UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
ESCOLA DE ENGENHARIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA
PROJETO DE DIPLOMAÇÃO EM ENGENHARIA ELÉTRICA

MAURÍCIO SPIAZZI RICHTER

APLICAÇÃO DO MÉTODO IFT (ITERATIVE FEEDBACK TUNING) A UM QUADRICÓPTERO

Porto Alegre
2015

MAURÍCIO SPIAZZI RICHTER

APLICAÇÃO DO MÉTODO IFT (ITERATIVE FEEDBACK TUNING) A UM QUADRICÓPTERO

Projeto de Diplomação submetido ao Curso de Engenharia Elétrica da Universidade Federal do Rio Grande do Sul, como requisito parcial à obtenção do título de Engenheiro Eletricista.

Orientador: Prof. Dr. Alexandre Sanfelice Bazanella

Porto Alegre

2015

AGRADECIMENTOS

Aos meus pais, Carlos e Magali, e à minha irmã, Letícia, pelo apoio incondicional, em todos os momentos.

À minha namorada, Jeanne, por estar sempre ao meu lado, em qualquer situação.

Aos meus avós, pela ajuda, sempre que necessário, e pelo carinho.

Aos meus colegas e amigos que, longe ou perto, fizeram parte desta jornada.

À UFRGS e a todos os professores, em especial ao meu orientador Alexandre Sanfelice Bazanella e ao professor Diego Eckhard, pelo suporte a este projeto.

RESUMO

Os quadricópteros têm sido objeto de muitos estudos relacionados a sistemas de controle nos últimos anos. Dentro desse contexto, este projeto tem o objetivo de testar a implementação do *Iterative Feedback Tuning* (IFT), que é um método de controle baseado em dados para o auto-ajuste de parâmetros do controlador, em um quadricóptero. O relatório também explica aspectos básicos da dinâmica dos quadricópteros, assim como da teoria relacionada ao IFT. O método utilizado foi efetivo nesta aplicação, pois os resultados adquiridos mostram um comportamento similar entre o algoritmo implementado e as simulações feitas.

Palavras-chave: Quadricóptero. *Iterative Feedback Tuning*. Controle baseado em dados.

ABSTRACT

Quadcopters have been the subject of much research related to control systems in the last few years. In this context, this project aims to test the implementation of the Iterative Feedback Tuning (IFT), which is a data-driven control method for the auto-tuning of controller parameters, in a quadcopter. The report also provides the basics of quadcopter dynamics as well as background on the IFT theory. The method has proven to be effective in this application, because the results acquired show a similar behaviour between the implemented algorithm and the simulations performed.

Keywords: Quadcopter. Iterative Feedback Tuning. Data-driven control.

LISTA DE ILUSTRAÇÕES

SUMÁRIO

1 INTRODUÇÃO

Os quadricópteros fazem parte da categoria de veículos aéreos não tripulados, sendo em sua maioria de tamanho reduzido. Eles se assemelham a helicópteros com quatro propulsores, igualmente espaçados no formato de um losango. Devido à sua agilidade e capacidade de realizar pequenas missões específicas, principalmente de vigilância ou produção de imagens aéreas artísticas, eles têm sido objeto de estudos já há algum tempo, tendo seu uso largamente difundido nos últimos anos.

Um dos focos de estudo neste assunto é o controle do quadricóptero, que é também o escopo deste trabalho. De forma simplificada, a força exercida pelo conjunto de quatro hélices é capaz de gerar variação nos ângulos e na posição do quadricóptero, sendo assim possível controlar estes parâmetros. Portanto, malhas de controle podem ser incluídas com o objetivo de controlar e verificar, através de sensores, tanto a variação quanto o estado atual de tais parâmetros, bem como estabelecer critérios de desempenho desejados. Este assunto será tratado de forma mais detalhada nos próximos capítulos.

Um ou mais microcontroladores podem ser utilizados para comandar um quadricóptero, fazendo parte assim do sistema, que, ao receber uma referência a seguir, calcula um erro e alimenta o microcontrolador. Esta referência pode ser, no caso do quadricóptero utilizado neste projeto, de posição ou de ângulo, e é digitalizada antes de ser utilizada pelo microcontrolador (no caso de haver uma interface analógica com o usuário, como um controle remoto). A saída digital deste controlador é que excita os motores do quadricóptero, gerando PWM (*pulse width modulation*, ou modulação por largura de pulso) e consequentemente uma tensão equivalente, sendo então produzido torque nos motores.

Um tipo de controlador que pode ser utilizado para tal função é o PID (proporcional-integral-derivativo), já largamente conhecido e difundido na área de controle, e que é ilustrado mais adiante no trabalho. Na aplicação prática deste projeto utiliza-se apenas a parte proporcional desta categoria, sendo, entretanto, expansível às partes integral e derivativa em trabalhos futuros.

Para que seja utilizada esta estratégia de controle, são necessários ajustes aos parâmetros que definem o controlador, e a proposta geral deste projeto é implementar no microcontrolador, aplicando ao quadricóptero, um método de ajuste automático de parâmetros, mais especificamente através do controle baseado em dados, ou *Data-Driven Control*. O método a ser explorado é o IFT (*Iterative Feedback Tuning*), que busca o ajuste dos parâmetros de forma iterativa e, a despeito do método VRFT (*Virtual*

Reference Feedback Tuning, por exemplo, leva em conta a rejeição a ruídos, como mostrado em (Bazanella, et al., 2012).

2 FUNDAMENTAÇÃO TEÓRICA

2.1 DINÂMICA DO QUADRICÓPTERO

A ilustração da Figura 1 mostra, além do formato do quadricóptero, o sentido de rotação das hélices que é utilizado, permitindo que, estando elas à mesma velocidade, o quadricóptero se mantenha em uma condição estável, com altitude definida e paralelo ao horizonte.

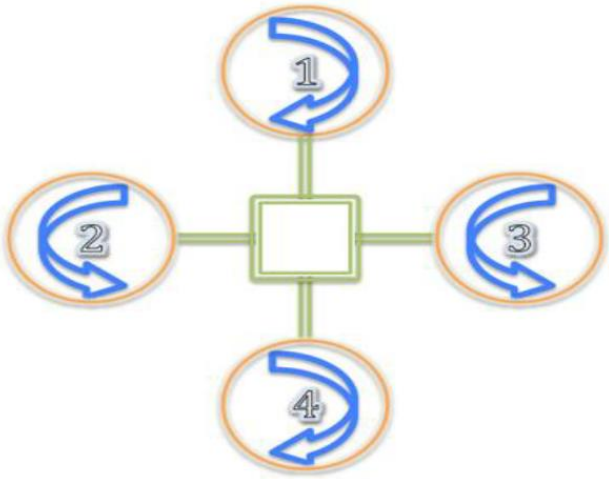

Figura 1. Ilustração de um quadricóptero (retirada de **(Khan, 2014)**), mostrando o sentido de rotação das hélices.

Dentro do contexto sendo estudado, e como explicado em (Mahony, et al., 2002), existem dois sistemas de coordenadas, um referente ao chão, chamado de inercial (*inertial frame*), e outro referente ao corpo do quadricóptero (*body frame*).

Para relacionar tais sistemas de coordenadas, definindo assim a atitude do quadricóptero, existe uma matriz de rotação, mostrada na equação (1), baseada nos ângulos de Euler. São eles *yaw* (guinada, representado por ϕ), *pitch* (arfagem, representado por θ) e *roll* (rolamento, representado por ψ), como na Figura 2. É importante notar que existem diferentes convenções para esta sequência de ângulos, e neste trabalho é utilizada a convenção aeronáutica para a atitude, segundo (Mahony, et al., 2002). Além disso, na sequência do texto os ângulos são mencionados pelos seus termos em inglês, por conveniência.

$$R = \begin{pmatrix} c_\theta c_\phi & s_\psi s_\theta c_\phi - c_\psi s_\phi & c_\psi s_\theta c_\phi + s_\psi s_\theta \\ c_\theta s_\phi & s_\psi s_\theta s_\phi + c_\psi c_\phi & c_\psi s_\theta s_\phi - s_\psi c_\phi \\ -s_\theta & s_\psi c_\theta & c_\psi c_\theta \end{pmatrix} \quad \textbf{(1)}$$

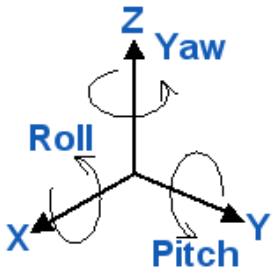

Figura 2. Representação dos ângulos de atitude sobre o *body frame* (retirada de https://developer.valvesoftware.com/w/images/7/7e/Roll_pitch_yaw.gif, acessada em 14/04/2015).

Estes ângulos podem ser medidos e controlados, e de acordo com estes, é possível gerar deslocamento do quadricóptero em relação ao sistema de coordenadas inercial, também controlável, representado pelas variáveis x, y e z, sendo que um deslocamento positivo em z representa uma aproximação do quadricóptero ao chão.

Como o objetivo do trabalho é elaborar o controle do quadricóptero por métodos baseados em dados, de forma que não há necessidade de criar um modelo matemático, então não são abordados aspectos físicos adicionais em relação à dinâmica do quadricóptero, como a transformação eletromecânica dos motores e as forças que são exercidas. Ao invés disso, apenas são definidas algumas variáveis adicionais, as quais são de interesse para o desenvolvimento do projeto. São elas as velocidades lineares $\dot{x}$, $\dot{y}$ e $\dot{z}$, e as velocidades lineares $\dot{\phi}$, $\dot{\theta}$ e $\dot{\psi}$, associadas às variáveis de posição e ângulo anteriormente citadas.

Para fins de simulação, é posteriormente utilizado um modelo matemático já existente para o quadricóptero.

2.2 ESTRATÉGIA DE CONTROLE

Foi utilizado, durante o projeto, o controle proporcional que já estava implementado no microcontrolador utilizado no quadricóptero disponível para o trabalho, sendo que para algumas simulações o controle integral foi também levado em conta.

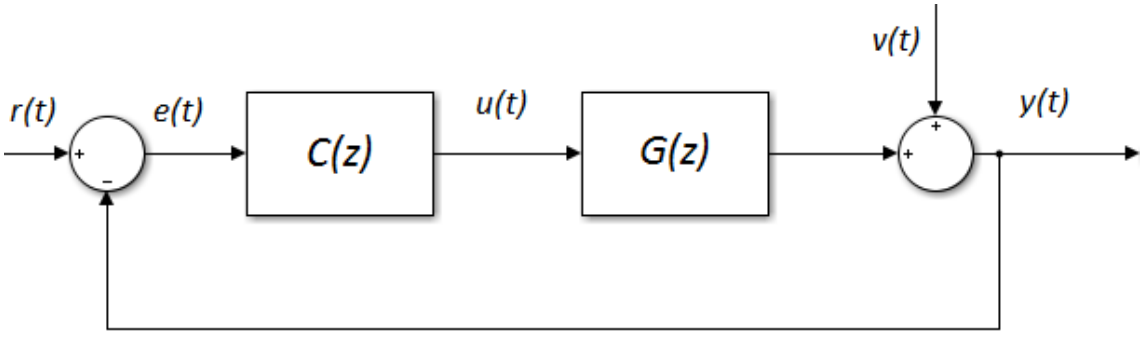

Figura 3. Diagrama de blocos que representa os sinais, o controlador e o processo do sistema.

O diagrama de blocos do sistema é representado na Figura 3, onde: *r(t)* é o sinal de referência, *e(t)* é o sinal de erro, *u(t)* é o sinal de controle, *v(t)* é o ruído na medição da saída, *y(t)* é o sinal de saída, *C(z)* representa a função de transferência discreta do controlador e *G(z)* representa a função de transferência discreta do processo.

O cálculo do sinal de saída desta malha fechada é mostrado na equação (2), onde *T(z)* é a realimentação com relação à referência *r(t)* (equação (3)) e *S(z)* é a realimentação em relação ao ruído *v(t)* (equação (4)).

$$y(t) = T(z)r(t) + S(z)v(t) \quad \textbf{(2)}$$

$$T(z) = \frac{C(z)G(z)}{1 + C(z)G(z)} \quad \textbf{(3)}$$

$$S(z) = \frac{1}{1 + C(z)G(z)} \quad \textbf{(4)}$$

Na realidade, esta topologia é aplicável para cada uma das variáveis a serem controladas, sendo que a aplicação prática foi feita para a posição angular do *roll*. Por tal motivo, para efeitos de cálculo de parâmetros do controlador, o acoplamento não-linear existente entre as variáveis do sistema completo não é levado em conta diretamente, apenas indiretamente, através da utilização resposta real do sistema para cada uma delas.

