidade da partícula anterior, c_1 e c_2 são constantes pré estabelecidas, r_{1j} e r_{2j} são valores aleatórios entre 0 e 1, p_{ij}^{k} é o *pbest* da partícula, g_j^k é o *gbest* do enxame e x_{ij}^{k} é a posição da partícula i.

A partir do cálculo da velocidade da partícula, calcula-se então sua nova posição. A Equação 2.6 apresenta a maneira como a posição da nova partícula é atualizada com base na velocidade e na posição atual da partícula:

$$x_{ij}^{k+1} = x_{ij} + v_{ij}^{k+1} \tag{2.6}$$

em que x_{ij}^{k+1} é a posição influenciada pelo contexto social do enxame, x_{ij} é a posição anterior da partícula e v_{ij}^{k+1} é a velocidade.

A Figura 8 ilustra a sequência de etapas de um PSO genérico, em que:

1. Inicia-se o enxame com posições aleatórias;
2. Avalia-se a aptidão das partículas;
3. Compara-se a aptidão obtida com o *pBest* da partícula. Se ela for maior, atualiza-se o *pBest* com o novo valor;
4. Compara-se o *gBest* atual com o anterior. Se for maior, atualiza-o com o novo valor;
5. Atualiza-se as velocidades das partículas, utilizando-se a Equação 2.6;
6. Atualiza-se as posições das partículas, utilizando-se a Equação 2.6;
7. Avalia-se a aptidão das partículas novamente;
8. Verifica-se o critério de parada. Caso este tenha sido satisfeito, a solução é o *gBest* encontrado. Caso contrário, repete-se os passos anteriores.

O Algorítimo 4 apresenta um pseudocódigo do PSO utilizado neste trabalho. Inicialmente, são geradas um número pré-determinado de partículas por meio da função *initializeParticles*() e então cada uma delas é avaliada pela função *evaluateParticle*(). Feito isso, o algoritmo inicializa a primeira geração encontrando o índice da partícula com a melhor posição atingida pelo enxame até o momento, por meio da função *gBest*(). Após isso, cada partícula guarda a sua melhor posição já atingida pela função *pBest*(). Em seguida, calcula-se as velocidades da partícula em cada uma das direções (K_p, K_i e K_d) e atualiza-se sua posição somando-se as velocidades com a posição que a partícula estava. Por fim, avalia-se a nova posição da partícula. Esse processo é feito para cada partícula ao longo de 15 gerações. Após todo esse processo, o algoritmo retorna a melhor partícula.

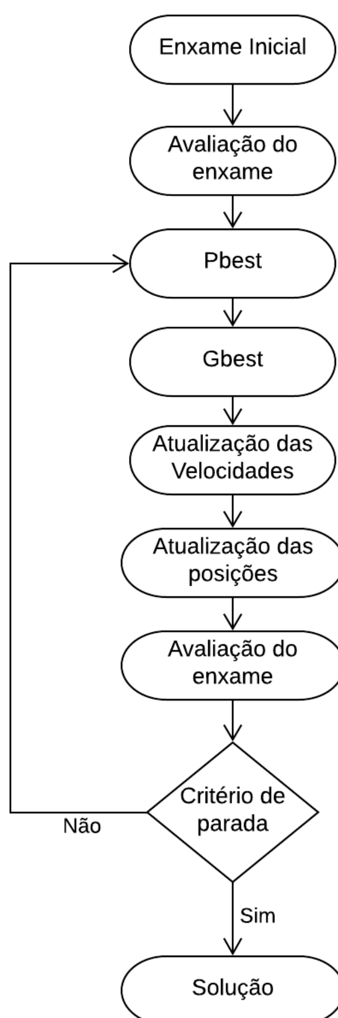


Figura 8 – Fluxograma de um Algoritmo de Otimização por Enxame de Partículas.

Algorithm 4: Algoritmo PSO-PID

-

1: $s[] \leftarrow initializeSwarm()$;
2: **for** i = 0; i < N; i++ **do**
3: $evaluateParticle(s[i])$;
4: **for** i = 0; i < M; i++ **do**
5: $gBestIndex \leftarrow gBest()$;
6: **for** j in $s[]$ **do**
7: $s[j] \leftarrow pBest()$;
8: $velocity[] \leftarrow getVelocity()$;
9: $s[i] \leftarrow s[i] + velocity[]$);
10: $evaluateParticle()$;
11: **return**: $s[gBestIndex]$

2.6 Filtro de Kalman

O filtro de Kalman é um método estatístico elaborado por Rudolf Kalman em 1960 Welch, Bishop et al. (1995). Ele consiste em um filtro recursivo que opera em sistemas dinâmicos lineares e os estima por meio de suas medições, que em grande parte dos casos, serão ruidosas.

A natureza recursiva do filtro de Kalman permite que ele possa ser executado em tempo real. A recursão é uma das grandes vantagens desse filtro em relação aos outros, uma vez que o algoritmo prevê o novo estado no tempo t+1, por meio da adição de um termo de correção. Esse novo estado corrigido passa a ser a condição inicial do próximo estágio. Com isso, as próximas variáveis calculadas utilizarão todas as informações disponíveis até o momento.

No que se refere ao algoritmo do filtro de Kalman, esse requer dois tipos de equações: equações principais, que relacionam as variáveis de estado com as variáveis observáveis e equações de estado, que determinam a estrutura temporal das variáveis de estado.

A dinâmica desse filtro ocorre da seguinte maneira: primeiro, a etapa de previsão, onde estima-se as variáveis de estado do sistema utilizando sua dinâmica e, segundo, a etapa de correção, em que melhora-se a estimativa da etapa anterior, utilizando-se os dados das variáveis observáveis.

As equações e metodologia de implementação do filtro de Kalman utilizados neste trabalho estão presentes em Lauszus (2012).

3 DESENVOLVIMENTO

Um sistema instável é uma planta física na qual a posição esperada é diferente da posição natural (BINGI et al., 2017). Ele precisa ser estabilizado para atingir a condição esperada.

A posição natural de um birrotor apresenta um motor suspenso e outro encostado no chão. Em contrapartida, a posição esperada se dá quando os dois motores estão alinhados e o sistema permanece paralelo ao solo. Outros exemplos de sistemas instáveis são: pêndulo invertido, quadrotores ou qualquer variação com a mesma premissa.

A estabilização da planta requer um sensor para orientar o processo, como um giroscópio, para o caso específico de um birrotor. O giroscópio fornece informações que permitem ao microcontrolador medir o ângulo de inclinação do sistema em relação ao solo. Um mecanismo de repetição do controle da planta utiliza o ângulo calculado pelo microcontrolador, por meio do giroscópio, para calcular um erro e, a partir deste, gerar uma resposta condizente com as constantes PID do sistema. Essa resposta visa corrigir o comportamento dos atuadores para que a planta retorne à posição desejada.

Existem várias técnicas de controle para sistemas instáveis presentes na literatura, dentre elas destaca-se o controle de estabilidade de Lyapunov (TAHRI et al., 2016) e o controle PID (ÅSTRÖM; HÄGGLUND; ASTROM, 2006; FRANKLIN et al., 1994). No entanto, o mais popular e amplamente utilizado é o controlador PID, aplicado neste trabalho.

Neste trabalho, uma estrutura suspensa com dois motores sem escova, que funcionam como atuadores, são utilizados nos experimentos. Os motores sem escova são colocados nas bordas junto com as hélices, com o objetivo de empurrar o ar para baixo e elevar o eixo. A Figura 9 mostra um esquema do sistema birrotor guiado por um controlador PID. $x(t)$ é a referência em graus, que neste trabalho é 180° (posição horizontal) e $y(t)$ é o ângulo atual do birrotor. A diferença entre $x(t)$ e $y(t)$ fornece o erro $e(t)$. Este último é enviado ao controlador PID, que transforma essa entrada em um ganho que aumenta ou diminui a modulação por largura de pulso necessária para definir a velocidade dos motores. O resultado desse ganho altera ou mantém a inclinação do sistema $y(t)$.

3.1 Controlador PID para Sistemas Reais

Um grande desafio na aplicação do PID em sistemas reais é a calibração das constantes. A combinação de valores para cada constante é um grande espaço de pesquisa e é dispendioso encontrar uma solução adequada manualmente.

Uma alternativa a essa calibração manual é a implementação de técnicas de otimização utilizadas para automatizar o processo de calibração dos ganhos PID. Essas técnicas promovem

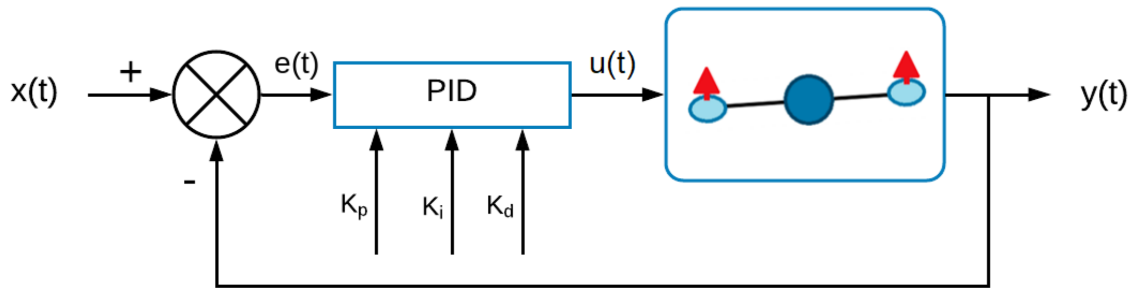


Figura 9 – PID aplicado para estabilizar o birrotor.

um ajuste de forma mais inteligente e rápida do que as soluções convencionais. Tendo isso em mente, vários trabalhos na literatura têm mostrado sucesso nessa forma de calibração, como indicam os estudos que utilizam GA e PSO (KROHLING; REY, 2001; ZHANG et al., 2009; GAING, 2004; AHMADI; ABDI; KAKAVAND, 2017). Nas próximas seções, será apresentado as abordagens desenvolvidas, baseadas no VNS, GA e PSO, as quais são aplicadas a um contexto *hardware-in-the-loop*.

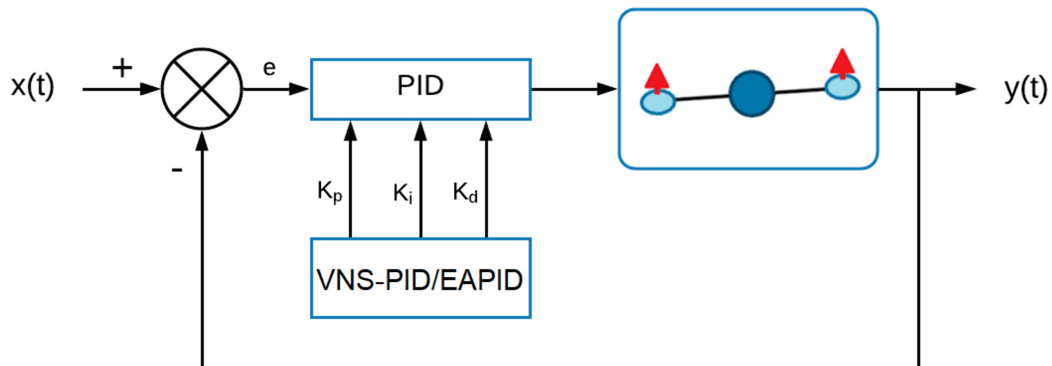


Figura 10 – Forma geral do funcionamento do sistema.

3.2 Hardware do birrotor

O sistema do birrotor usado neste trabalho consiste em uma estrutura de madeira maciça, em formato de um "U"usada como base do sistema, mantendo-o no chão enquanto os experimentos são executados. Por meio da Figura 11, observa-se as dimensões dessa base e as dimensões de sua parte lateral e inferior, respectivamente. A fixação dessas é feita por meio de parafusos cabeça chata 3,5x40 mm, como os da Figura 12.

Para prender o birrotor na base, utiliza-se uma barra de metal cilíndrica de rosca infinita de 34 cm, como a da Figura 13, que passa através das extremidades da estrutura de madeira e é fixada por porcas.

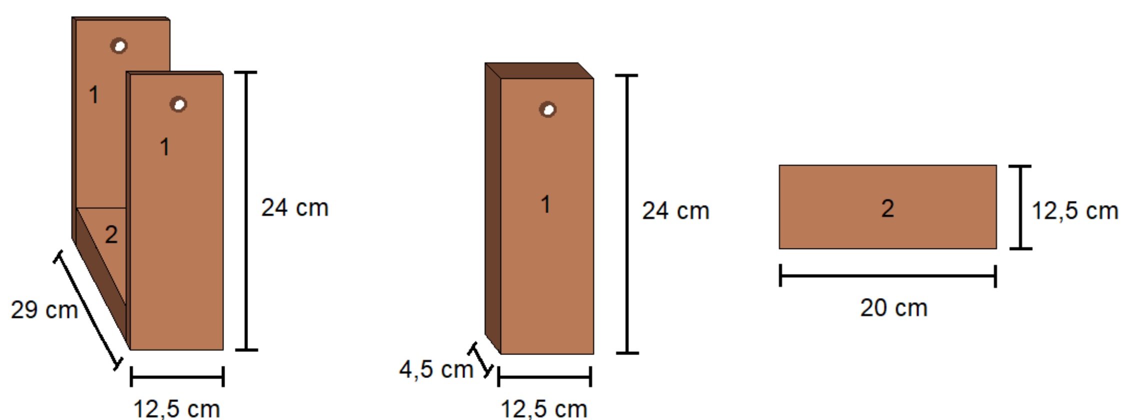


Figura 11 – Base do sistema.

Figura 12 – Parafuso utilizado para fixação da base.

Figura 13 – Barra cilíndrica utiliza para fixar o birrotor na base.

Com relação aos componentes que compõem a parte do birrotor, utilizou-se uma MPU-6050 como sensor para adquirir as informações necessárias para o microcontrolador calcular o ângulo desse sensor. A Figura 14 ilustra a MPU 6050 utilizada neste trabalho. Utiliza-se os pinos SCL e SDA desse sensor para efetuar uma comunicação I2C com o microcontrolador. O pino ADO fica desconectado, com isso o endereço I2C do sensor é 0x68. Caso esse estivesse conectado a uma tensão de 3,3 V, o endereço seria 0x69. Esse mecanismo é utilizado quando se deseja utilizar dois sensores em um mesmo circuito. Por fim, não se utiliza o pino INT, o qual é responsável por gerar interrupções.

Como dito anteriormente, é necessário um microcontrolador para calcular o ângulo da MPU 6050 com base nas informações que essa fornece. Nesse sentido, utilizou-se o ATmega 328p, que trabalha com a frequência de 16 MHz. A escolha desse microcontrolador foi com base na disponibilidade no laboratório. Além disso, o microcontrolador também é responsável por gerar o sinal de controle que será responsável por regular a velocidade de rotação dos motores.

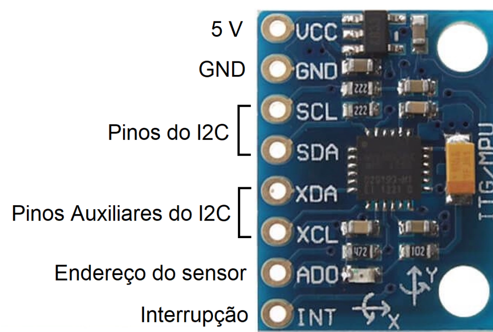


Figura 14 – MPU 6050 utilizada.

Esse sinal de controle não é enviado diretamente para os motores, mas sim para um controlador eletrônico de velocidade (em inglês, *Eletronic Speed Controller* ou ESC), que realiza a interface entre o microcontrolador e os motores. Esse sinal de controle, na verdade, regula a intensidade com que se aciona as bobinas dos motores, quanto mais intenso, mais rápida são as suas velocidades de giro. O ESC se comporta como um filtro de 5 V para acionar os motores com segurança. A Figura 15 ilustra o ESC utilizado neste trabalho. Vale ressaltar que o fio branco do conector preto é ligado a uma das portas PWM do microcontrolador, o fio vermelho e preto desse mesmo conector são ligados no 5 V e GND do microcontrolador, respectivamente. Os fios vermelho e preto do ESC são os seus fios de alimentação, sendo assim, eles são ligados em uma fonte ajustável, que é regulada em 12 V e possui capacidade de fornecer até 30 A. Por fim, seus fios azuis são ligados nos fios dos enrolamentos das bobinas do motor sem escova, são os ESCs que fornecem a sua alimentação.

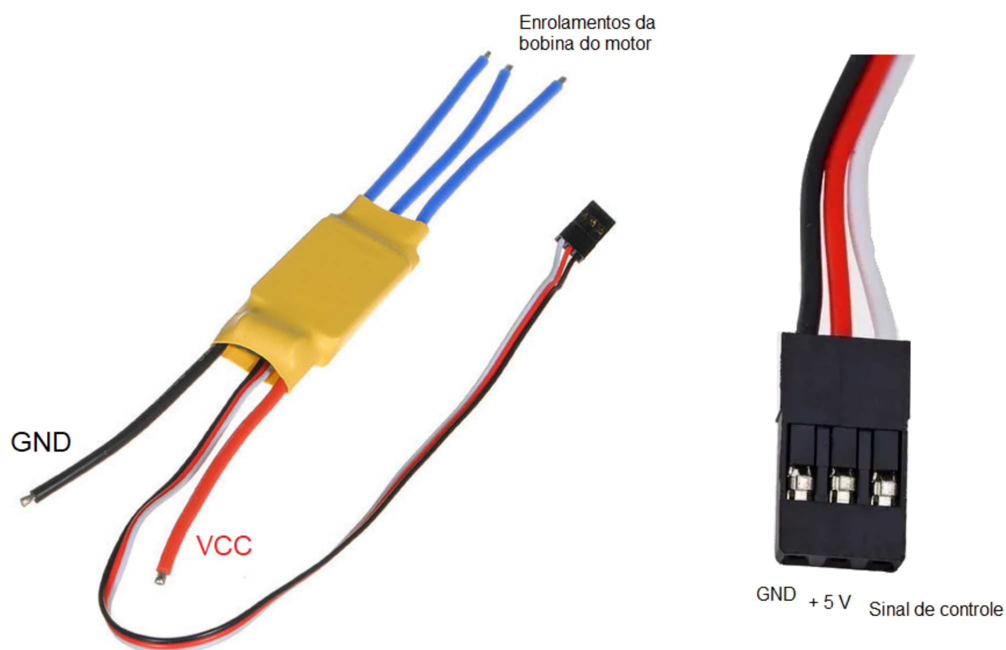


Figura 15 – Controlador eletrônico de velocidade utilizado para acionar os motores.

Com relação aos motores sem escova, utilizou-se dois motores de corrente contínua (CC) de 5 V e 1000 KV, conforme o da Figura 16. A sua escolha não envolveu nenhum aspecto teórico, mas sim de disponibilidade no laboratório.

Figura 16 – Motor sem escova utilizado.

O sentido de rotação dos motores é definido de acordo com a forma com que se liga os enrolamentos de suas bobinas e os fios do ESC que alimentam as mesmas. Com isso, quando se deseja alterar o sentido de rotação, a ligação dos fios azuis do ESC com os fios dos enrolamentos das bobinas do motor muda. A ligação dos motores da estrutura do birrotor foi feita como na Figura 17.