Durante o projeto, o controlador foi aplicado na maior parte dos testes segundo as parametrizações mostradas: na equação (5), no caso do controlador P (proporcional); e nas equações (6) e (7), no caso do controlador PI (proporcional-integral). Os parâmetros do controlador são representados por ρ_1 e ρ_2, e z é a variável discreta no domínio da frequência:

$$C(z,\rho) = \rho^T \bar{C}(z) = [\rho_1][1] \quad \textbf{(5)}$$

$$C(z,\rho) = \rho^T \bar{C}(z) = [\rho_1 \quad \rho_2]\begin{bmatrix} 1 \\ \frac{z}{z-1} \end{bmatrix} \quad \textbf{(6)}$$

$$C(z,\rho) = \rho^T \bar{C}(z) = [\rho_1 \quad \rho_2]\begin{bmatrix} \frac{z}{z-1} \\ \frac{1}{z-1} \end{bmatrix} \quad \textbf{(7)}$$

Como os controladores PID são largamente empregados nos quadricópteros, como em (Mahony, et al., 2002), (Hoffman, et al., 2007) e (Gibiansky, 2012), uma possível aplicação futura é o desenvolvimento e implementação dos algoritmos feitos neste trabalho também para as partes integral e derivativa deste tipo de controlador. A principal vantagem de se adicionar a parte integral no controle é o seguimento de

referência constante com erro nulo, e o principal motivo de se adicionar a parte derivativa no controle é tornar o transiente do sistema mais rápido. Isto é demonstrado em (Bazanella & Silva Jr, 2005), onde os aspectos do controle PID são amplamente explorados.

2.3 CONTROLE BASEADO EM DADOS (DATA-DRIVEN CONTROL)

O controle baseado em dados é uma técnica que contrasta com as mais tradicionais no sentido de não necessitar da construção de um modelo matemático para o processo que está sendo controlado, fazendo uso apenas das informações de entrada e saída do sistema. Exemplos de metodologias seguindo esta técnica são o IFT (*Iterative Feedback Tuning*), o VRFT (*Virtual Reference Feedback Tuning*), o FDT (*Frequency Domain Tuning*) e o CbT (*Correlation-based Tuning*), sendo que todos estes visam minimizar um critério de desempenho conhecido como H_2, de acordo com (Bazanella, et al., 2012).

Devido ao fato de não ser necessário o conhecimento completo da planta controlada, este tipo de controle se torna muito útil na aplicação desenvolvida neste projeto, por dois motivos principais: com possíveis pequenos ajustes, poderá ser utilizado em diferentes quadricópteros; e devido ao modelo de cada quadricóptero ser de difícil cálculo e natureza não-linear, envolvendo muitas variáveis físicas internas e externas e diversos acoplamentos dinâmicos entre as variáveis do sistema.

O critério H_2 pode ser representado por uma função custo dependente desses parâmetros, $J(\rho)$, mostrada nas equações (8) e (9), separadamente para o seguimento de referência e para a rejeição ao ruído, respectivamente, de acordo com (Bazanella, et al., 2012) (y_d representa a saída desejada do sistema, de acordo com um desempenho previamente estabelecido para o sistema, sendo a função de transferência que adquire idealmente esta saída é $T_d(z)$).

$$y_d(t) = T_d(z)r(t)$$
$$y_r(t,\rho) = T(z,\rho)r(t)$$
$$J_y(\rho) = \bar{E}[y_r(t,\rho) - y_d(t)]^2 = \bar{E}\left[\left(T(z,\rho) - T_d(z)\right)r(t)\right]^2 \quad \textbf{(8)}$$

$$y_e(t,\rho) = S(z,\rho)v(t)$$
$$J_e(\rho) = \bar{E}[y_e(t,\rho)]^2 = \bar{E}[S(z,\rho)v(t)]^2 \quad \textbf{(9)}$$

De forma a unir ambas as funções de custo dentro do mesmo critério para minimização, pode ser desenvolvida a equação (10), devido à não-interdependência

entre os sinais de referência e ruído. Há também um possível parâmetro λ, também explicado em (Bazanella, et al., 2012), que poderia ser aplicado ao método com a finalidade de economizar esforço de controle. Entretanto, este não será utilizado neste projeto.

$$J_T(\rho) = \bar{E}[y(t,\rho) - y_d(t)]^2 = \bar{E}\left[\left(T(z,\rho) - T_d(z)\right)r(t)\right]^2 + \bar{E}[S(z,\rho)v(t)]^2$$

$$J_T(\rho) = J_y(\rho) + J_e(\rho) \quad \textbf{(10)}$$

2.3.1 Iterative Feedback Tuning

O IFT é um dos métodos de controle baseado em dados, que busca o auto-ajuste dos parâmetros do controlador, através de iterações que buscam minimizar o critério H_2, previamente discutido.

Para tal, neste método, os parâmetros são calculados em cada iteração levando em conta os parâmetros atuais e buscando, em uma direção definida, a minimização da função custo. De forma geral, a otimização dos parâmetros se dá pela equação (11).

$$\rho_{i+1} = \rho_i - \gamma_i R_i \nabla \mathrm{J}(\rho_i), \quad \textbf{(11)}$$

onde ρ é o vetor de parâmetros do controlador, γ é o tamanho do passo da busca, R é a direção da busca, $\nabla \mathrm{J}(\rho)$ é o gradiente da função custo e i é o índice da iteração.

Dentro deste trabalho, é utilizado o método *"Steepest Descent"* para a procura dos parâmetros ótimos, no qual $R = I$, com γ positivo (equação (12)), sendo então o cálculo de $\nabla \mathrm{J}(\rho)$ a parte principal para tal finalidade, de forma que a busca se dará na direção oposta ao crescimento do gradiente da função custo. Este cálculo é expresso pela equação (13), deduzida e explorada mais a fundo analiticamente em (Bazanella, et al., 2012), onde N é o número de amostras em cada experimento.

$$\rho_{i+1} = \rho_i - \gamma_i \nabla \mathrm{J}(\rho_i) \quad \textbf{(12)}$$

$$\frac{\partial J_T(\rho)}{\partial \rho} = \frac{2}{N}\sum_{t=1}^{N}[y(t,\rho) - y_d(t)]\frac{\partial y(t,\rho)}{\partial \rho} \quad \textbf{(13)}$$

A equação (16), por sua vez, mostra a expressão relativa à estimativa de $\frac{\partial y(t,\rho)}{\partial \rho}$, sendo a equação (14) relativa ao filtro aplicado ao sinal expresso na equação (15) para determinar $\frac{\partial y(t,\rho)}{\partial \rho}$.

$$Q(z,\rho) = \frac{1}{C(z,\rho)}\frac{\partial C(z,\rho)}{\partial \rho} \quad \textbf{(14)}$$

$$\vartheta(t) = T(z,\rho)(r(t) - y(t,\rho)) \quad \textbf{(15)}$$

$$\frac{\partial y(t,\rho)}{\partial \rho} = Q(z,\rho)\vartheta(t) \quad \textbf{(16)}$$

O problema passar a ser, então, o cálculo da equação (15). Entretanto, como também explicado em (Bazanella, et al., 2012), o formato desta equação remete ao cálculo da saída do próprio sistema a uma nova entrada, definida por *(r(t) – y(t,ρ))* (desconsiderando o ruído). Assim, um novo experimento é realizado utilizando esta entrada. O resultado deste novo experimento é a estimativa necessária de $\vartheta(t)$ para o cálculo de $\frac{\partial y(t,\rho)}{\partial \rho}$ e posteriormente de $\frac{\partial J_T(\rho)}{\partial \rho}$.

Devido a estas conclusões, é conveniente definir uma nova nomenclatura para algumas variáveis: $r_1(t)$ é a referência utilizada no primeiro experimento, $u_1(t)$ é o sinal de controle do primeiro experimento, $y_1(t)$ é a saída do primeiro experimento, $r_2(t) = r_1(t) - y_1(t)$ é a referência utilizada no segundo experimento, $u_2(t)$ é o sinal de controle do segundo experimento, $y_2(t)$ é a saída do segundo experimento e $\frac{\partial y_1(t,\rho)}{\partial \rho}$ é a estimativa da derivada parcial de $y_1(t,\rho)$ em relação a ρ. Assim, o algoritmo é desenvolvido como segue, sendo repetido de acordo com o número de iterações desejado ou necessário:

Passo 1: Realizar o primeiro experimento adquirindo $y_1(t)$ e $u_1(t)$ com a referência $r_1(t)$;

Passo 2: Calcular $r_2(t) = r_1(t) - y_1(t)$;

Passo 3: Realizar o segundo experimento adquirindo $y_2(t)$ e $u_2(t)$ com a referência $r_2(t)$;

Passo 4: Calcular $\frac{\partial y_1(t,\rho)}{\partial \rho}$ através da equação (17);

$$\frac{\partial y_1(t,\rho)}{\partial \rho} = \frac{1}{C(z,\rho)}\frac{\partial C(z,\rho)}{\partial \rho} y_2(t) \quad \textbf{(17)}$$

Passo 5: Calcular $\nabla \mathrm{J}(\rho_i)$ através da equação (13);

Passo 6: Calcular o novo vetor de parâmetros através da equação (12).

Neste método, pode ser conveniente dividir $\nabla \mathrm{J}(\rho_i)$ pela sua norma para o cálculo do novo vetor de parâmetros, para que estes parâmetros sejam sempre atualizados de acordo com o tamanho do passo utilizado.

Associado ao método *"Steepest Descent"*, assim que a função custo estiver suficientemente próxima do mínimo, pode ser também utilizado o método *"Newton-Raphson"*, de forma a tornar mais eficiente, neste ponto, a busca pelos parâmetros próximos do ideal. Este método, como explicado em (Bazanella, et al., 2012), possui tamanho de passo fixo e a direção de busca é dada em função dos parâmetros ρ a cada iteração, através do cálculo das raízes da equação $\nabla J = 0$, o que resulta na convergência

tanto para um mínimo quanto para um máximo da função custo. Isto explica a utilização do método apenas após os passos iniciais, que levam a função custo para perto de seu mínimo.

A equação (18) descreve o algoritmo do método *"Newton-Raphson"*, que possui o mesmo formato e parâmetros da equação (11), porém com $\gamma = 1$ e $R_i = (\nabla^2 \mathrm{J}(\rho_i))^{-1}$ (função inversa da matriz Hessiana de *J*):

$$\rho_{i+1} = \rho_i - (\nabla^2 \mathrm{J}(\rho_i))^{-1} \nabla \mathrm{J}(\rho_i) \quad \textbf{(18)}$$

Desta forma, para calcular o novo vetor de parâmetros é necessário estimar não só o gradiente da função custo, como no método *"Steepest Descent"*, mas também a Hessiana da função, a qual é explicada a seguir.

A fim de estimar $\frac{\partial^2 J_T(\rho)}{\partial \rho^2}$, pode-se tomar como base a equação (13) e derivá-la para obter a equação **(19)**, a seguir.

$$\frac{\partial^2 J_T(\rho)}{\partial \rho^2} = \frac{2}{N} \sum_{t=1}^{N} \left[[y(t,\rho) - y_d(t)] \frac{\partial^2 y(t,\rho)}{\partial \rho^2} + \frac{\partial y(t,\rho)}{\partial \rho} \frac{\partial y(t,\rho)}{\partial \rho}^T \right] \quad \textbf{(19)}$$

Para que seja utilizada a equação (19) são necessários dois experimentos adicionais para cada iteração (para o cálculo numérico da derivada de segunda ordem), aumentando o tempo necessário para executar o algoritmo devido às novas aquisições de dados, bem como consumindo mais tempo do processador pela utilização destes dados adicionais. Por tais motivos, uma simplificação pode ser feita excluindo-se o primeiro fator, como mostrado em (Bazanella, et al., 2012), levando em conta que $[y(t,\rho) - y_d(t)]$ se aproxima de zero à medida que a função custo se aproxima de seu mínimo. O resultado é que a direção de busca dos novos parâmetros (R_i) é dada pelo cálculo exposto na equação **(20)**. Este cálculo utiliza a estimativa da derivada parcial da saída em relação ao vetor de parâmetros, que é obtida também no método *"Steepest Descent"*, portanto não há informações adicionais necessárias.

$$R_i = \left(\frac{\partial^2 J_T(\rho)}{\partial \rho^2} \right)^{-1} = \frac{2}{N} \sum_{t=1}^{N} \left[\frac{\partial y(t,\rho)}{\partial \rho} \frac{\partial y(t,\rho)}{\partial \rho}^T \right]^{-1} \quad \textbf{(20)}$$

Esta variação do método, onde o cálculo da Hessiana é aproximado, pode ser considerado um *"Newton-Raphson"* aproximado.