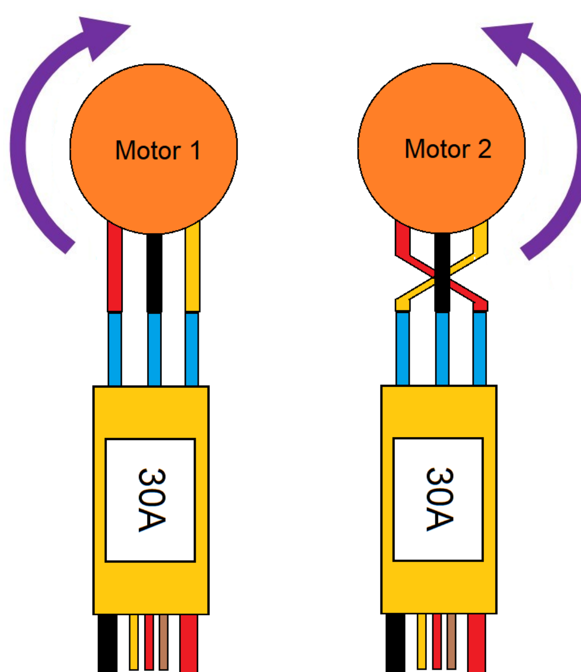


Figura 17 – Ligação dos fios de acionamento das bobinas dos motores para se alterar o sentido de rotação dos motores.

A presença somente dos motores não promoveria a instabilidade do sistema. Para isso,

são necessárias hélices fixadas neles. Quando os motores giram em sentidos diferentes de rotação (conforme pode-se observar na Figura 19), as hélices produzem um impulso ascendente que desequilibra todo o sistema devido ao seu torque. Como o sentido de rotação é diferente, as hélices precisam ter formatos diferentes para que elas empurrem o ar apropriadamente. Por isso, utiliza-se um par de hélices como o da Figura 18, em que uma hélice é própria para girar no sentido horário e outra é própria para girar no sentido anti-horário. As hélices utilizadas neste trabalho é um par de hélices 8045 de carbono, conforme a Figura 18.

Figura 18 – Conjunto de hélices utilizadas no motor.

A superfície tem uma grande importância para o ajuste das constantes PID, visto que a avaliação de todos os indivíduos é baseada no ângulo em que o sistema está relacionado à superfície. Por esse motivo, os testes são feitos no chão em um lugar plano.

Para todos os três algoritmos estudados, o ângulo mais alto alcançado pelo birrotor é de 220 graus e o *setpoint* é ajustado para 180 graus. A função de pontuação final para cada solução possível é a soma da pontuação obtida em 6000 ciclos. Este tempo é suficiente para permitir que o sistema se estabilize.

O esquema do protótipo é apresentado na Fig. 19 e a calibração do PID em execução no modelo real pode ser visto em:

<https://drive.google.com/file/d/13UhPe1TlZUdM0s9KPE74UIdYBMqTKHGB/view?usp=sharing>.

3.3 Código elaborado para sintonizar o controlador PID do birrotor

Para a elaboração do código para a sintonia do controlador PID do birrotor, utilizou-se o conceito de objetos para tornar o código não só mais modular, mas também mais simples de se entender conforme descrito na Seção 2.1.

O algoritmo possui as seguintes classes:

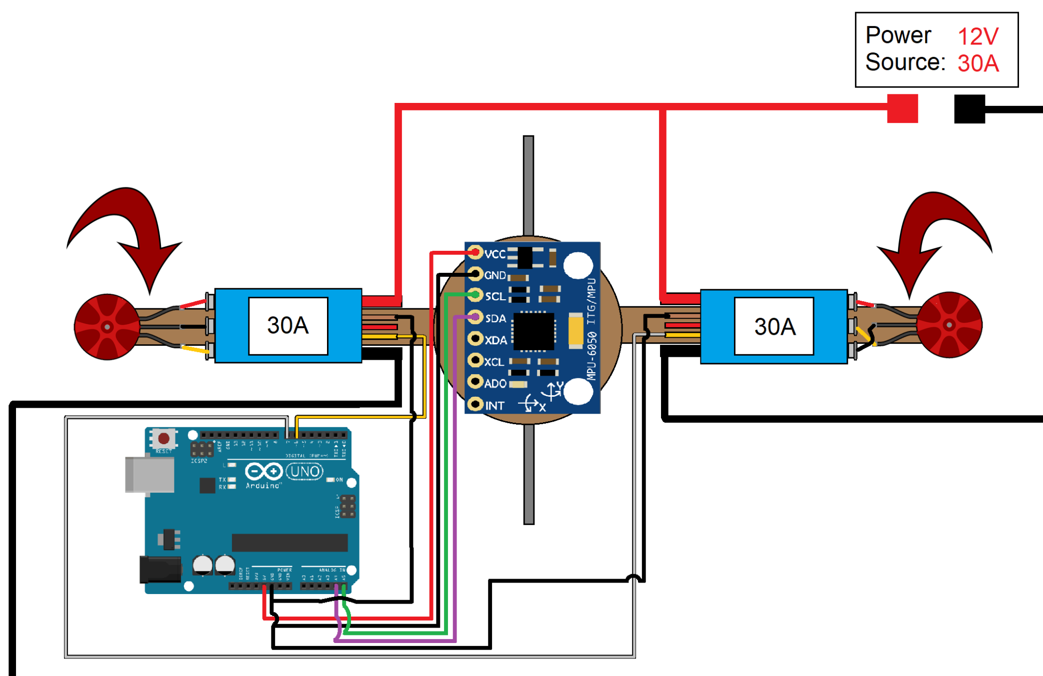


Figura 19 – Esquema do birrotor

GA:

Essa classe possui os métodos e atributos necessários para a execução do Algoritmo Genético. A Figura 20 apresenta o arquivo *header* da classe *GA*, o qual é um arquivo com as funções protótipo dos métodos, os atributos e as definições das constantes da classe.

Essa classe possui as seguintes bibliotecas, que no caso são as classes incluídas:

- *Individual*: Todos os atributos e métodos de um indivíduo do GA estão definidos nessa classe. É por meio dela que os objetos dos indivíduos serão criados e manipulados;
- *PID*: O controlador PID do birrotor está implementado nessa classe. Essa biblioteca é fundamental para adquirir o sinal de controle proveniente da aplicação das constantes PID (Indivíduo) no controlador PID;
- *Motor*: A interface entre o microcontrolador e os ESCs é feita nessa classe. Sendo assim, ela serve para enviar o sinal de controle até os ESCs;
- *Parameters*: Todas as definições de constantes estão declaradas nesse *header*.

Essa classe possui os seguintes atributos:

- *vetIndividual*: armazena os ponteiros de toda a população, tanto de pais quanto de filhos, por isso seu tamanho é o dobro do número de indivíduos. A constante *NUM-*

```
#ifndef GA_H
#define GA_H

#include "Individual.h"
#include "PID.h"
#include "Motor.h"
#include "Parameters.h"

class GA{
  public:
    GA();
    void deleteIndividual();
    void printIndividual(int index);
    void printScore(int index);
    void crossover(Individual *a, Individual *b, Individual *result);
    void mutate(Individual *a, Individual *m);
    float evaluate(Individual *a, Motor *m, PID *p, TiltSensor *s);
    Individual run( Motor *m, PID *p, TiltSensor *s);
    void evolution( Motor *m, PID *p, TiltSensor *s);
    float randomDouble(float minf, float maxf);
    int torneio(int minId, int maxId);
  private:
    Individual *vetIndividual[NUMBER_OF_INDIVIDUALS * 2];
    Individual *swap;
    float mutateProbability;
    float crossoverProbability;
};

#endif
```

Figura 20 – Header da classe GA.

BER_OF_INDIVIDUALS representa a quantidade de indivíduos e é declarada em *Parameters*;

- *swap*: variável auxiliar, que auxilia na ordenação dos ponteiros do atributo *vetIndividual* em ordem crescente, considerando a aptidão de cada indivíduo;
- *mutateProbability*: probabilidade de mutação
- *crossoverProbability*: probabilidade de cruzamento.

Os métodos da classe são indicados abaixo:

- *deleteIndividual*: exclui toda a população da memória do microcontrolador. Essa medida é necessária porque a população é um vetor normal, por isso, a alocação de memória tem que ser manual. Devido ao fato de que os algoritmos VNS-PID, GA-PID e PSO-PID são executados sequencialmente de forma automática, a não desalocação da memória do microcontrolador inviabilizaria a execução dos demais testes;
- *printIndividual* e *printScore* são responsáveis por exibir as informações para o usuário. O parâmetro de ambos os métodos são as variávei *index*, a qual é o índice do indivíduo que se deseja imprimir do atributo *vetIndividual*;

- *crossover*: realiza o cruzamento de dois indivíduos. Seus parâmetros são passados por referência, a fim de tornar o algoritmo mais rápido e com menor consumo de memória, uma vez que método não cria uma cópia da variável. Seus parâmetros são 3 objetos da classe *Individual*: os pais e o filho, respectivamente.
- *mutate*: realiza a mutação de um indivíduo. Seus parâmetros são: os ponteiros dos objetos do indivíduo que sofrerá a mutação e do indivíduo resultante da mutação, respectivamente;
- *evaluate*: calcula a aptidão do indivíduo sob análise. Para isso, ele recebe como parâmetros um ponteiro do indivíduo e os ponteiros dos objetos das classes *Motor*, *PID* e *TiltSensor*, respectivamente. É nesse método onde faz-se a aquisição do ângulo atual (por meio da classe *TiltSensor*), gera-se um sinal de controle à partir de um controlador PID (por meio da classe *PID*) e manda-o até os motores (por meio da classe *Motor*). O retorno dessa função é um valor *float*, que é a aptidão do indivíduo;
- *run*: controla as gerações do GA e avalia a primeira população. Seu retorno é um objeto da classe *Individual*, o qual é a solução final encontrada pelo GA-PID. Seus parâmetros são os objetos das classes: *Motor*, *PID* e *TiltSensor*. A finalidade desses parâmetros é chamar a função: *evaluate*, que necessita desses objetos;
- *evolution*: executa os métodos da classe na ordem correta, a fim de seguir o fluxo da metodologia do GA (Conforme explicado na Seção 2.5.1). É nesse método que o algoritmo genético será posto em prática. Seus parâmetros são os objetos das classes: *Motor*, *PID* e *TiltSensor*. A finalidade desses parâmetros é chamar a função: *evaluate*, que necessita desses objetos;
- *randomDouble*: gera um número aleatório dentro de um intervalo. Seus parâmetros são o valor mínimo e máximo, respectivamente, desse intervalo;
- *torneio*: seleciona, para o cruzamento, um indivíduo dentre dois outros escolhidos randomicamente. Seus atributos são o índice inferior e superior de indivíduos (zero e o número de indivíduos da população, respectivamente) para a escolha de dois indivíduos de forma randômica (Conforme discutido na Seção 2.5.1. Seu retorno é o índice do indivíduo vencedor do torneio.