Desta forma, os passos de 1 a 5 do algoritmo são semelhantes aos já explicados anteriormente, seguidos de:

Passo 6: Calcular R_i através da equação (20);

Passo 7: Calcular o novo vetor de parâmetros através da equação (11).

2.4 ASPECTOS DO CONTROLE DO QUADRICÓPTERO

Como já mencionado anteriormente, o controlador proporcional feito anteriormente no quadricóptero foi mantido para o desenvolvimento do trabalho.

De acordo com os códigos que já haviam sido implementados, existem duas malhas de controle em cada um dos sistemas de interesse (sendo os sistemas de interesse os relacionados ao controle dos ângulos *roll* e *pitch*). Estas malhas estão identificadas na Figura 4 como "malha interna" e "malha externa". A malha interna se refere ao controle da velocidade angular e a malha externa se refere ao controle da posição angular.

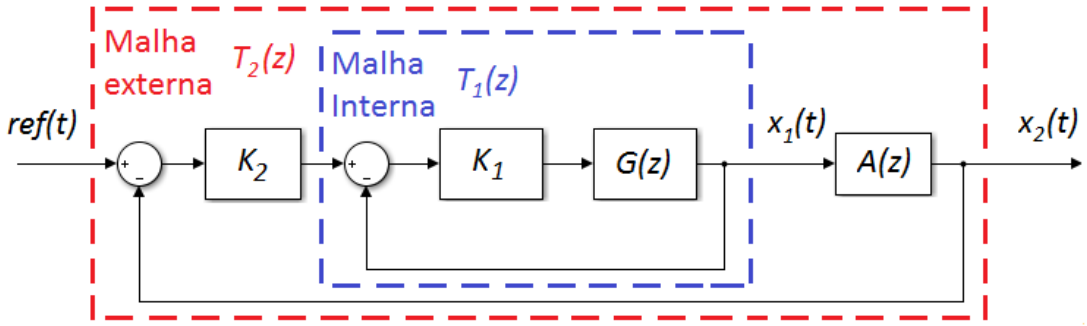

Figura 4. Malhas de controle implementadas no quadricóptero, sendo a "malha interna" o controle da velocidade angular e a "malha externa" o controle da posição angular.

Na Figura 4, $T_1(z)$ representa a função de transferência da malha interna (sem ruído), mostrada na equação (21), e $T_2(z)$ representa a função de transferência da malha externa (sem ruído), mostrada na equação (22). $G(z)$ representa a função de transferência referente a todo o processo de transformação da tensão aplicada aos motores, passando pela força que as hélices proporcionam, e resultando na velocidade angular. $A(z)$ é um tratamento de sinal que calcula a posição angular a partir da velocidade angular, que envolve integração do sinal e sua filtragem. Este tratamento de sinal não faz parte do escopo deste trabalho, portanto não será discutido. *ref(t)* é o sinal de referência de posição angular, em *rad*, $x_1(t)$ é a velocidade angular, em *rad/s*, e $x_2(t)$ é a posição angular, em *rad*.

$$T_1(z) = \frac{K_1 G(z)}{1 + K_1 G(z)} \tag{21}$$

$$T_2(z) = \frac{K_2 T_1(z) A(z)}{1 + K_2 T_1(z) A(z)} \tag{22}$$

3 DESCRIÇÃO DO EQUIPAMENTO E FORMA DE OPERAÇÃO

Este capítulo detalha o equipamento utilizado para a aplicação prática do projeto, incluindo especificações físicas do quadricóptero e especificações do *hardware* que permite o funcionamento deste, bem como a forma de operação e relação entre os componentes envolvidos.

O quadricóptero utilizado no trabalho possui hélices com dimensões de 10 x 4,5 polegadas, sendo que duas delas são construídas de forma a produzir aceleração que eleve o quadricóptero girando num sentido e as duas restantes girando no sentido contrário, equilibrando assim as forças que variam o ângulo *yaw*, como explicado anteriormente na seção **2.1** . A distância entre hélices opostas (1 – 4 e 2 – 3, segundo a Figura 1) é de 450mm.

No centro do quadricóptero há uma placa que contém os principais componentes que permitem a operação e controle do mesmo, incluindo um microcontrolador, sensores (um giroscópio e um acelerômetro) e um controlador eletrônico de velocidade, os quais serão explicados a seguir. Os motores estão situados junto das hélices, nas extremidades.

O microcontrolador utilizado é o Atmel ATmega328 (*8-bit*). Este modelo possui frequência máxima de operação de 20MHz, estando disponíveis dois *timers* de 8 *bits* e um de 16 *bits*, além do suporte a interrupções internas e externas, conversor analógico-digital e 6 canais de PWM. Em termos de memória disponível, possui 32 Kbytes de memória não-volátil auto-programável Flash, 2 Kbytes de SRAM (*Static Random-Access Memory* – Memória de Acesso Aleatório Estática) e 1 Kbyte de EEPROM (*Electrically Erasable Programmable Read-Only Memory* – Memória Somente de Leitura Programável Apagável Eletricamente). Maiores informações estão disponíveis em (Atmel, 2009). A família ATmega de microcontroladores é comumente utilizada no controle de quadricópteros, como exemplificado em (Hoffman, et al., 2007), (Sam, et al., 2013) e (Deshmukh, et al., 2015), principalmente pelo fato de ser a família que a plataforma Arduino utiliza.

O giroscópio tem a função de prover ao microcontrolador a informação de velocidade de rotação em relação a cada eixo (variação dos ângulos *roll*, *pitch* e *yaw*). O modelo utilizado é o InvenSense PS-ITG-3200, que internamente possui três conversores analógico-digitais de 16 *bits* para a digitalização de suas saídas, conforme (InvenSense, 2010), bem como um filtro passa-baixas com largura de banda configurável e uma interface I^2C de 400KHz, a qual é utilizada para o envio das informações que serão consumidas pelo sistema.

O acelerômetro, por sua vez, tem a função de prover ao microcontrolador a informação de aceleração exercida sobre cada um dos três eixos. O modelo utilizado é o Bosch BMA180, que possui internamente um conversor analógico-digital de 14 *bits*, filtros digitais configuráveis e interface I^2C para comunicação externa, conforme (Bosch, 2010).

A lógica de operação do sistema, que permite o vôo do quadricóptero de acordo com os componentes descritos, é a seguinte:

O usuário define as referências relativas à altitude e aos ângulos através de um controle remoto. Este envia por comunicação sem fio as informações dessas referências ao microcontrolador, por meio de sinais PWM com ciclos entre 1 e 2ms. O módulo de recepção sem fio está conectado a alguns dos pinos do microcontrolador, que é responsável pela conversão deste sinal em informação e sua subsequente interpretação. As referências dos ângulos *roll* e *pitch*, por exemplo, são as entradas para suas respectivas malhas de controle, ilustradas pela Figura 4.

Os sensores explicados anteriormente (giroscópio e acelerômetro) são responsáveis pela realimentação que é feita ao microcontrolador, referente a cada ângulo. O envio das informações de ambos, já digitalizadas, é feito via comunicação I^2C, permitindo ao microcontrolador o cálculo da ação de controle necessária para o seguimento das referências.

Esta ação de controle é transformada, efetivamente, em movimento do quadricóptero por meio da definição de uma tensão que é aplicada a cada motor. O microcontrolador possui saídas PWM, que estão conectadas a ESCs (*Electronic Speed Controllers* – Controladores Eletrônicos de Velocidade), os quais são ligados, por sua vez, aos motores. Assim, basta ao programa implementado no microcontrolador definir uma largura de pulso de PWM que equivalha, após as conversões necessárias, à tensão desejada (esta largura é calculada segundo a lógica de controle utilizada).

Os ESCs são circuitos elétricos que transformam o sinal de PWM recebido do microcontrolador, de acordo com a largura desse pulso, em uma tensão contínua equivalente que é aplicada ao motor, variando assim sua velocidade, como explicado em (De Villeneuve, 1982). No caso de motores trifásicos, para que o motor seja rotacionado, os enrolamentos do estator devem ser energizados de acordo com uma sequência, como explicado em (Microchip Technology Inc., 2003), a qual é executada da forma correta pelo ESC. O ESC utilizado neste quadricóptero possui corrente máxima de 20A.

Os motores possuem constante de velocidade de 1100 K_v, o que significa, neste caso, 1100 rpm/V. De acordo com (MICROMO, 2014), a força contra-eletromotriz gerada pelo motor V_e (dada em V) é diretamente proporcional à sua velocidade angular ω (dada em rpm), sendo esta proporção dada pela constante de força contra-eletromotriz K_e (dada em V/rpm, ou seja, o inverso da constante de velocidade), segundo a equação (23).

$$V_e = \omega \times K_e \quad \textbf{(23)}$$

O diagrama de blocos da Figura 5 mostra, de forma simplificada, a relação entre os componentes do sistema.

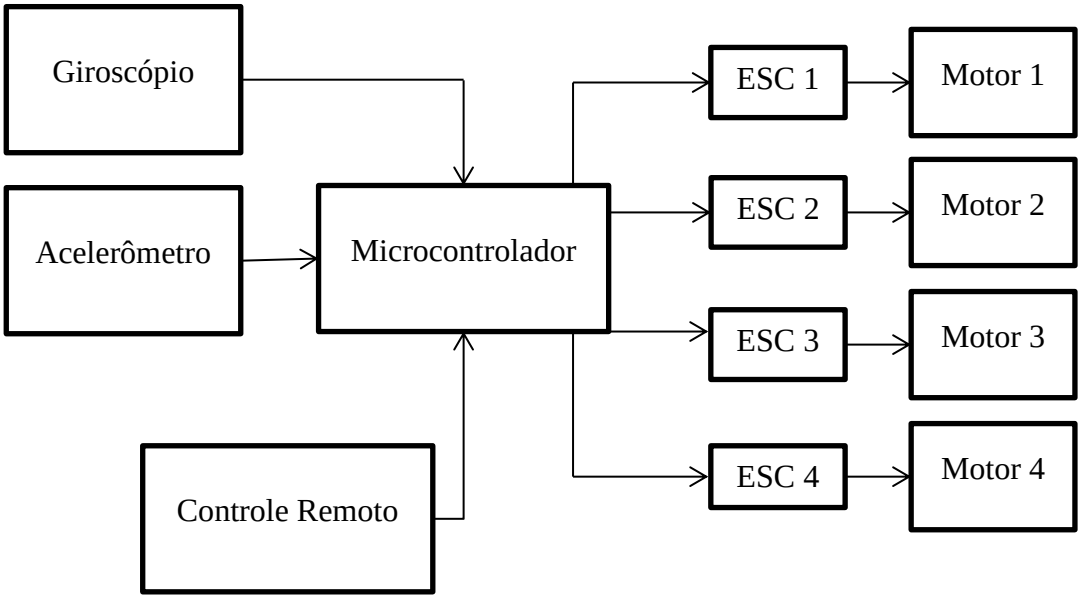

Figura 5. Diagrama de blocos resumido, mostrando a relação entrada/saída entre os componentes de *hardware* do sistema.

Além destes componentes, é utilizada uma bateria de polímero de lítio (LiPo) de 3 células com 3.7V, num total de 11.1V.

4 DESCRIÇÃO DO PROGRAMA IMPLEMENTADO NO MICROCONTROLADOR

Como já explicado, parte substancial do código já programado para este quadricóptero foi mantida, incluindo a lógica de leitura dos sensores e a lógica do controlador, por exemplo.

Por conta disso, e de forma a relacionar o *software* com o *hardware*, este capítulo explica, de forma resumida, as funções executadas pelo microcontrolador, de acordo com a sequência lógica e chamada de funções do código.

O programa roda, primeiramente, *main*, que é a função principal, porém bem simples. Ela apenas faz a chamada da função de configuração (*setup*) e, após o término desta, roda um *loop* infinito, detalhado mais à frente.

A configuração do microcontrolador se dá dentro da função *setup*, por meio da escrita em endereços de memória específicos. Primeiro, há a definição da direção dos pinos (entrada ou saída). Na sequência, a configuração dos *timers* e interrupções, incluindo a definição da frequência e dos modos destes, sendo que a interrupção ligada ao *Timer0* tem a função de incrementar uma variável *tempo*, e a cada 10ms fazer com que a parte principal do programa realmente rode. Esta lógica é que define o período de amostragem e de atualização dos motores. Por fim se dá a definição do *clock* da comunicação I^2C com os sensores e a subsequente chamada das funções de inicialização de cada sensor. Estas inicializações configuram, por exemplo, os filtros nas saídas de cada sensor.