I2C

Essa classe contém os atributos e métodos necessários para estabelecer a comunicação entre o giroscópio e o microcontrolador. A Figura 21 apresenta o *header* da classe *I2C*.

As bibliotecas utilizadas nessa classe são:

- *Arduíno*: inclui todos os métodos e constantes pré-definidos no ambiente de desenvolvimento do Arduíno;

```
#ifndef I2C_h
#define I2C_h

#include <Arduino.h>
#include <Wire.h>

extern const uint8_t IMUAddress;
extern const uint16_t I2C_TIMEOUT;

void i2cBegin();
uint8_t i2cWrite(uint8_t registerAddress, uint8_t data, bool sendStop);
uint8_t i2cWrite(uint8_t registerAddress, uint8_t *data, uint8_t length, bool sendStop);
uint8_t i2cRead(uint8_t registerAddress, uint8_t *data, uint8_t nbytes);

#endif
```

Figura 21 – Header da classe I2C.

- *Wire*: permite a comunicação i2c entre o microcontrolador e outros dispositivos.

Os atributos da classe são:

- *IMUAddress*: endereço do registrador da MPU 6050;
- *I2C_TIMEOUT*: tempo máximo de espera da comunicação.

Os métodos da classe *I2C* são:

- *i2cBegin*: inicializa a comunicação i2c;
- *i2cWrite*: desabilita o *sleep mode* da MPU. Seus parâmetros são: *registerAddress*, que é o endereço do registrador da MPU; *data*, que é a informação contida na MPU e *sendStop*, que é o número de bits de parada definidos na comunicação I2C. Seu retorno é um inteiro de oito bits que indica o êxito da comunicação;
- *i2cWrite*: esse método possui o mesmo nome que o anterior. No entanto, possui um número superior de parâmetros e finalidade diferente. Sua finalidade é o envio de informações para a MPU. Os parâmetros desse método são os mesmos que o anterior, porém o adicional, *length* é o tamanho da mensagem que o microcontrolador está enviando. Seu retorno é um inteiro de oito bits que indica o êxito da comunicação;
- *i2cRead*: lê os registradores da MPU. Seus parâmetros são: *registerAddress*, que é o endereço do registrador da MPU; *data*, que são os dados contidos na MPU e *nbytes*, que é o tamanho da mensagem proveniente da MPU. Seu retorno é um inteiro de oito bits que indica o êxito da comunicação;

Individual

Essa classe realiza a abstração do indivíduo do GA, que no caso são as constantes PID e sua respectiva aptidão. A Figura 22 apresenta o *header* da classe Individual.

```
#ifndef INDIVIDUAL_H
#define INDIVIDUAL_H

class Individual {
  public:
    Individual(float p, float i, float d);

    void setKp(float p);
    void setKd(float d);
    void setKi(float i);
    void setPID(float p, float i, float d);
    void setScore(float s);

    float const getScore();
    float const getKp();
    float const getKd();
    float const getKi();

  private:
    float kp;
    float ki;
    float kd;
    float score;
};

#endif
```

Figura 22 – Header da classe Individual.

Os atributos dessa classe são: K_p, K_i, K_d e a aptidão. Já os métodos são funções *get* e *set* para cada um dos atributos;

Motor

A classe *Motor* realiza a interface entre o microcontrolador e os motores. A Figura 23 apresenta o *header* da classe *Motor*.

```
#ifndef Motor_h
#define Motor_h

#include <Arduino.h>
#include <Servo.h>
#include "Parameters.h"

class Motor{
  public:
    Motor();
    void startMotor();
    void set(int pidGain);
    void initialPosition(int initTime);

  private:
    Servo motor1;
    Servo motor2;
};

#endif
```

Figura 23 – Header da classe Motor.

As bibliotecas necessárias para a execução dessa classe são: *Arduino*, *Parameters*, as quais foram explicadas anteriormente, e a *Servo*, a qual permite o controle dos motores.

Com relação aos atributos, a classe *Motor* possui dois: motor1 e motor2, os quais são objetos da classe *Servo*. Eles são necessários para utilizar um método da classe *Servo*, que permite enviar um sinal PWM.

No que diz respeito aos métodos dessa classe, eles estão sumarizados abaixo:

- *startMotor*: inicializa os motores e deixa-os prontos para receber o sinal de controle;
- *set*: envia o sinal de controle para os motores. Essa função possui apenas um parâmetro: *pidGain*, o qual é o sinal de controle gerado pela classe *PID*.
- *initialPosition*: deixa o birrotor na posição inicial para o início da avaliação de cada indivíduo. Seu parâmetro *initTime* é o tempo em que o birrotor ficará na posição inicial.

PID

Essa classe implementa o controle PID do birrotor. A Figura 24 apresenta o *header* da classe PID.

```
#ifndef PID_h
#define PID_h

#include <Arduino.h>
#include "Motor.h"
#include "TiltSensor.h"


class PID{
  public:
    PID();
    void setPID(float p, float i, float d);
    void setContEstable(int c);
    void robotBalance(Motor *m, TiltSensor *s, PID *p);
    int control(float val);
    float getContEstable();

  private:
    float kp;
    float ki;
    float kd;
    float contEstable;
    float lastError;
};

#endif
```

Figura 24 – Header da classe PID.

As bibliotecas utilizadas nesta classe são: *Arduino*, *Motor*, a qual é necessária para enviar o sinal de controle gerado pelo controlador PID aos motores e *TiltSensor*, a qual é necessária para a aquisição do ângulo atual do birrotor, permitindo o cálculo do sinal de controle.

Os atributos dessa classe são:

- K_p, K_i e K_d: constantes PID do indivíduo sob análise;
- *contEstable*: guarda o valor da aptidão do indivíduo sob análise;
- *lastError*: guarda o erro entre o ângulo da amostragem anterior e a referência.

Os métodos dessa classe são:

- *setPID*: atribui os valores das constantes PID do indivíduo a ser testado para os atributos K_p, K_i e K_d. Seus parâmetros são as constantes PID do indivíduo sob análise;
- *setContEstable* e *getContEstable*: são as funções *get* e *set* do atributo *contEstable*, respectivamente;
- *robotBalance*: calcula a aptidão a cada período de amostragem da MPU e envia o sinal de controle para os motores. Seus parâmetros são objetos das classes: *Motor*, *TiltSensor* que servem para chamar o método *set* da classe motor e pegar o ângulo calculado da MPU, respectivamente;
- *control*: calcula o sinal de controle por meio da aplicação do controle PID. Seu parâmetro é *angle*, que é o ângulo calculado da MPU. O seu retorno é o sinal de controle, o qual é uma variável do tipo inteiro.

PSO

Essa classe possui os métodos e atributos necessários para a execução do PSO. A Figura 25 apresenta o *header* da classe *PSO*.

As bibliotecas que a classe PSO necessita para a execução de seus métodos são: *Particle*, que é responsável pela abstração das partículas do PSO. Ela é importante para a classe *PSO* porque é por meio dela que se cria os objetos das partículas. As demais bibliotecas foram explicadas para a classe *GA* e possuem o mesmo propósito para esta.

Com relação aos métodos, *evaluate* e *randomDouble* possuem a mesma finalidade que os métodos de mesmo nome da classe *GA*. Os demais métodos da classe *PSO* são:

- O método *pBest* compara a aptidão atual com a melhor aptidão que a partícula atingiu, a fim de atualizar o *pBest* da partícula, caso a aptidão atual for maior. Essas informações de aptidões ficam guardadas na classe *Particle*;
- *getVelocity*: atualiza as velocidades das partículas em cada uma das direções: K_p, K_i, K_d. Seus parâmetros são: *gBest_index*, que é o índice da partícula que obteve a melhor aptidão; *a*, que é a partícula em que se calculará as novas velocidades (utiliza-se a passagem por referência para aumentar a velocidade do código e diminuir o espaço gasto pelo mesmo, conforme discutido anteriormente);

```
#ifndef PSO_H
#define PSO_H

#include "Particle.h"
#include "Parameters.h"
#include "PID.h"
#include "Motor.h"

#define min 0
#define max 10

class PSO{
  public:
    PSO();
    float evaluate(Particle *a,  Motor *m, PID *p, TiltSensor *s);
    float randomDouble(float minf, float maxf);
    void pBest();
    void getVelocity(int gBest_index, Particle *a);
    void updateParticles(int gBest_index, Motor *m, PID *p, TiltSensor *s, int cont);
    void printInTxt(Particle *solution2, String type, int iteration, int numberLocalSearch);
    void printInTxtGbest(Particle *solution2, String type, int iteration, int numberLocalSearch);
    void run( Motor *m, PID *p, TiltSensor *s);
    void deleteParticle();
    int gBest();
    int worstParticle();
    int randomInt();

  private:
    int gBest_index = 0;
    float velp = 0.0, veli = 0.0, veld = 0.0;
    float ponderacao;
    Particle *vetParticle[NUMBER_OF_PARTICLES*2];
    Particle *swap;
};

#endif
```

Figura 25 – Header da classe PSO.

- *updateParticles*: atualiza as posições das partículas. Seus parâmetros são: *gBest_index*, que é o índice da partícula que obteve a melhor aptidão; os objetos das classes: *Motor*, *PID* e *TiltSensor*; *cont*, que é a geração atual do algoritmo;

- *printInTxt* e *printInTxtGbest*: imprimem informações para o usuário;

- *run*: gerencia as gerações do PSO e ordena a execução dos métodos de forma a executar o PSO de forma correta. Seus parâmetros são os objetos das classes: *Motor*, *PID* e *TiltSensor*;

- *deleteParticle*: exclui da memória do microcontrolador todas as partículas, após a execução completa do PSO. Esse método é importante para liberar espaço para novos testes;

- *gBest*: encontra a partícula que obteve o melhor *pbest*. Ele retorna uma variável do tipo *int*, que é o índice da melhor partícula.

Particle

Essa classe realiza a abstração da partícula do PSO. A Figura 26 apresenta o *header* da classe *Particle*.

```
#ifndef Particle_H
#define Particle_H

class Particle {
  public:
    Particle(float p, float i, float d);

    void setCurrentKp(float p);
    void setCurrentKd(float d);
    void setCurrentKi(float i);
    void setPbestKp(float p);
    void setPbestKd(float d);
    void setPbestKi(float i);
    void setPID(float p, float i, float d);
    void setPbestPID(float p, float i, float d);
    void setPbestFitness(float pb);
    void setPreviousVelocityKp(float vel);
    void setPreviousVelocityKi(float vel);
    void setPreviousVelocityKd(float vel);
    void setCurrentFitness(float s);
    void setAllVelocity(float p, float i, float d);

    float getPbestFitness();
    float getCurrentKp();
    float getCurrentKd();
    float getCurrentKi();
    float getPbestKp();
    float getPbestKd();
    float getPbestKi();
    float getPreviousVelocityKp();
    float getPreviousVelocityKi();
    float getPreviousVelocityKd();
    float getCurrentFitness();

  private:
    float kp;
    float ki;
    float kd;
    float best_kp;
    float best_ki;
    float best_kd;
    float pBest_Fitness;
    float pVelp;
    float pVeli;
    float pVeld;
    float Fitness;
};
```

Figura 26 – Header da classe Particle.

Os atributos dessa classe são:

- k_p, k_i, $_d$: constantes PID;
- best_k_p, best_k_i, best_k_d: *pBest* da partícula;
- *Fitness*: aptidão da partícula;
- *pBest_Fitness*: aptidão do *pBest* da partícula;
- *pVelp*, *pVeli* e *pVeld*: velocidades da partícula nas direções da constante proporcional, integral e derivativa, respectivamente.

Os métodos dessa classe são *get* e *set* dos atributos.

TiltSensor

Essa classe possui os atributos e métodos para ler os registradores do giroscópio. A Figura 27 apresenta o *header* da classe *TiltSensor*.

As bibliotecas dessa classe são: *Arduino*, *I2C* e *Kalman*, a qual é responsável por eliminar ruídos na leitura dos ângulos utilizando o método do filtro de Kalman.

Os atributos dessa classe são:

- *accX*, *accY* e *accZ*: aceleração do acelerômetro contido no IMU nos eixos x, y e z, respectivamente;
-
- *gyroX*, *gyroY* e *gyroZ*: armazena a posição do giroscópio no espaço;
- *accXangle*: Ângulo do eixo x calculado utilizando o acelerômetro;
- *gyroXangle*: Ângulo do eixo x calculado utilizando o giroscópio;
- *kalAngleX*: Ângulo do eixo x calculado utilizando o filtro de Kalman;
- *i2cData*: buffer com as informações necessárias para acessar os registradores da MPU;
- *gyroXrate*: Variação do ângulo do eixo x do giroscópio;
- *kalmanX*: objeto da classe Kalman necessário para filtrar o ângulo calculado com base nas informações da MPU..

Os métodos dessa classe são:

```
#ifndef Particle_H
#define Particle_H

class Particle {
  public:
    Particle(float p, float i, float d);

    void setCurrentKp(float p);
    void setCurrentKd(float d);
    void setCurrentKi(float i);
    void setPbestKp(float p);
    void setPbestKd(float d);
    void setPbestKi(float i);
    void setPID(float p, float i, float d);
    void setPbestPID(float p, float i, float d);
    void setPbestFitness(float pb);
    void setPreviousVelocityKp(float vel);
    void setPreviousVelocityKi(float vel);
    void setPreviousVelocityKd(float vel);
    void setCurrentFitness(float s);
    void setAllVelocity(float p, float i, float d);

    float getPbestFitness();
    float getCurrentKp();
    float getCurrentKd();
    float getCurrentKi();
    float getPbestKp();
    float getPbestKd();
    float getPbestKi();
    float getPreviousVelocityKp();
    float getPreviousVelocityKi();
    float getPreviousVelocityKd();
    float getCurrentFitness();

  private:
    float kp;
    float ki;
    float kd;
    float best_kp;
    float best_ki;
    float best_kd;
    float pBest_Fitness;
    float pVelp;
    float pVeli;
    float pVeld;
    float Fitness;
};
```

Figura 27 – Header da classe TiltSensor.

- *getXaxis*: calcula o ângulo do eixo x da MPU em relação ao eixo do espaço. Seu parâmetro é *temp*, que indica o tempo decorrido desde a inicialização do microcontrolador;
- *begin*: inicializa a IMU.

Os métodos dessa classe são *get* e *set* dos atributos.

VNS

Essa classe possui os métodos e atributos necessários para a execução do VNS. A Figura 28 apresenta o *header* da classe *VNS*.

```
#ifndef VNS_h
#define VNS_h

#include "Individual.h"
#include "Parameters.h"
#include "PID.h"
#include "Motor.h"

class VNS{
  public:
    VNS();
    Individual run(Motor *m, PID *p, TiltSensor *s);
    void localSearch(Motor *m, PID *p, TiltSensor *s, int iter, int *k);
    int randomInt();
    float evaluate(Motor *m, PID *p, TiltSensor *s);
    float randomFloat(float minf, float maxf);
    void generateSolution(Motor *m, PID *p, TiltSensor *s);
    void getPID(float *kp, float *ki, float *kd);
    void printInTxt(String type, int iteration, int numberLocalSearch);
    void changeNeighborhood(int *k);

  private:
    Individual *solution;
    float neighborhoodRadious;
};

#endif
```

Figura 28 – Header da classe VNS.

Os atributos dessa classe são:

- *solution*: objeto da classe *Individual*.
- *neighborhoodRadious*: valor do intervalo de busca local do VNS (Conforme discutido na Seção 2.1.

Os métodos dessa classe são:

- *run*: Realiza o controle das gerações do VNS. Seus parâmetros são os objetos das classes: *Motor*, *PID* e *TiltSensor*. Esse método retorna a solução encontrada pelo VNS-PID;
- *localSearch*: realiza a busca local. Seus parâmetros são os objetos das classes: *Motor*, *PID* e *TiltSensor*; *iter*, que é a iteração em que o algoritmo está e *k*, que é a vizinhança atual;

- *evaluate* e *randomFloat*: possuem a mesma finalidade dos métodos de mesmo nome da classe *GA* e *PSO*;
- *generateSolution*: gera a solução inicial do VNS-PID (Conforme discutido na Seção 2). Seus parâmetros são os objetos das classes: *Motor*, *PID* e *TiltSensor*;
- *getPID*: método para atribuir os valores das constantes PID da solução para outra variável. Esse método serve para diminuir a quantidade de código;
- *printInTxt*: exibe informações do processo do VNS para o usuário;
- *changeNeighborhood*: muda a vizinhança da solução gerada. Seu parâmetro é *k*, que é a vizinhança atual do algoritmo.

O *header*: *Parameters* contém todas a definição das constantes das classes, como: o número de partículas, número máximo de gerações, etc. O seu objetivo é facilitar a alteração dessas constantes no código.

A classe *Kalman* foi feita por Lauszus (2012).

Vale ressaltar que a divisão em classes permite que o código elaborado possa ser mais facilmente utilizado em uma outra aplicação. Por exemplo, caso se queira fazer o controle PID de um robô do tipo pêndulo invertido, que tenha motores com escova, pode-se alterar somente a classe *Motor* para alterar a forma como se aciona os motores. Além disso, caso se deseja implementar outro controlador, que não seja o PID, basta alterar a implementação da classe *PID*.

O código completo utilizado neste trabalho pode ser acessado pelo link: <https://github.com/SilvaGAL/Autotune-PID-Controller.git>.

3.4 Código do VNS-PID

Conforme descrito na Seção 2.4, a solução do sistema é um conjunto de três constantes PID: K_p, K_i e K_d. Neste trabalho, as constantes PID são limitadas entre 0,0 e 7,0. O limite superior é definido como 7,0 por experimentos empíricos com este birrotor em específico.

No VNS-PID apresentado na Seção 2.4, o vetor da solução armazena quatro informações diferentes: K_p, K_i, K_d e a pontuação final da solução. O número máximo de pesquisas locais sem aprimoramento (*k*) e o número máximo de pesquisas locais a serem executadas (*i*) são definidas para 6 e 15, respectivamente. Esses valores foram definidos de forma empírica, tendo em vista que falta na literatura trabalhos que utilizam o VNS para sintonizar um controlador PID (conforme visto na Seção 1). Com isso, não se tem uma relação entre custo computacional e número de buscas viável para se aplicar no contexto deste trabalho.

Além disso, a estrutura de vizinhança é descrita como a adição de um valor aleatório diferente, inicialmente, no intervalo $[-0,40\ ;\ 0,40]$ para cada constante PID. Como se pode

observar, o valor a ser adicionado pode ser tanto negativo quanto positivo. Então pode-se adicionar ou subtrair um valor das constantes PID, de acordo com a estrutura de vizinhança. Tendo isso em mente, há sete estruturas de vizinhança diferentes:

1. adição em K_p;
2. adição em K_i;
3. adição em K_d;
4. adição em K_p e K_i;
5. adição em K_p e K_d;
6. adição em K_i e K_d;
7. adição em K_p, K_i e K_d.