Após a configuração inicial, o programa roda um *loop* infinito, onde a cada 10ms (segundo a interrupção do *Timer0*) se dá a leitura dos sensores e a atualização dos motores, além de determinadas funções que são executadas dependendo se o quadricóptero está “armado”, ou seja, em modo vôo, ou “desarmado”, ou seja, com os motores desligados.

A lógica da função *loop* é mostrada na Figura 6, através de um fluxograma.

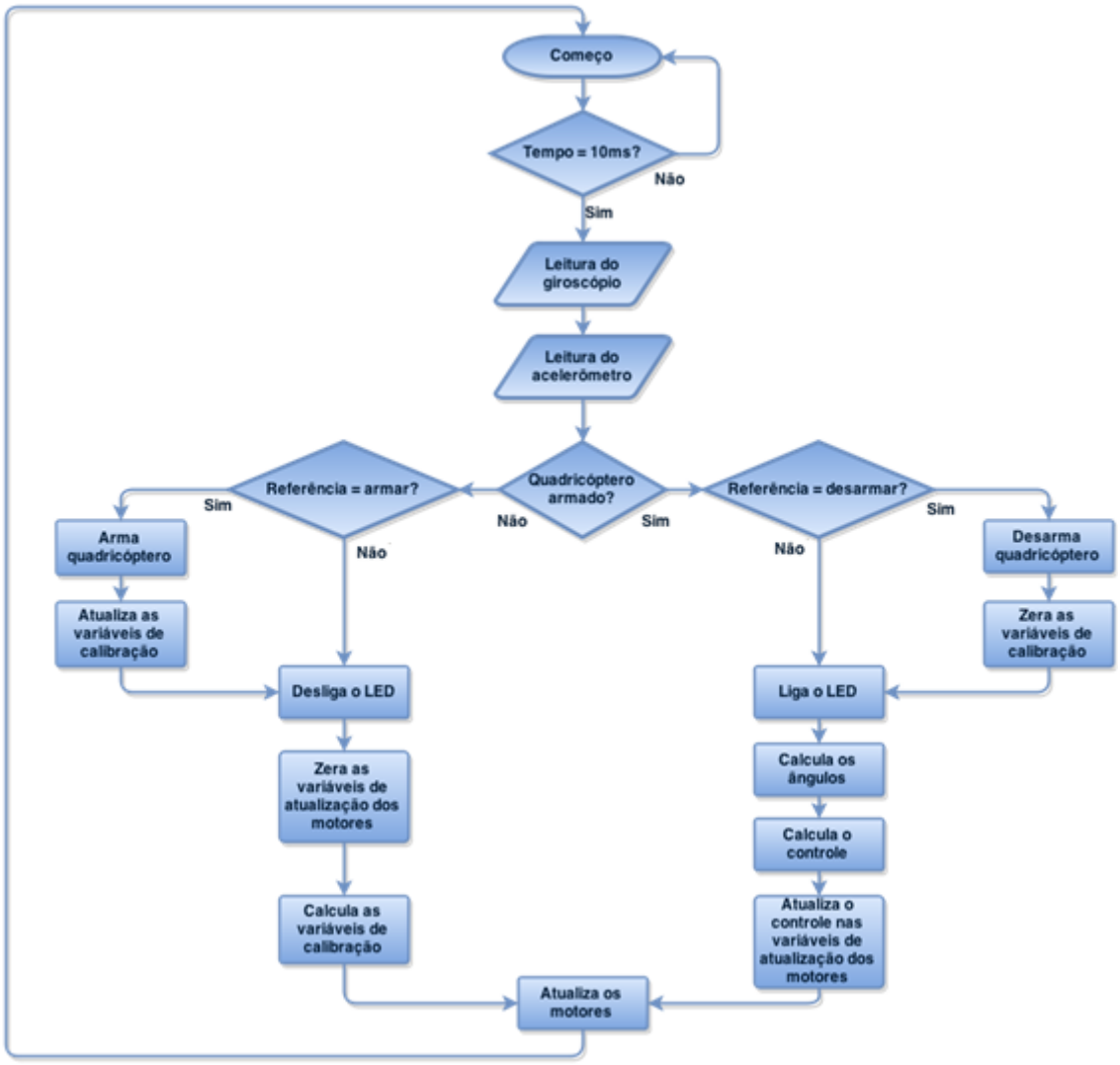

Figura 6. Fluxograma representando o *loop* principal do *software* do microcontrolador.

Como mostrado na Figura 6, no início do *loop* há um teste de tempo (este tempo, como mencionado anteriormente, está ligado a uma interrupção). Caso não tenham se passado 10ms desde a última execução, o programa volta ao início da função. Caso os 10ms já tenham passado, há a leitura dos sensores e o teste que verifica se o quadricóptero está “armado”.

No caso do quadricóptero não estar “armado”, um novo teste verifica se a referência lida do controle remoto (a leitura ocorre por meio de uma interrupção externa) define que o quadricóptero deve ser “armado”. Há uma referência específica que atende a condição, que significa uma posição definida do controle remoto para a qual o quadricóptero deve ser “armado”.

No caso do quadricóptero estar “armado”, outro teste verifica se a referência lida do controle remoto define que o quadricóptero deve ser “desarmado”. Há também uma referência específica que atende esta condição.

Além disso, alguns dos blocos da Figura 6 representam funções de grande importância no controle do quadricóptero. O bloco que faz o cálculo dos ângulos, por exemplo, se refere a uma função que faz esse cálculo baseado em entradas dos dois sensores. O bloco de cálculo do controle, por sua vez, tem como entradas os ângulos previamente calculados e as referências lidas do controle remoto, sendo que esta função está associada, para as malhas de *roll* e *pitch*, ao diagrama da Figura 4, e para a malha de controle de *yaw*, ao diagrama da Figura 7.

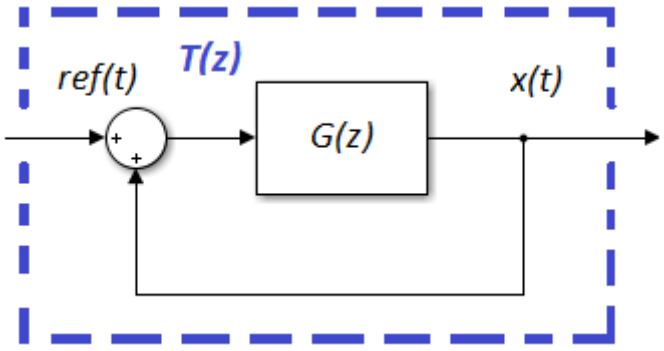

Figura 7. Malha de controle da velocidade angular do *yaw*.

Na Figura 7, *T(z)* representa a função de transferência da malha de controle da velocidade angular do *yaw* (sem ruído), mostrada na equação (24). *G(z)* representa a função de transferência referente a todo o processo de transformação da tensão aplicada aos motores, passando pela força que as hélices proporcionam, e resultando na velocidade angular. *ref(t)* é o sinal de referência de velocidade angular, em *rad/s*, e *x(t)* é a velocidade angular, em *rad/s*.

$$T(z) = \frac{G(z)}{1 + G(z)} \tag{24}$$

Novamente na Figura 6, o bloco que atualiza o controle nas variáveis dos motores realiza esta tarefa como segue (tomando como base a numeração dos motores na Figura 1):

Se a frente do quadricóptero for considerada o ponto médio entre os motores 1 e 2, a ação de controle (que pode ser negativa ou positiva, dependendo do sentido de rotação desejado) referente ao ângulo *roll* é somada aos motores 2 e 4 e subtraída dos motores 1 e 3, para que haja a variação deste ângulo no sentido desejado porém sem elevar ou diminuir a média das rotações dos quatro motores, sendo mantida assim a altitude.

A mesma lógica se dá para a ação de controle referente ao *pitch*, sendo, neste caso, somada aos motores 1 e 2 e subtraída dos motores 3 e 4. O mesmo ocorre para o *yaw*, sendo a ação de controle somada aos motores 1 e 4 e subtraída dos motores 2 e 3.

Além disso, uma referência de altitude do quadricóptero é somada diretamente aos quatro motores, sendo assim possível aumentar ou diminuir a média das rotações deles.

Por fim, ainda dentro da função *loop*, a Figura 6 possui um bloco que atualiza os motores. Este bloco se refere a uma função que utiliza os valores calculados para as variáveis referentes a cada motor e atualiza certos registradores nos *timers*, a fim de gerar um PWM para cada um desses valores.

Afora as funções explicadas, há duas rotinas de interrupção implementadas no código. Uma delas, como já explicado, é ativada pelo *Timer0* e tem a função de incrementar uma variável *tempo*. Esta rotina é executada a cada 10µs e, quando a variável *tempo* chega a 1000, ou seja, após 10ms, o programa roda sua parte principal (leitura dos sensores, atualização dos motores, etc.).

A segunda rotina de interrupção é classificada como uma interrupção externa, pois cada vez que um dos pinos que recebem as entradas do controle remoto varia seu estado esta parte do código é executada pelo programa. Dentro dela, as quatro variáveis que são utilizadas como referência para o controle são calculadas de acordo com o estado de cada pino, através de uma lógica que adquire o valor da largura de pulso dos PWMs de entrada.

Há também, no programa, lógicas para o envio de dados do sistema, como valor das referências, valor dos sensores e valor dos ângulos calculados, por meio de comunicação serial. Este código foi desenvolvido pelo Prof. Dr. Diego Eckhard e o arquivo principal se encontra no ANEXO A.

5 DESENVOLVIMENTO DO ALGORITMO IFT E SIMULAÇÕES

Foram desenvolvidas simulações no programa MATLAB, utilizando o método IFT, para fins de testes do comportamento do método e melhor compreensão do mesmo.

A função de transferência, neste caso, foi definida para cada um dos testes. Entretanto, do ponto de vista do algoritmo IFT, ela continua sendo desconhecida.

Na primeira simulação foi utilizado um controlador PI, com parâmetros ρ_1 e ρ_2, com a parametrização mostrada na equação (6). O número de iterações foi 1000, o número de amostras para cada experimento foi 100, o período de amostragem foi de 0,01 segundos, o tamanho do passo (γ) foi 0,001 e a referência utilizada no primeiro experimento de cada iteração foi uma onda quadrada de frequência 1Hz. É importante lembrar que, como já explicado, em cada iteração há dois experimentos, de onde são coletados os dados necessários para o cálculo dos novos parâmetros, ao fim de cada iteração.

O processo $G(z)$ foi definido conforme a equação (25), e a função de transferência de malha-fechada desejada $T_d(z)$ foi escolhida conforme a equação (26). Desta forma, o controlador desejado tem parâmetros $\rho_1 = 0{,}05$ e $\rho_2 = -0{,}015$.

$$G(z) = \frac{2}{z - 0{,}3} \tag{25}$$

$$T_d(z) = \frac{0{,}1}{z - 0{,}9} \tag{26}$$

Os parâmetros adquiridos com esta simulação foram $\rho_1 = 0{,}0509$ e $\rho_2 = -0{,}0152$, os quais têm sua evolução ao longo das iterações mostrada na Figura 8.

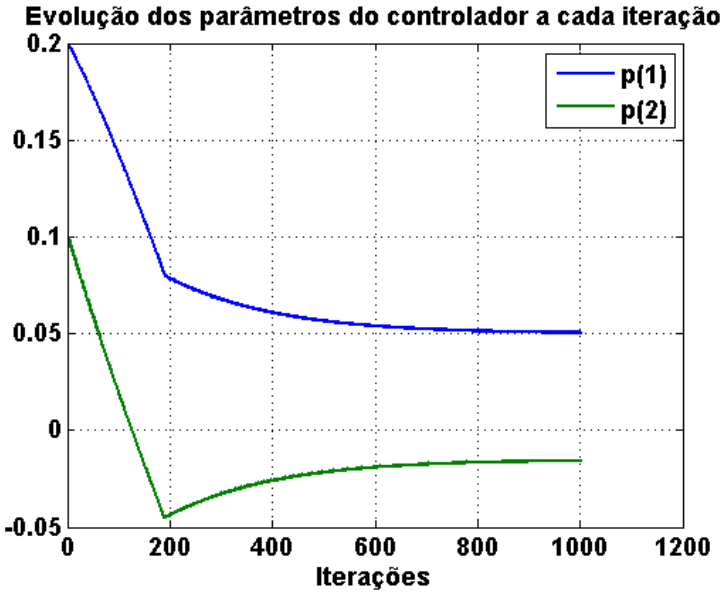

Figura 8. Evolução dos parâmetros do controlador a cada iteração, sendo ρ_1 a linha azul e ρ_2 a linha verde.