Com relação ao intervalo, sempre que uma busca local encontra uma solução melhor, ele é dividido por dois. Para deixar mais clara a ideia sobre o intervalo e as estruturas de vizinhança, considere o seguinte exemplo: o algoritmo do VNS-PID encontrou a primeira solução e agora vai realizar uma busca local na primeira estrutura de vizinhança, que no caso é adicionar um valor em K_p. A cada iteração da busca local nessa estrutura, um valor aleatório, dentro do intervalo, é gerado e adicionado somente em K_p da solução gerada. Se a busca local encontrar uma solução melhor, seja qual for a iteração, ela é interrompida e considera-se agora a solução encontrada como a nova solução gerada. Além disso, o intervalo é dividido por dois passando para $[-0,20\ ;\ 0,20]$. Essa estratégia visa encontrar o máximo local da solução na estrutura de vizinhança em que está inserida. A busca local nessa estrutura continuará até que se realize uma busca local completa sem encontrar uma solução melhor. Caso isso aconteça, passasse para a segunda estrutura de vizinhança, que no caso é adição somente em K_d. Além disso, o intervalo volta a ser $[-0,40\ ;\ 0,40]$. Esses eventos ocorrem até se chegar a última estrutura de vizinhança.

A função $generateSolution$ gera 6 soluções aleatórias e retorna a melhor. A função $randomNeighbor$ retorna um vizinho (s') de uma solução (s) adicionando um valor aleatório no intervalo de vizinhança atual para cada constante PID. Por fim, a função $localSearch$ busca 20 novas soluções aleatórias na estrutura de vizinhança s' por meio da adição de um valor aleatório em seu intervalo atual para cada constante PID (semelhante ao $randomNeighbor$). A estrutura de vizinhança será uma das sete descritas anteriormente, dependendo do valor de k. Além disso, também é aplicado o *first improvement* na função $localSearch$ para reduzir o número de avaliações.

3.5 Código GA-PID

O tamanho da população é um parâmetro relevante para a convergência de um algoritmo evolucionário como Reeves (1993) destaca. Em seu trabalho, o autor apresentou como um algoritmo com uma pequena população pode aumentar a chance de convergência prematura para uma solução. Soluções maiores, por outro lado, podem acarretar em um custo de processamento maior. O autor também apresenta alguns trabalhos experimentais sugerindo que populações entre n e $2n$ fornecem melhores resultados, em que n é o número de variáveis a serem otimizadas. Por essas razões, como se está otimizando 3 variáveis (K_p, K_i, K_d), realizou-se os experimentos com um grupo de seis soluções, que são evoluídas ao longo de 15 gerações. Esse número foi escolhido com base no número de soluções médio que o VNS-PID avalia, que é 120. Com o número de gerações e indivíduos, tem-se 120 soluções analisadas pelo GA-PID, o mesmo número de soluções é analisado pelo PSO-PID. Com isso, faz-se uma comparação justa em os algoritmos evolucionários e de busca local.

As probabilidades de cruzamento e mutação são escolhidas com base no trabalho apresentado em Srinivas e Patnaik (1994). Neste trabalho, os autores sugerem que valores baixos do parâmetro de cruzamento e valores altos do parâmetro de mutação podem transformar o GA em uma busca aleatória. Por esse motivo, a fim de evitar a aleatoriedade do sistema e, ainda assim, garantir que o algoritmo busque o espaço de soluções como um todo, definimos as probabilidades de cruzamento e mutação em 95% e 5%, respectivamente.

O vetor de uma solução individual armazena quatro informações diferentes: K_p, K_i, K_d e aptidão da solução em questão.

3.6 Código PSO-PID

No que tange as características específicas do PSO, na Equação 2.6 utilizou-se c_1 e c_2 iguais a 2, r_{1j} e r_{2j} como valores aleatórios entre 0,0 e 1,0 e a função de peso de inércia calculada pela Equação 3.1:

$$w = I_{max} - i \cdot \frac{I_{max} - I_{min}}{i_{max}} \tag{3.1}$$

em que I_{max} é o peso máximo inercial, I_{min} o peso mínimo inercial, i é a geração atual e i_{max} é o número máximo de gerações.

O vetor de partículas do PSO armazena onze informações diferentes: K_p atuais, K_i atuais, K_d atuais, pontuação de aptidão atual, *pBest* K_p, *pBest* K_i, *pBest* K_d, *pBest* pontuação de aptidão, velocidade anterior de K_p, K_i e K_d. Com isso, como o PSO usa mais variáveis do que o GA, o uso de memória é maior. Utilizou-se um grupo de 6 partículas que "voam"pelo espaço de busca durante 15 gerações.

4 RESULTADOS

Nesta seção, são apresentados e discutidos os resultados alcançados com a abordagem proposta.

O fato de que os resultados apresentarem aptidão abaixo de 95% é aceitável, uma vez que o birrotor inicia com um motor na parte superior e outro na parte inferior. Com isso, o sistema precisa de um tempo para estabilizar até chegar em sua referência, conforme descrito na Seção 2.4. Durante esse tempo alguns pontos são descontados de sua aptidão. Essa foi uma forma de eliminar soluções com tempo de subida maior.

Como pode ser visto na Tabela 1, a qual apresenta cinco execuções do VNS-PID. Nessa Tabela, a “Primeira solução” representa a solução inicial encontrada pela função *generateSolution* (Conforme explicado na Seção 3), enquanto que a “Melhor solução” representa a solução final encontrada após toda a execução do VNS-PID. Observando-se a Tabela 1 a “Melhor solução”de cada execução é diferente entre si, o que significa que o algoritmo pesquisou vizinhanças diferentes e, consequentemente, encontrou soluções diferentes. Além disso, é possível verificar que o VNS-PID melhorou a solução inicial em todas as execuções feitas. Vale ressaltar que os valores de aptidão na Tabela 1 apresentam uma média igual a 87,67 e um desvio padrão igual a 2,50, ou seja, as soluções finais têm uma consistência de execução semelhante mesmo partindo de soluções geradas diferentes.

Além disso, verifica-se que os valores de K_p (Tabela 1) estão abaixo de 1,5. Isso se deve ao fato de que a estrutura que contém os motores é leve. Por esse motivo, um ganho proporcional alto levaria o sistema a um comportamento instável, uma vez que o sobressinal seria mais alto e os ganhos derivativos e integrais não seriam capazes de estabilizar o sistema.

Tabela 1 – Resultados das cinco execuções do VNS-PID no birrotor

Execução	Primeira solução				Melhor solução			
	K_p	K_i	K_d	Aptidão	K_p	K_i	K_d	Aptidão
1	2,19	3,05	2,51	71,08	1,43	3,13	2,51	85,43
2	1,04	0,83	3,67	87,22	0,57	0,83	4,68	86,52
3	1,10	2,43	0,09	79,32	0,91	2,43	0,09	85,63
4	1,84	1,51	1,91	71,84	0,69	1,27	1,91	92,12
5	1,26	1,34	0,19	86,71	0,63	1,34	0,19	88,67

A Tabela 2 apresenta os resultados dos cinco testes feitos com o PSO-PID. A “Primeira solução” representa a melhor solução dentre as soluções do enxame após a primeira geração. A “Melhor solução” representa a solução final encontrada pelo PSO-PID. Como pode-se observar na Tabela 2, em todas as execuções houve melhora das soluções. Ademais, diferentemente do que foi observado nas execuções do VNS-PID (Tabela 1), as constantes proporcionais (K_p) de

todos os testes foram abaixo de 1. Vale ressaltar que os valores de aptidão na Tabela 2 apresentam uma média igual a 86,06 e um desvio padrão igual a 2,63, valores piores aos observados nas execuções do VNS-PID, uma vez que a média de aptidão das soluções encontradas são menores e o desvio padrão maior.

Tabela 2 – Resultados das cinco execuções do PSO-PID no birrotor

Execução	Melhor partícula do primeiro enxame				Melhor partícula encontrado			
	K_p	K_i	K_d	Aptidão	K_p	K_i	K_d	Aptidão
1	0,23	2,04	0,81	73,42	0,64	2,17	0,97	85,01
2	0,61	0,71	0,39	81,05	0,67	0,83	3,34	89,05
3	0,66	2,08	1,20	85,51	0,73	1,45	2,25	86,06
4	0,39	0,04	2,01	86,61	0,47	1,43	2,97	88,48
5	0,48	0,31	6,09	75,06	0,50	0,12	3,87	81,73

A Tabela 3 apresenta os resultados referentes ao GA-PID. A "Primeira solução" representa a melhor solução dentre as soluções da população após a primeira geração. A "Melhor solução" representa a solução final encontrada pelo GA-PID. Através da Tabela 3, observa-se que em uma execução não houve melhora (teste 5) e, em outras duas (testes 2 e 3), a melhora obtida não foi relevante em comparação às obtidas nas execuções do VNS-PID (Tabela 1) e do PSO (Tabela 2). Uma das possíveis razões para isso pode ser a elitização feita no GA desenvolvido para este trabalho, em que apenas os indivíduos com as melhores aptidões perduram para a próxima geração. Com isso, as gerações posteriores possuem uma tendência maior a serem descendentes dos mesmos pais. Com isso, a variabilidade genética da população diminui, o que inviabiliza a exploração do espaço de busca de forma satisfatória. Dessa forma, o algoritmo fica preso em um máximo local. Vale ressaltar que os valores de aptidão na Tabela 3 apresentam uma média igual a 82,06 e um desvio padrão igual a 6,68, valores piores aos observados nas execuções do VNS-PID e PSO-PID.