A evolução da função custo ao longo das iterações é mostrada na Figura 9.

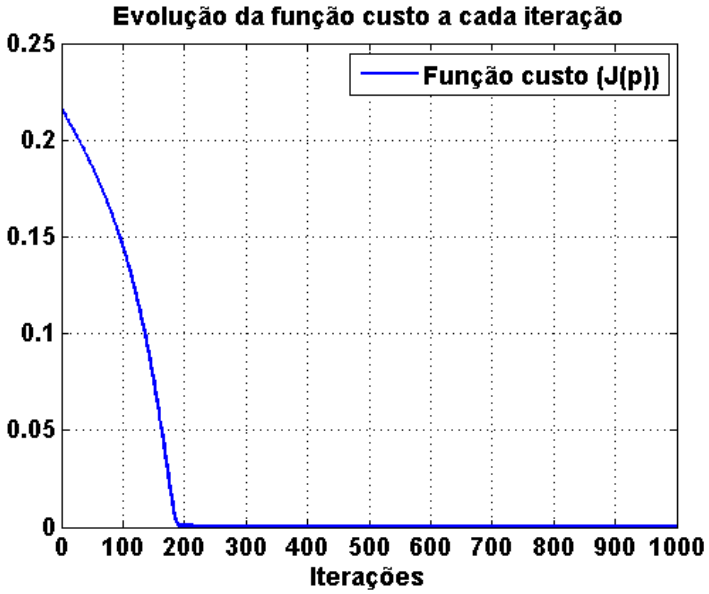

Figura 9. Evolução da função custo a cada iteração.

A função custo $J(\rho)$ calculada com as saídas desejada e real da Figura 10, representando a última iteração, é igual a $5{,}485x10^{-5}$.

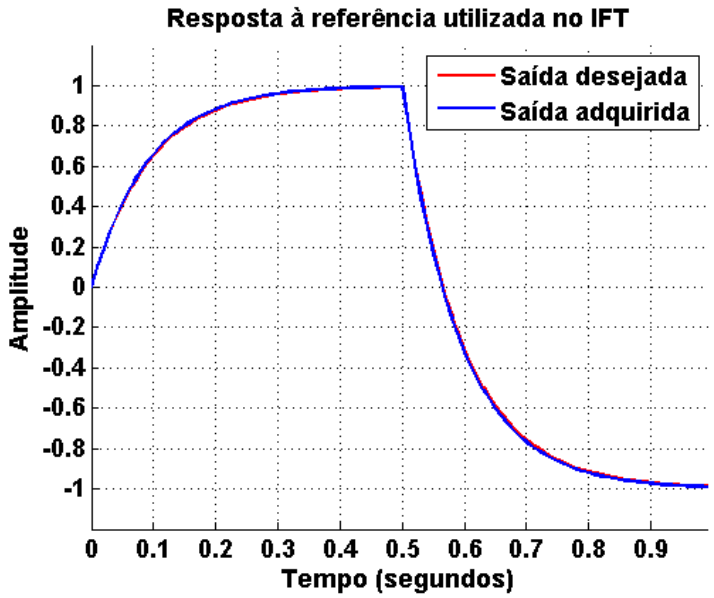

Figura 10. Resposta à referência utilizada no *loop* do IFT, sendo a saída desejada a linha vermelha e a saída adquirida a linha azul.

A função custo $J(\rho)$ calculada com as saídas desejada e real da Figura 11 é igual a $1{,}1152x10^{-5}$.

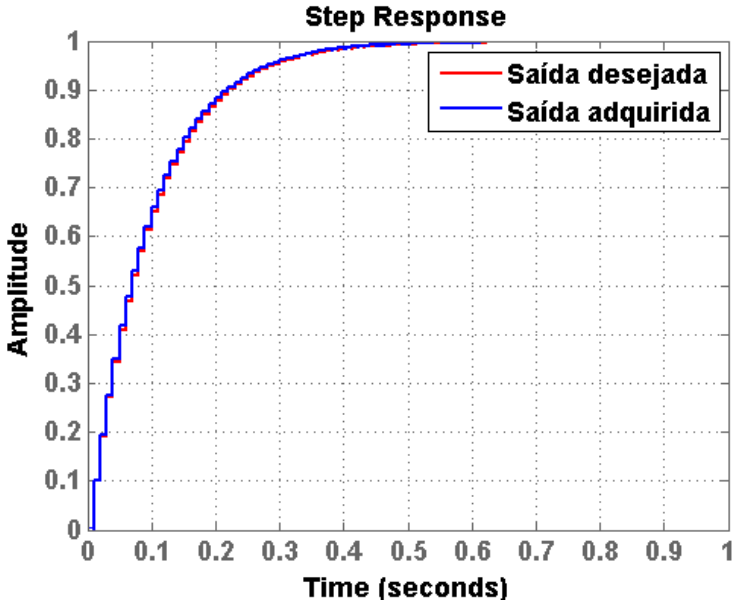

Figura 11. Resposta ao degrau da *T(z)* desejada, em vermelho, e da *T(z)* adquirida, em azul.

A segunda simulação foi feita com o mesmo "cenário" da primeira, ou seja, controlador PI segundo a equação (6), 1000 iterações, 100 amostras por experimento, período de amostragem de 0,01 segundos e tamanho de passo 0,001, sendo a referência uma onda quadrada de 1Hz. Entretanto, desta vez é aplicado um ruído branco (pseudoaleatório) de variância $1x10^{-4}$. Este ruído é aplicado na posição estipulada por *v(t)* na Figura 3, simulando um ruído na medição do sinal de saída.

O *G(z)* utilizado é mostrado na equação (25), e o *T(z)* desejado utilizado é mostrado na equação (26).

Os parâmetros adquiridos com esta simulação foram $\rho_1 = 0{,}051$ e $\rho_2 = -0{,}0152$, os quais têm sua evolução ao longo das iterações mostrada na Figura 12.

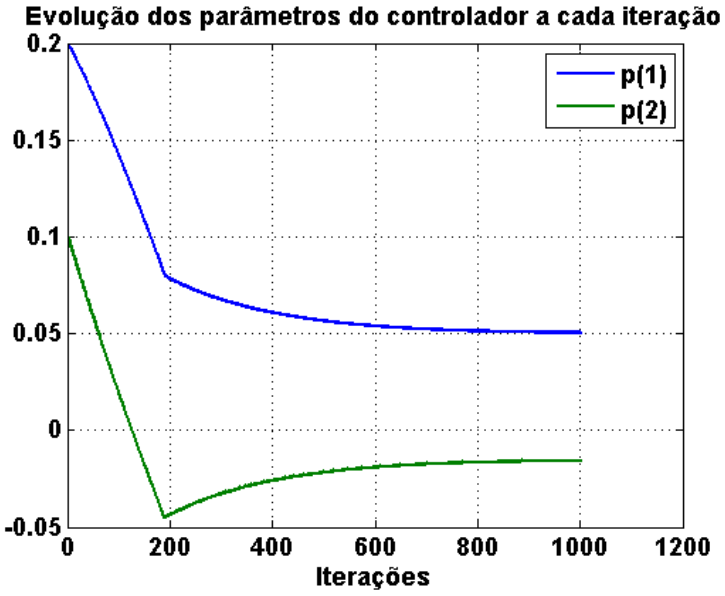

Figura 12. Evolução dos parâmetros do controlador a cada iteração, sendo ρ_1 a linha azul e ρ_2 a linha verde.

A evolução da função custo ao longo das iterações é mostrada na Figura 13.

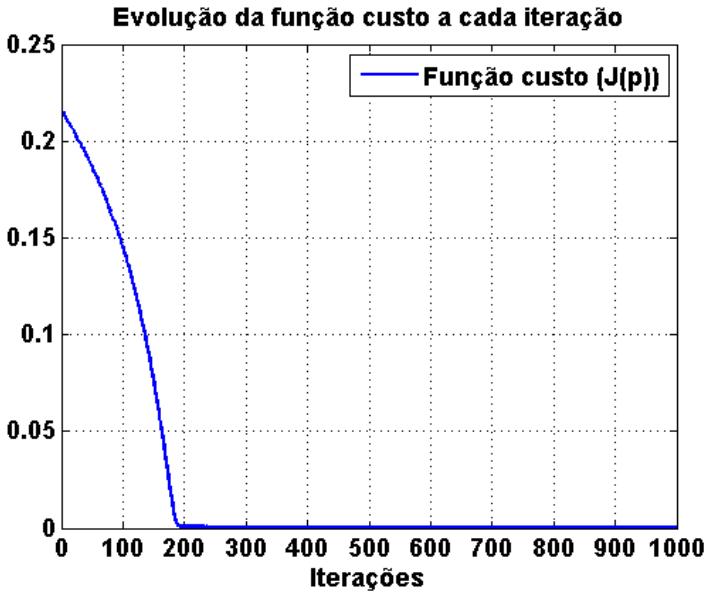

Figura 13. Evolução da função custo a cada iteração.

A função custo $J(\rho)$ calculada com as saídas desejada e real da Figura 14, representando a última iteração, é igual a $1{,}4404x10^{-4}$.

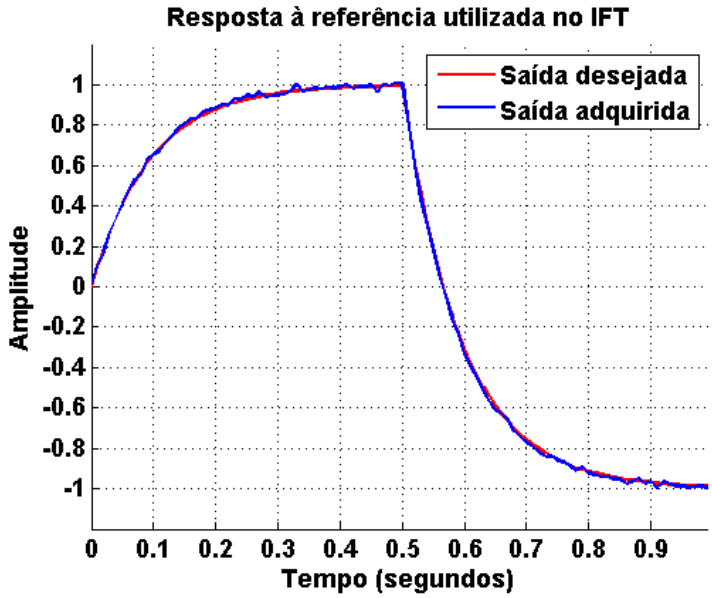

Figura 14. Resposta à referência utilizada no *loop* do IFT, sendo a saída desejada a linha vermelha e a saída adquirida a linha azul.

A função custo $J(\rho)$ calculada com as saídas desejada e real da Figura 15 é igual a $1{,}116x10^{-5}$.

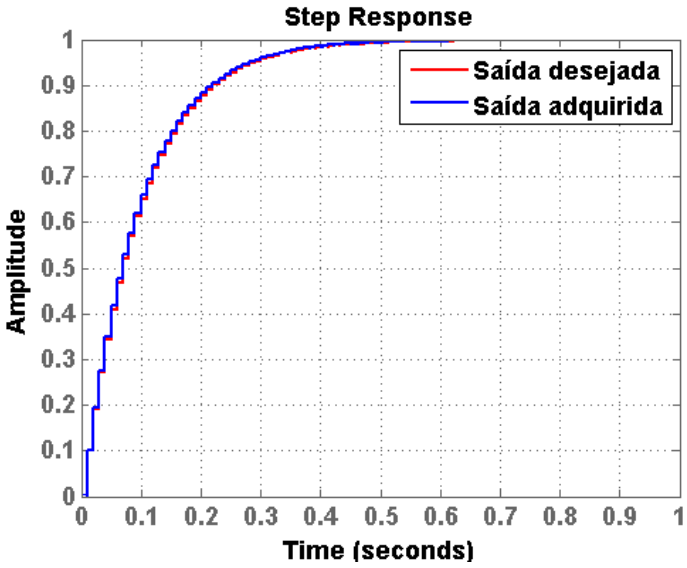

Figura 15. Resposta ao degrau da *T(z)* desejada, em vermelho, e da *T(z)* adquirida, em azul.

Analisando os gráficos gerados nas duas primeiras simulações, é possível concluir que o algoritmo chega numa solução aceitável em relação aos parâmetros ideais, baseado nas respostas à referência e funções custo resultantes. Além disso, o fato de haver ruído na segunda simulação não prejudica significativamente o processo de busca dos parâmetros, neste caso.

Além destes testes, uma vez que o foco para a aplicação real é o ajuste do controlador proporcional, como explicado anteriormente, foram também criadas simulações para este caso, trocando-se o *G(z)* para tornar o controle ideal possível, mantendo-se o *T(z)* desejado. O fato do controlador ideal ser possível é interessante para testes pois permite obter uma real estimativa do quão longe o controlador obtido está daquele, ou mesmo a que "velocidade" os parâmetros obtidos se aproximam dos ideais.

As simulações a seguir foram realizadas com a seguinte configuração: controlador do tipo proporcional, com parâmetro único ρ_1, parametrizado conforme a equação (5), o número de iterações foi 100, o número de amostras para cada experimento foi 100, o período de amostragem foi de 0,01 segundos e a referência utilizada no primeiro experimento de cada iteração foi uma onda quadrada de frequência 1Hz.

O processo *G(z)* foi definido conforme a equação (27), e a função de transferência de malha-fechada desejada $T_d(z)$ foi escolhida novamente conforme a equação (26). Desta forma, o controlador desejado tem parâmetro $\rho_1 = 0{,}5$.

$$G(z) = \frac{0{,}2}{z - 1} \qquad \textbf{(27)}$$

A simulação que segue possui tamanho de passo (γ) constante de 0,005 e é feita sem a presença de ruído.

O parâmetro adquirido com esta simulação foi $\rho_1 = 0{,}5$, o qual tem sua evolução ao longo das iterações mostrada na Figura 16.

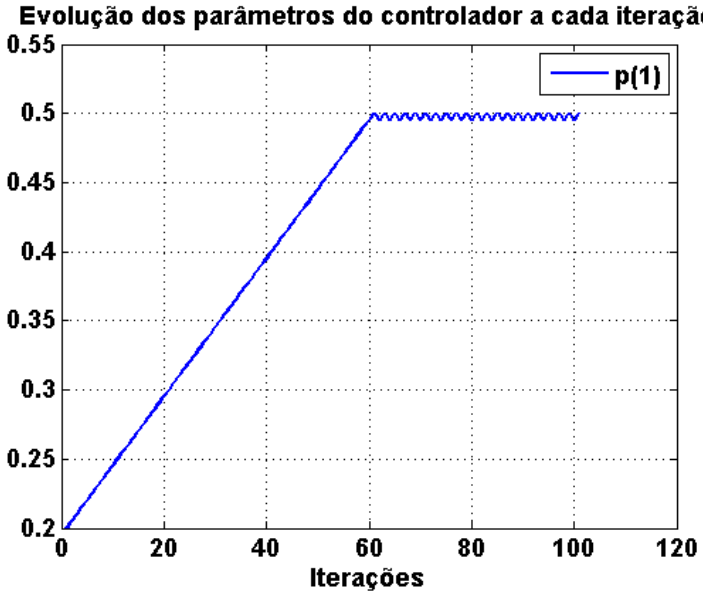

Figura 16. Evolução dos parâmetros do controlador a cada iteração, sendo ρ_1 a linha azul.

A evolução da função custo ao longo das iterações é mostrada na Figura 17.

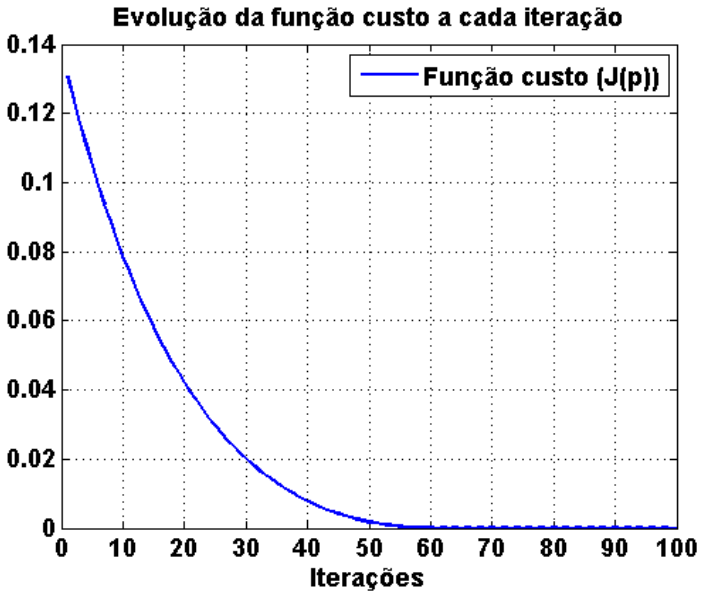

Figura 17. Evolução da função custo a cada iteração.

A função custo $J(\rho)$ calculada com as saídas desejada e real da Figura 18, representando a última iteração, é igual a $6{,}0626x10^{-32}$.

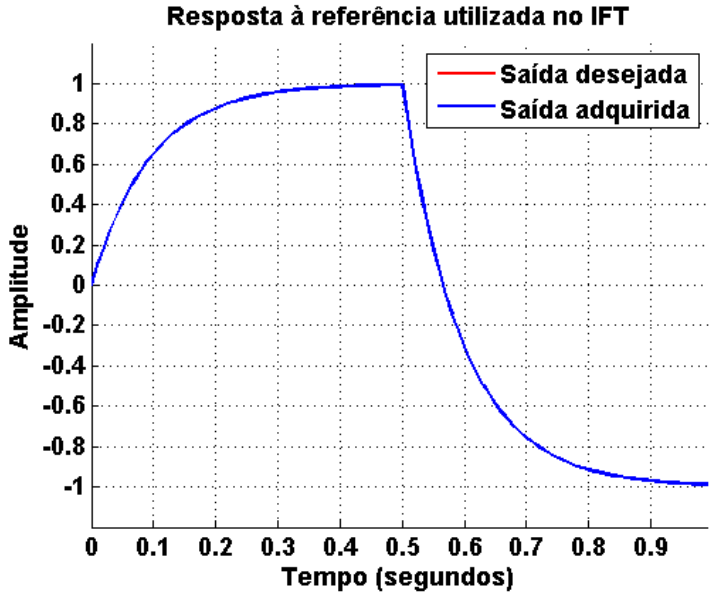

Figura 18. Resposta à referência utilizada no *loop* do IFT, sendo a saída desejada a linha vermelha e a saída adquirida a linha azul.

A próxima simulação possui tamanho de passo (γ) constante de 0,005 e é feita com a presença de ruído de variância $1x10^{-4}$.

O parâmetro adquirido com esta simulação foi $\rho_1 = 0{,}5$, o qual tem sua evolução ao longo das iterações mostrada na Figura 19.

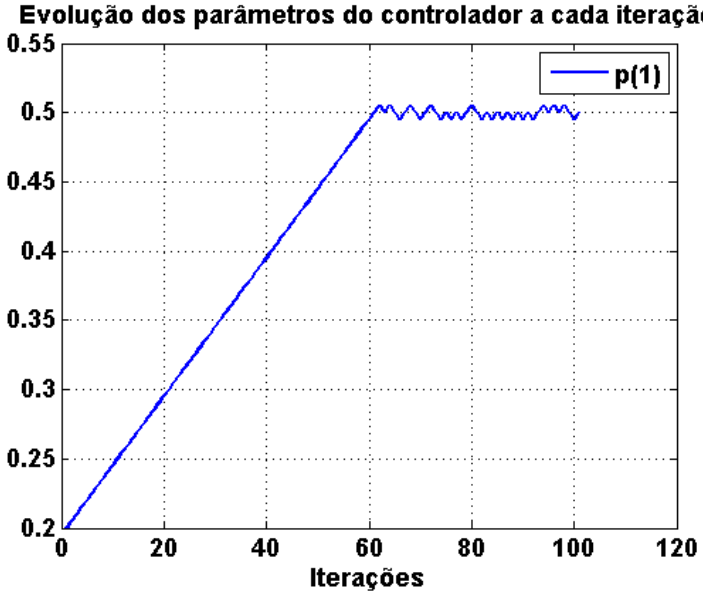

Figura 19. Evolução dos parâmetros do controlador a cada iteração, sendo ρ_1 a linha azul.

A evolução da função custo ao longo das iterações é mostrada na Figura 20.

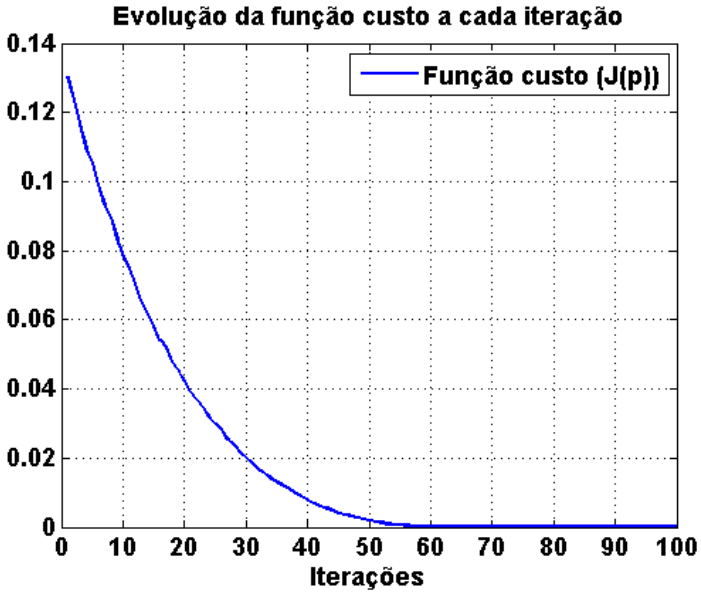

Figura 20. Evolução da função custo a cada iteração.

A função custo $J(\rho)$ calculada com as saídas desejada e real da Figura 21, representando a última iteração, é igual a $8{,}6467x10^{-5}$.

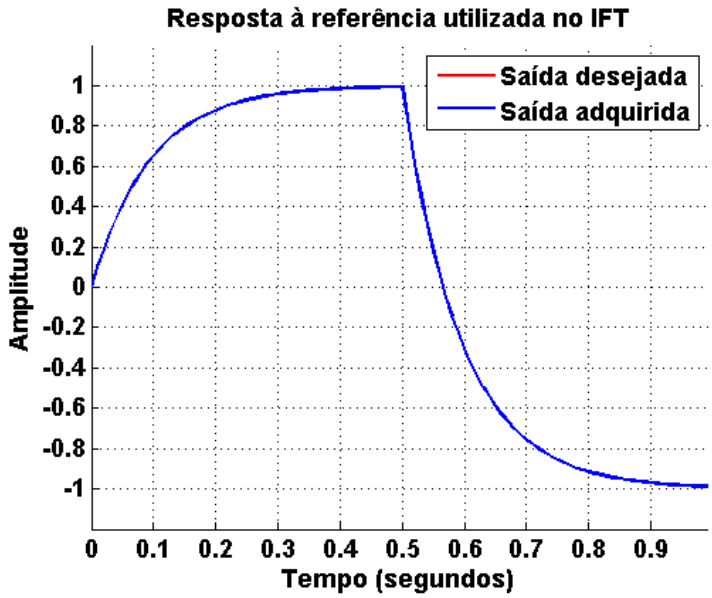

Figura 21. Resposta à referência utilizada no *loop* do IFT, sendo a saída desejada a linha vermelha e a saída adquirida a linha azul.

Novamente, a presença de ruído não afetou significativamente os resultados obtidos nos testes.

Para os próximos testes, uma nova abordagem surge em relação ao tamanho do passo, que passa a poder variar a cada iteração. Uma estratégia possível é diminuir pela metade o tamanho do passo sempre que o custo na iteração presente for maior que o custo na iteração imediatamente anterior, ou seja, utilizando a equação (28). Esta forma de variação de passo é explicada em (Bazanella, et al., 2012).

$$\gamma_i = \frac{\gamma_i}{2}, \qquad se\ J_i(\rho) > J_{i-1}(\rho) \tag{28}$$

Esta variação de tamanho de passo tem vantagem em relação ao passo constante no sentido de que sempre que a função custo ultrapassar seu mínimo (estando, neste caso, suficientemente próxima deste mínimo) o tamanho do passo diminuirá, tornando possível um ajuste mais fino dos parâmetros do controlador e, consequentemente, melhor desempenho.