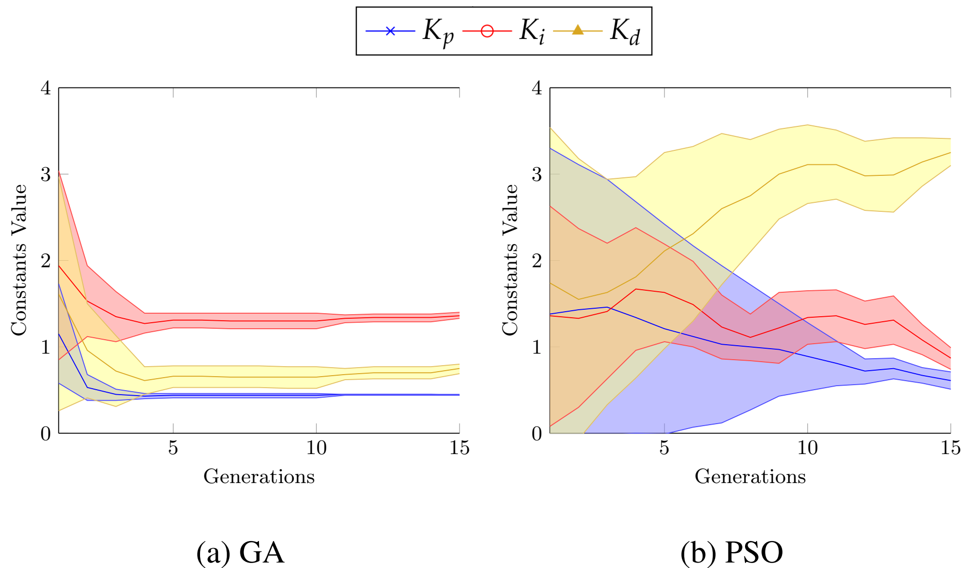
Tabela 3 – Resultados das cinco execuções do GA-PID no birrotor

Execução	Melhor indivíduo da primeira população				Melhor indivíduo encontrado			
	K_p	K_i	K_d	Aptidão	K_p	K_i	K_d	Aptidão
1	0,38	1,10	0,38	90,00	0,44	1,38	0,77	91,39
2	0,64	4,55	1,11	71,24	0,65	4,54	1,12	73,46
3	1,32	3,18	6,07	65,47	1,78	1,46	4,10	75,46
4	0,72	0,95	4,77	74,49	0,78	0,90	4,22	84,13
5	0,56	0,22	0,56	85,87	0,56	0,22	0,56	85,87

Em nenhum dos testes houve soluções similares. Ou seja, o espaço de busca para o problema estudado neste trabalho é muito amplo.

A Figura 29 apresenta o gráfico referente à média e desvio padrão da população ao longo das gerações da melhor execução do GA (Figura 29 a) e do PSO (Figura 29 b). Observa-se

que na Figura 29 a o desvio padrão inicial e final são menores do que aqueles observados na Figura 29 b. Esse resultado está diretamente ligado às características específicas tanto do GA-PID quanto do PSO-PID. Enquanto o primeiro possui um processo elitista e de seleção, o segundo possui um processo cooperativo entre as soluções. Ou seja, na abordagem do GA as soluções com características desfavoráveis para o problema possuem uma probabilidade pequena de transmitir suas características para a próxima geração. Com isso, a população das gerações mais avançadas, tendem a ter características mais parecidas entre si. Além disso, como salientado anteriormente, o processo de elitização empregado neste trabalho contribui para que os descendentes das últimas gerações tenham pais muito parecidos. O PSO, em contrapartida, emprega um modelo cooperativo. Dessa maneira, as soluções com menor aptidão continuam sendo soluções candidatas. Nesse cenário, sua posição é atualizada em cada geração com o objetivo de se aproximar da solução com melhor posição. Por isso, o PSO possui um desvio muito alto no início e muito menor no final.



(a) GA (b) PSO

Figura 29 – Média e desvio padrão ao longo das gerações dos melhores testes de GA e PSO

A Figura 30 apresenta o comportamento da primeira solução e da solução final da quarta execução do VNS-PID (consulte Tabela 1), a qual apresentou a melhor solução dentre todas as geradas. É possível verificar que o termo integral (K_i) da primeira solução diminuiu em relação à solução gerada. O termo proporcional (K_p) diminuiu, enquanto que o termo K_d aumentou. Dessa forma, conforme discutido na Seção 3, quando K_p ou K_i diminui, o sobressinal tende a diminuir. O mesmo ocorre com o aumento de K_d, uma vez que quando a planta está elevando um de seus motores para atingir a referência, se a velocidade dessa elevação for alta, a constante derivativa vai agir contra essa variação, diminuindo a a potência dos motores. Por esses motivos, o sobressinal da solução apresentada na Figura 30 diminuiu. Além disso, a primeira solução apresenta um erro em regime permanente maior que a solução encontrada pelo VNS-PID. Esse fato ocorre devido a diminuição da constante proporcional, tendo em vista que uma diferença

entre o ângulo atual e a referência gera uma resposta menor por parte do ganho proporcional. Com isso, o ganho dos motores é menor e, consequentemente, o sistema fica mais estável.

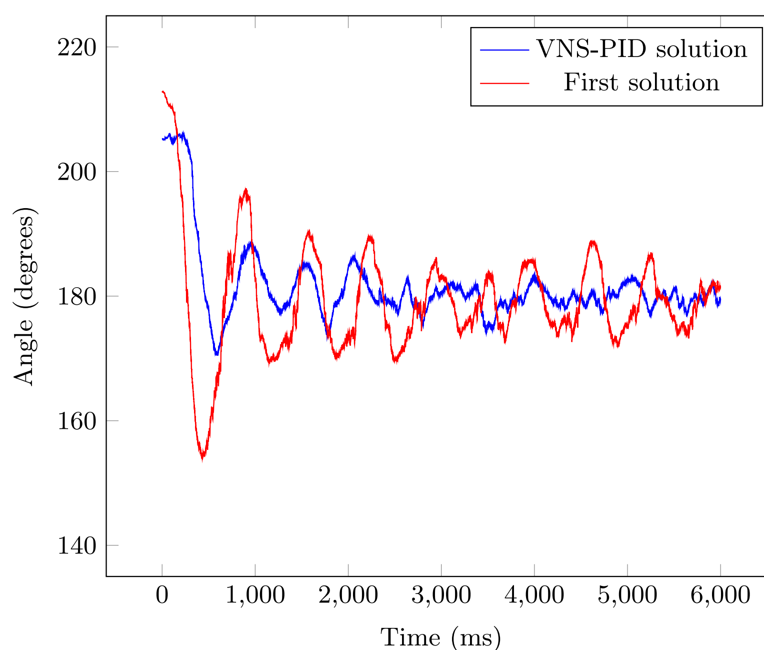


Figura 30 – Comparando a primeira solução com a solução gerada pelo VNS-PID

Por meio dos experimentos realizados e conforme a revisão da literatura feita na Seção 2, faltam trabalhos que abordem um método que combine VNS e PID em uma abordagem *hardware-in-the-loop*. No entanto, existe um número significativo de trabalhos que utilizam métodos que combinam algoritmos evolutivos com PID, como os Algoritmos Evolucionários estudados neste trabalho. Os algoritmos evolutivos apresentam uma grande desvantagem em um cenário embarcado com memória restrita (2 Mb). Eles consomem uma quantidade considerável de memória para lidar com a população e suas operações. É preciso guardar um número significativo de informações de cada indivíduo. O VNS, por outro lado, consome menos memória, uma vez que trabalha com apenas uma única solução.

Como foi visto nesta seção, o VNS mostrou o melhor resultado dentre todas as heurísticas abordadas. Dessa forma, ele se mostra como uma abordagem viável para a calibração das constantes PID em sistemas embarcados de baixo poder computacional.

5 CONCLUSÃO

Neste trabalho, é proposto o uso da meta-heurística VNS para a calibração dos parâmetros de um controlador PID com o objetivo de estabilizar um sistema birrotor em uma abordagem *hardware-in-the-loop*. Além disso, faz-se uma comparação entre o VNS e os algoritmos evolucionários: GA e PSO. A contribuição deste trabalho é a elaboração de um método de auto ajuste das constantes PID utilizando o algoritmo de busca local VNS. Outra contribuição deste trabalho é a avaliação da abordagem baseada em VNS em relação à métodos evolutivos.

Experimentos revelaram a viabilidade de se equilibrar um birrotor com o VNS-PID. O algoritmo é capaz de convergir para boas soluções, mesmo usando pouca memória.

Acredita-se que esta abordagem possa ser utilizada em sistemas reais ou comerciais, ajudando a ajustar o controlador PID a novos ambientes ou mesmo modificações de última hora no design. Além disso, a abordagem pode ser útil para leigos na área de controle, principalmente no que se refere ao ajuste de um controlador PID, sem a necessidade de modelagem matemática da planta.

Trabalhos futuros incluem (i) propor uma nova função de aptidão, a qual leve em consideração e penalize os erros de sobressinal, tempo de subida e tempo de acomodação e (ii) analisar a robustez da abordagem proposta em diferentes ambientes, desde cenários controlados até irrestritos.

REFERÊNCIAS

AHMADI, S.; ABDI, S.; KAKAVAND, M. Maximum power point tracking of a proton exchange membrane fuel cell system using pso-pid controller. *International Journal of Hydrogen Energy*, Elsevier, v. 42, n. 32, p. 20430–20443, 2017. Citado na página 32.

AMARAL, J.; TANSCHEIT, R.; PACHECO, M. Tuning pid controllers through genetic algorithms. *complex systems*, v. 2, p. 3, 2018. Citado na página 13.

ANG, K. H.; CHONG, G.; LI, Y. Pid control system analysis, design, and technology. *IEEE Transactions on Control Systems Technology*, IEEE, v. 13, n. 4, p. 559–576, 2005. Citado 2 vezes nas páginas 13 e 17.

ANGEL, L.; VIOLA, J.; VEGA, M. Hardware in the loop experimental validation of pid controllers tuned by genetic algorithms. In: IEEE. *2019 IEEE 4th Colombian Conference on Automatic Control (CCAC)*. [S.l.], 2019. p. 1–6. Citado na página 15.

ÅSTRÖM, K. J.; HÄGGLUND, T.; ASTROM, K. J. *Advanced PID control*. [S.l.]: ISA-The Instrumentation, Systems, and Automation Society Research Triangle . . . , 2006. v. 461. Citado na página 31.