Entretanto, uma variante desta abordagem será aplicada na simulação a seguir. Para este caso, a configuração é a mesma da simulação anterior, incluindo o ruído no sistema, mas o passo inicial é definido com tamanho 0,05 e os passos seguintes são calculados da seguinte forma: uma variável auxiliar *(step_factor)* é iniciada com valor 1, sendo que cada vez que o custo na iteração presente for maior que o custo na iteração imediatamente anterior esta variável será incrementada, ou seja, como na equação (29). O tamanho de passo para a iteração presente é então calculado segundo a equação (30).

$$step_{factor} = step_{factor} + 1, \qquad se\ J_i(\rho) > J_{i-1}(\rho) \tag{29}$$

$$\gamma_i = \frac{\gamma_1}{step_{factor}} \tag{30}$$

Desta forma, o tamanho de passo é reduzido de forma gradual e não muito abrupta, possibilitando melhorias significativas na evolução dos parâmetros do controlador, principalmente nas iterações iniciais.

O parâmetro adquirido com esta simulação foi $\rho_1 = 0{,}499$, o qual tem sua evolução ao longo das iterações mostrada na Figura 22.

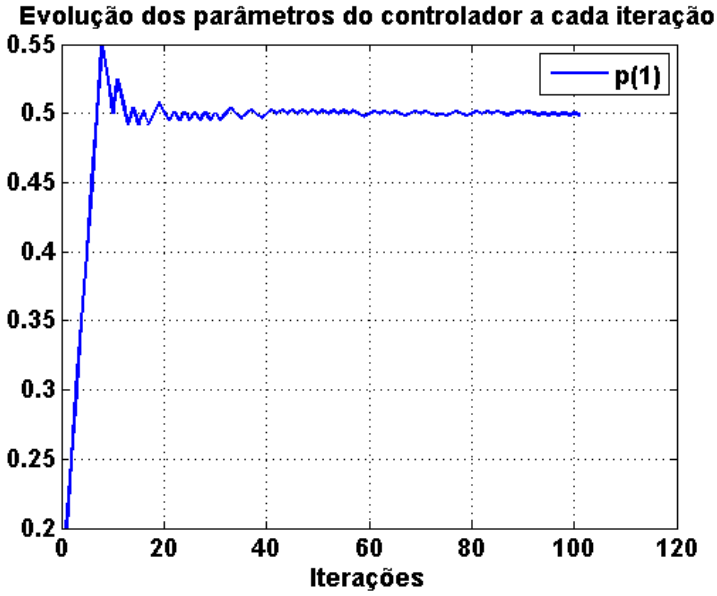

Figura 22. Evolução dos parâmetros do controlador a cada iteração, sendo ρ_1 a linha azul.

A evolução da função custo ao longo das iterações é mostrada na Figura 23.

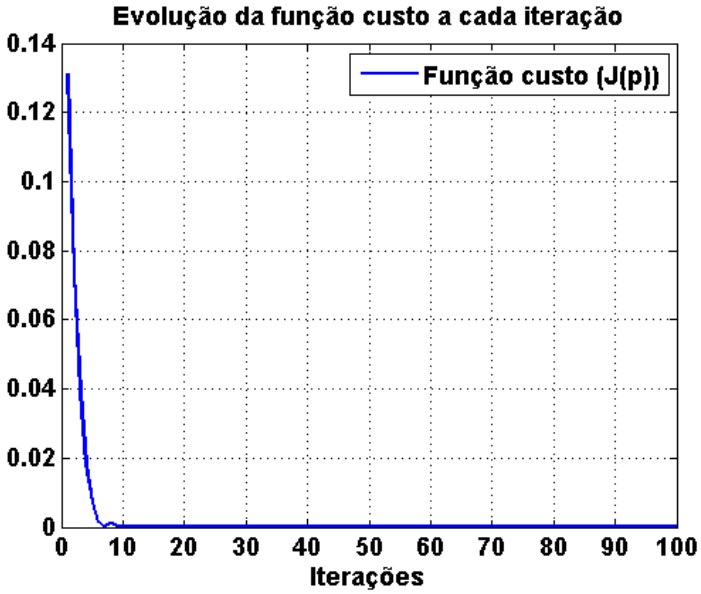

Figura 23. Evolução da função custo a cada iteração.

A função custo $J(\rho)$ calculada com as saídas desejada e real da Figura 24, representando a última iteração, é igual a $9{,}5758x10^{-5}$.

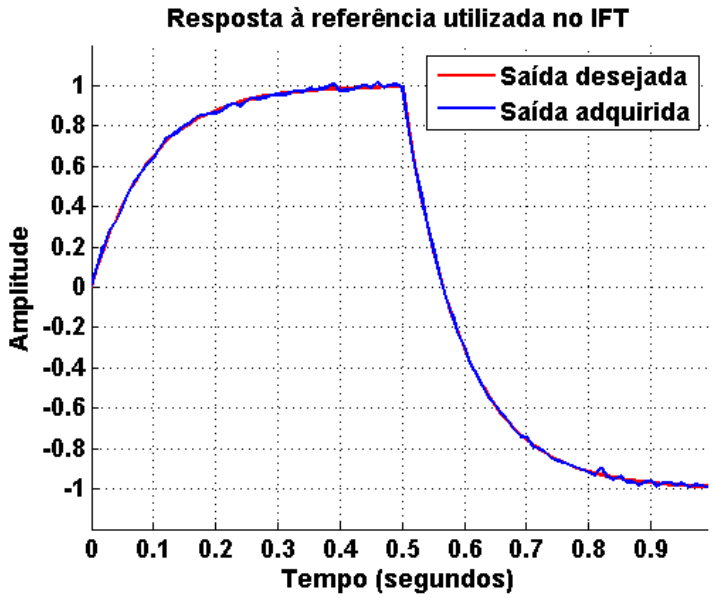

Figura 24. Resposta à referência utilizada no *loop* do IFT, sendo a saída desejada a linha vermelha e a saída adquirida a linha azul.

Os gráficos resultantes desta simulação mostram um comportamento interessante em função da forma de variação do tamanho do passo utilizada. Embora o custo na iteração final tenha sido levemente superior em relação à simulação anterior, a evolução, tanto do parâmetro $\rho_1 = 0{,}499$ quanto da função custo, foi significativamente mais rápida, ou seja, foram necessárias menos iterações para se chegar num desempenho aceitável do sistema.

O arquivo das simulações, o qual possui pequenas variações para cada uma delas (mostradas em comentários no arquivo), se encontra no ANEXO B.

6 SIMULAÇÕES NUM MODELO TEÓRICO DE QUADRICÓPTERO

Foram realizadas simulações adicionais no programa MATLAB, desta vez utilizando um modelo de simulação existente de um quadricóptero, desenvolvido no Simulink conforme (Bouabdallah, et al., 2004) e disponibilizado para este trabalho pelo Eng. Douglas Tesch.

O método utilizado para a programação do algoritmo e para a configuração de cada simulação foi, dentro do possível, o mesmo do capítulo anterior, porém neste foram incluídas algumas inicializações de variáveis necessárias para executar o modelo, bem como algumas adaptações ao código para o bom funcionamento dos testes. Além disso, ao contrário da abordagem prática (implementação no quadricóptero), o controle, nestas simulações, foi baseado numa referência para a velocidade angular, não para a posição angular.

Assim como no restante dos testes realizados neste projeto, o ângulo utilizado é o *roll*.

A simulação a seguir foi realizada com a seguinte configuração: controlador do tipo proporcional, com parâmetro único ρ_1, parametrizado conforme a equação (5), o número de iterações foi 50, o número de amostras para cada experimento foi 131, o período de amostragem foi de 0,01 segundos e a referência utilizada no primeiro experimento de cada iteração foi uma onda quadrada de frequência 1Hz. O tamanho de passo (γ) é constante, de valor 0,01.

O processo $G(z)$ é desconhecido, uma vez que se está utilizando o modelo do quadricóptero no Simulink, e a função de transferência de malha-fechada desejada $T_d(z)$ foi escolhida de acordo com a equação (31). O controlador ideal não é conhecido, mas é esperado que o algoritmo calcule o parâmetro ρ_1 que minimize a função custo.

$$T_d(z) = \frac{0{,}2}{z - 0{,}8} \tag{31}$$

O parâmetro adquirido com esta simulação foi $\rho_1 = 0{,}32$, o qual tem sua evolução ao longo das iterações mostrada na Figura 25.

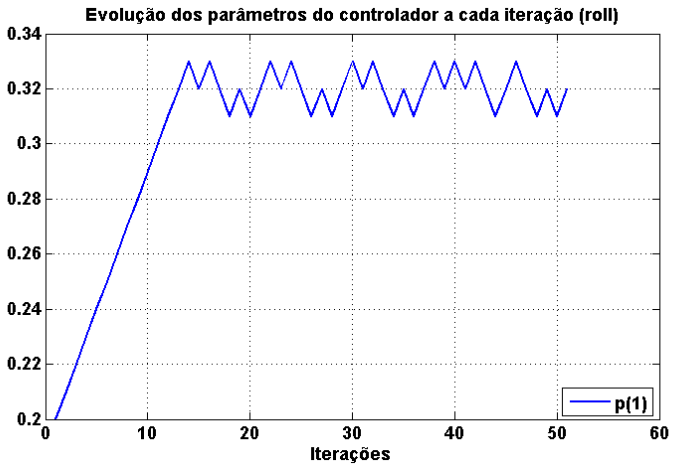

Figura 25. Evolução dos parâmetros do controlador a cada iteração, sendo ρ_1 a linha azul.

A evolução da função custo ao longo das iterações é mostrada na Figura 26.

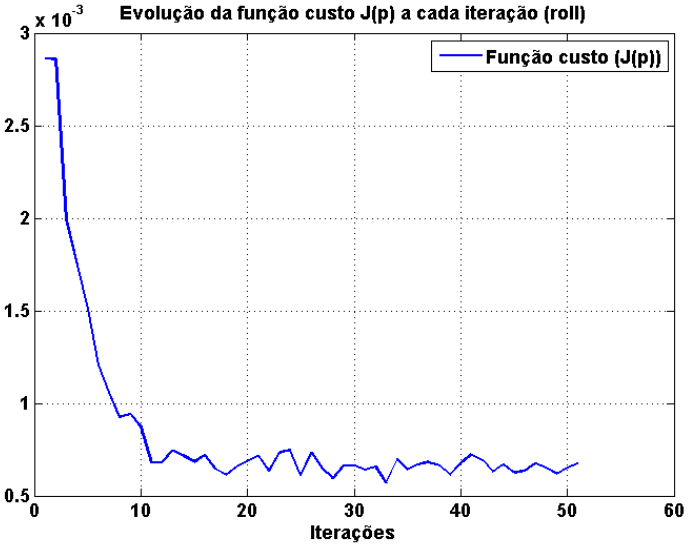

Figura 26. Evolução da função custo a cada iteração.

A função custo $J(\rho)$ calculada com as saídas desejada e real da Figura 27, representando a última iteração, é igual a $7{,}47x10^{-4}$.

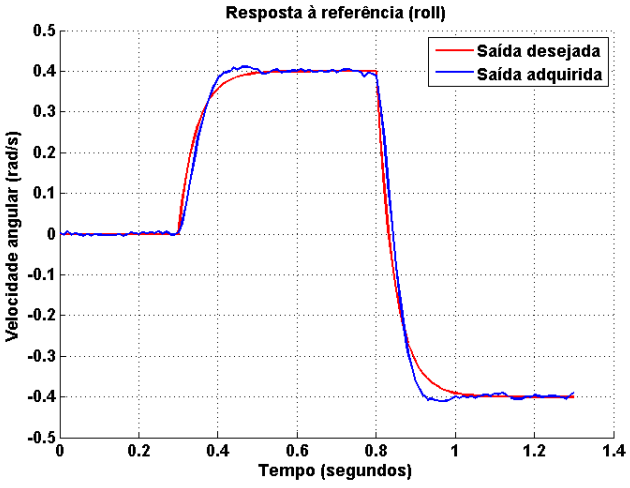

Figura 27. Resposta à referência utilizada no *loop* do IFT, sendo a saída desejada a linha vermelha e a saída adquirida a linha azul.

As duas simulações que seguem foram realizadas com a seguinte configuração: controlador do tipo proporcional, com parâmetro único ρ_1, parametrizado conforme a equação (5), o número de iterações foi 100, o número de amostras para cada experimento foi 531, o período de amostragem foi de 0,01 segundos e a referência utilizada no primeiro experimento de cada iteração está representada na Figura 28. O tamanho de passo (γ) é constante, de valor 0,01.