BINGI, K. et al. Fractional-order filter design for set-point weighted PID controlled unstable systems. *International Journal of Mechanical & Mechatronics Engineering (IJMME)*, v. 17, n. 5, p. 173–179, 2017. Citado na página 31.

BRIMBERG, J. et al. Solving the capacitated clustering problem with variable neighborhood search. *Annals of Operations Research*, Springer, v. 272, n. 1-2, p. 289–321, 2019. Citado na página 13.

CASTELLANOS, J. E. R.; BALLESTEROS, J. E. C. Implementation of a direct fuzzy controller applied to a helicopter with one degree of freedom. *IEEE Latin America Transactions*, IEEE, v. 17, n. 11, p. 1808–1814, 2019. Citado na página 13.

CASTRO, L. et al. Design of pid type local controller network with fuzzy supervision. *IEEE Latin America Transactions*, IEEE, v. 17, n. 05, p. 759–765, 2019. Citado na página 13.

COSTA, A. A. B.; BIAZI, E.; VÍTOR, J. Aplicação da metaheurística pso na identificação de pontos influentes por meio da função de sensibilidade de casos. *TEMA-Tendências em Matemática Aplicada e Computacional*, v. 11, n. 1, p. 41–48, 2010. Citado na página 27.

DAVIS, L. *Handbook of genetic algorithms*. [S.l.]: CUMINCAD, 1991. Citado na página 22.

FANG, M.; ZHUO, Y.; LEE, Z. The application of the self-tuning neural network pid controller on the ship roll reduction in random waves. *Ocean Engineering*, Elsevier, v. 37, n. 7, p. 529–538, 2010. Citado na página 13.

FARINELLI, F. Conceitos basicos de programação orientada a objetos. *Instituto Federal Sudeste de Minas Gerais*, 2007. Citado na página 18.

FENG, H. et al. Robotic excavator trajectory control using an improved ga based pid controller. *Mechanical Systems and Signal Processing*, Elsevier, v. 105, p. 153–168, 2018. Citado na página 16.

FEREIDOUNI, A.; MASOUM, M. A.; MOGHBEL, M. A new adaptive configuration of pid type fuzzy logic controller. *ISA transactions*, Elsevier, v. 56, p. 222–240, 2015. Citado na página 13.

FORTES, E. de V. et al. A VNS algorithm for the design of supplementary damping controllers for small-signal stability analysis. *International Journal of Electrical Power & Energy Systems*, Elsevier, v. 94, p. 41–56, 2018. Citado na página 13.

FRAGA-GONZALEZ, L. F. et al. Adaptive simulated annealing for tuning pid controllers. *AI Communications*, IOS Press, v. 30, n. 5, p. 347–362, 2017. Citado na página 15.

FRANKLIN, G. F. et al. *Feedback control of dynamic systems*. [S.l.]: Addison-Wesley Reading, MA, 1994. v. 3. Citado na página 31.

GAING, Z.-L. A particle swarm optimization approach for optimum design of pid controller in avr system. *IEEE transactions on energy conversion*, IEEE, v. 19, n. 2, p. 384–391, 2004. Citado na página 32.

GIRIRAJKUMAR, S.; JAYARAJ, D.; KISHAN, A. R. Pso based tuning of a pid controller for a high performance drilling machine. *International Journal of Computer Applications*, Citeseer, v. 1, n. 19, p. 12–18, 2010. Citado na página 13.

HANSEN, P.; MLADENOVIĆ, N. Variable neighborhood search: Principles and applications. *European journal of operational research*, Elsevier, v. 130, n. 3, p. 449–467, 2001. Citado na página 13.

HERNÁNDEZ-ALVARADO, R. et al. Neural network-based self-tuning pid control for underwater vehicles. *Sensors*, Multidisciplinary Digital Publishing Institute, v. 16, n. 9, p. 1429, 2016. Citado na página 12.

KROHLING, R. A.; REY, J. P. Design of optimal disturbance rejection pid controllers using genetic algorithms. *IEEE Transactions on Evolutionary computation*, IEEE, v. 5, n. 1, p. 78–82, 2001. Citado 2 vezes nas páginas 13 e 32.

KUMARI, S. et al. Ga based design of current conveyor pld controller for the speed control of bldc motor. In: IEEE. *2018 4th International Conference on Computational Intelligence & Communication Technology (CICT)*. [S.l.], 2018. p. 1–3. Citado na página 13.

LAUSZUS, K. *A practical approach to Kalman filter and how to implement it*. 2012. Disponível em: <http://blog.tkjelectronics.dk/2012/09/a-practical-approach-to-kalman-filter-and-how-to-implement-it/#comment-57783>. Citado 2 vezes nas páginas 30 e 49.

LEE, E. A.; SESHIA, S. A. *Introduction to embedded systems: A cyber-physical systems approach*. [S.l.]: Mit Press, 2016. Citado na página 12.

LIMA, G. V. et al. Stabilization and path tracking of a mini quadrotor helicopter: experimental results. *IEEE Latin America Transactions*, IEEE, v. 17, n. 03, p. 485–492, 2019. Citado na página 13.

MLADENOVIĆ, N.; HANSEN, P. Variable neighborhood search. *Computers & operations research*, Elsevier, v. 24, n. 11, p. 1097–1100, 1997. Citado na página 19.

MOUSAKAZEMI, S. M. H.; AYOOBIAN, N. Robust tuned pid controller with pso based on two-point kinetic model and adaptive disturbance rejection for a pwr-type reactor. *Progress in Nuclear Energy*, Elsevier, v. 111, p. 183–194, 2019. Citado na página 16.

MUKHTAR, A.; TAYAL, V. K.; SINGH, H. Pso optimized pid controller design for the process liquid level control. In: IEEE. *2019 3rd International Conference on Recent Developments in Control, Automation & Power Engineering (RDCAPE)*. [S.l.], 2019. p. 590–593. Citado 2 vezes nas páginas 13 e 16.

OGATA, K.; YANG, Y. *Modern control engineering*. [S.l.]: Prentice-Hall, 2002. v. 4. Citado na página 13.

OSINSKI, C.; LEANDRO, G. V.; OLIVEIRA, G. H. da C. Fuzzy pid controller design for lfc in electric power systems. *IEEE Latin America Transactions*, IEEE, v. 17, n. 01, p. 147–154, 2019. Citado na página 15.

PEI, J. et al. Continuous variable neighborhood search (c-vns) for solving systems of nonlinear equations. *INFORMS Journal on Computing*, INFORMS, v. 31, n. 2, p. 235–250, 2019. Citado na página 13.

REEVES, C. R. Using genetic algorithms with small populations. In: *ICGA*. [S.l.: s.n.], 1993. v. 590, p. 92. Citado na página 51.

ROSALES, C. D. et al. Neural adaptive pid control of a quadrotor using efk. *IEEE Latin America Transactions*, IEEE, v. 16, n. 11, p. 2722–2730, 2018. Citado na página 15.

SHI, Y.; EBERHART, R. C. Empirical study of particle swarm optimization. In: IEEE. *Proceedings of the 1999 Congress on Evolutionary Computation-CEC99 (Cat. No. 99TH8406)*. [S.l.], 1999. v. 3, p. 1945–1950. Citado na página 24.

SHI, Y. et al. Particle swarm optimization: developments, applications and resources. In: IEEE. *Proceedings of the 2001 congress on evolutionary computation (IEEE Cat. No. 01TH8546)*. [S.l.], 2001. v. 1, p. 81–86. Citado na página 24.

SHU, H.; PI, Y. Pid neural networks for time-delay systems. *Computers & Chemical Engineering*, Elsevier, v. 24, n. 2-7, p. 859–862, 2000. Citado na página 13.

SRINIVAS, M.; PATNAIK, L. M. Adaptive probabilities of crossover and mutation in genetic algorithms. *IEEE Transactions on Systems, Man, and Cybernetics*, IEEE, v. 24, n. 4, p. 656–667, 1994. Citado na página 51.

STEWART, P.; STONE, D.; FLEMING, P. J. Design of robust fuzzy-logic control systems by multi-objective evolutionary methods with hardware in the loop. *Engineering Applications of Artificial Intelligence*, Elsevier, v. 17, n. 3, p. 275–284, 2004. Citado na página 12.

TAHRI, A. et al. Nonlinear adaptive control of a hybrid fuel cell power system for electric vehicles–a lyapunov stability based approach. *Asian Journal of Control*, Wiley Online Library, v. 18, n. 1, p. 166–177, 2016. Citado na página 31.

VISIOLI, A. Tuning of pid controllers with fuzzy logic. *IEE Proceedings-Control Theory and Applications*, IET, v. 148, n. 1, p. 1–8, 2001. Citado na página 13.

WELCH, G.; BISHOP, G. et al. *An introduction to the Kalman filter*. [S.l.]: Citeseer, 1995. Citado na página 30.

WHITLEY, D. A genetic algorithm tutorial. *Statistics and computing*, Springer, v. 4, n. 2, p. 65–85, 1994. Citado na página 22.

ZAHIR, A. et al. Genetic algorithm optimization of pid controller for brushed dc motor. In: *Intelligent Manufacturing & Mechatronics*. [S.l.]: Springer, 2018. p. 427–437. Citado na página 16.

ZENG, G.-Q. et al. Design of fractional order pid controller for automatic regulator voltage system based on multi-objective extremal optimization. *Neurocomputing*, v. 160, p. 173 – 184, 2015. ISSN 0925-2312. Citado na página 16.

ZHANG, J. et al. Self-organizing genetic algorithm based tuning of pid controllers. *Information Sciences*, Elsevier, v. 179, n. 7, p. 1007–1018, 2009. Citado 2 vezes nas páginas 15 e 32.

ZHAO, Y. et al. A novel algorithm for wavelet neural networks with application to enhanced pid controller design. *Neurocomputing*, Elsevier, v. 158, p. 257–267, 2015. Citado na página 13.

ZIEGLER, J. G.; NICHOLS, N. B. Optimum settings for automatic controllers. *trans. ASME*, v. 64, n. 11, 1942. Citado na página 13.