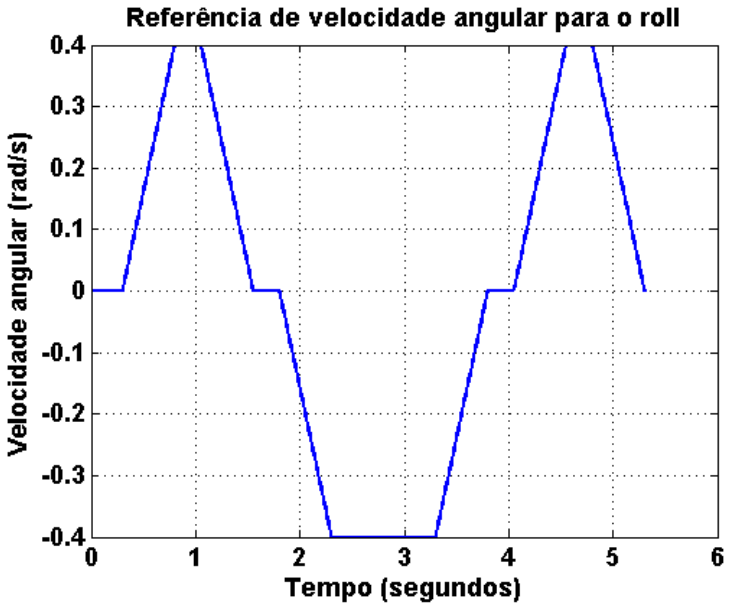

Figura 28. Referência de velocidade angular em rad/s.

A função de transferência de malha-fechada desejada $T_d(z)$ foi escolhida de acordo com a equação (31), da mesma forma que anteriormente.

A simulação mostrada na sequência tem seu parâmetro iniciando em $\rho_1 = 0{,}2$. O parâmetro adquirido após a simulação foi $\rho_1 = 0{,}32$, o qual tem sua evolução ao longo das iterações mostrada na Figura 29.

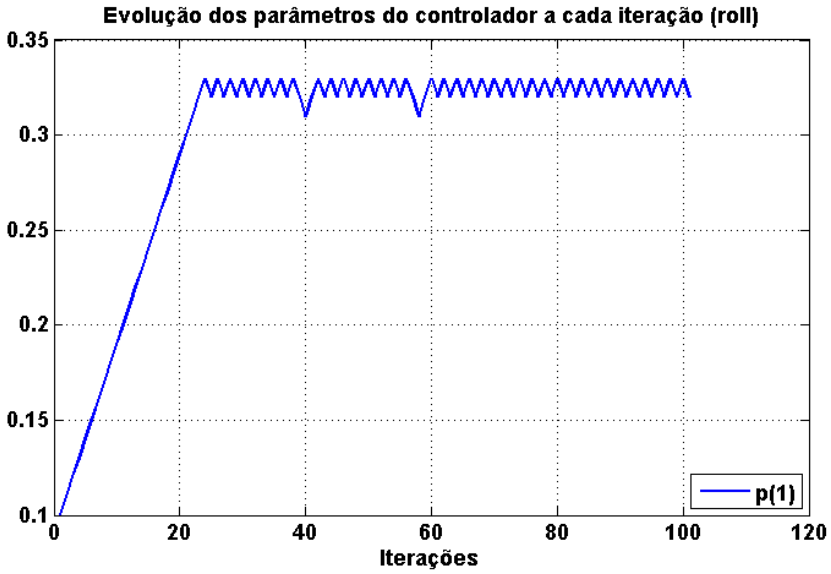

Figura 29. Evolução dos parâmetros do controlador a cada iteração, sendo ρ_1 a linha azul.

A evolução da função custo ao longo das iterações é mostrada na Figura 30.

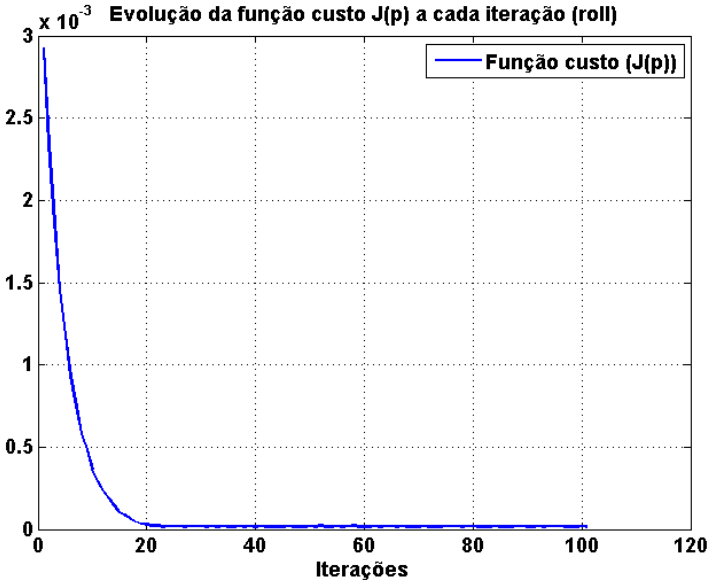

Figura 30. Evolução da função custo a cada iteração.

A função custo $J(\rho)$ calculada com as saídas desejada e real da Figura 31, representando a última iteração, é igual a $1{,}91x10^{-5}$.

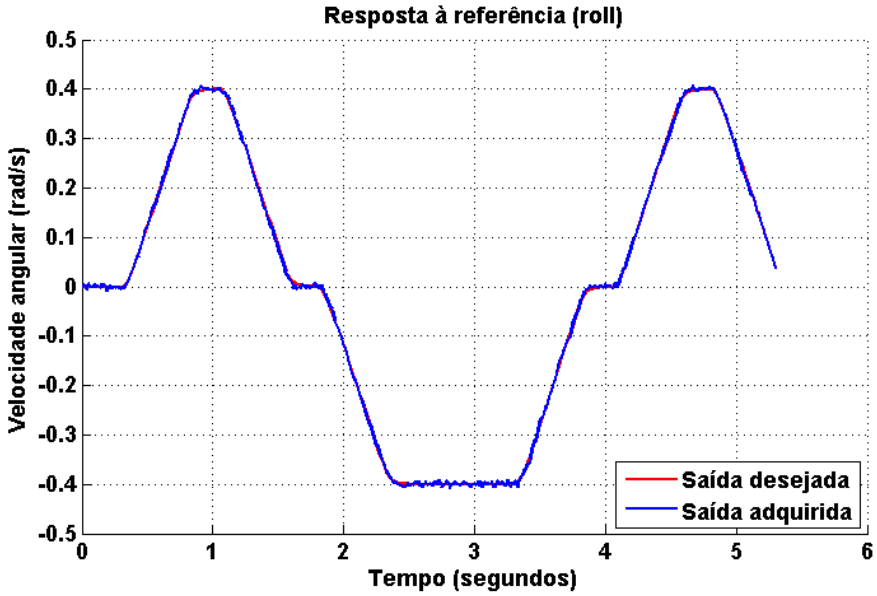

Figura 31. Resposta à referência utilizada no *loop* do IFT, sendo a saída desejada a linha vermelha e a saída adquirida a linha azul.

A próxima simulação tem a mesma configuração da anterior, porém seu parâmetro inicia em $\rho_1 = 1$. O parâmetro adquirido após a simulação foi $\rho_1 = 0{,}32$, o mesmo obtido anteriormente, e que tem sua evolução ao longo das iterações mostrada na Figura 32.

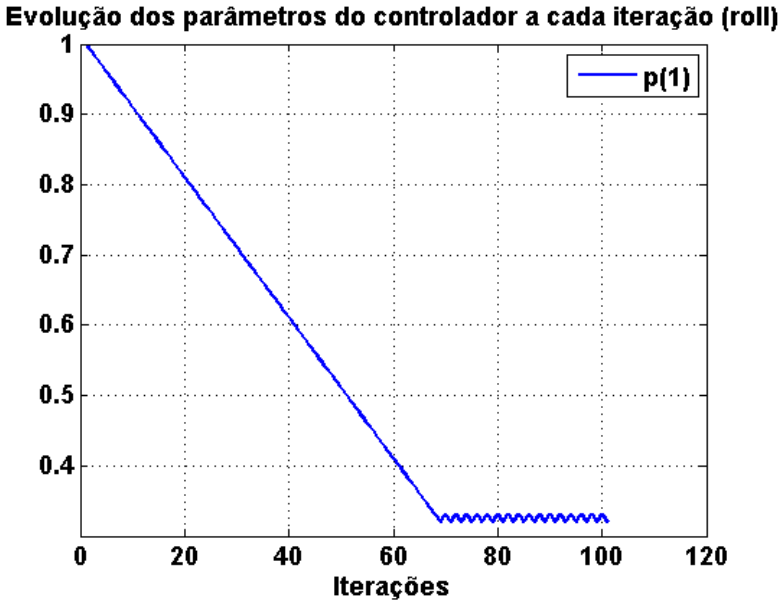

Figura 32. Evolução dos parâmetros do controlador a cada iteração, sendo ρ_1 a linha azul.

A evolução da função custo ao longo das iterações é mostrada na Figura 33.

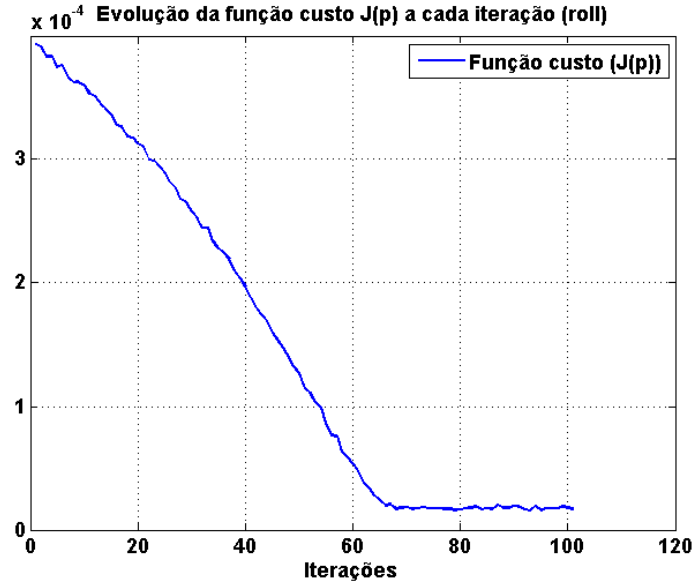

Figura 33. Evolução da função custo a cada iteração.

A função custo $J(\rho)$ calculada com as saídas desejada e real da Figura 34, representando a última iteração, é igual a $1{,}87x10^{-5}$.

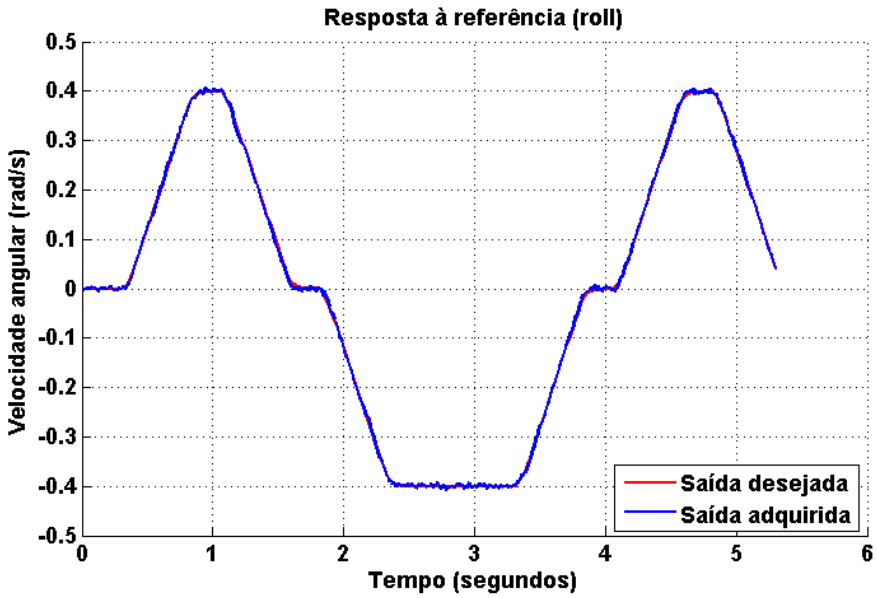

Figura 34. Resposta à referência utilizada no *loop* do IFT, sendo a saída desejada a linha vermelha e a saída adquirida a linha azul.

A última simulação deste capítulo tem o objetivo de testar, no modelo de quadricóptero, o algoritmo utilizado na última simulação do capítulo anterior, a qual introduziu uma redução gradual do tamanho de passo, através da variável auxiliar *step_factor*, incluída nas equações (29) e (30). Isto permite o aumento do passo inicial, neste caso para $\gamma = 0{,}1$. À exceção disto, a configuração utilizada foi a mesma das duas simulações anteriores.

O parâmetro adquirido nesta simulação foi $\rho_1 = 0{,}3255$, o qual tem sua evolução ao longo das iterações mostrada na Figura 35.

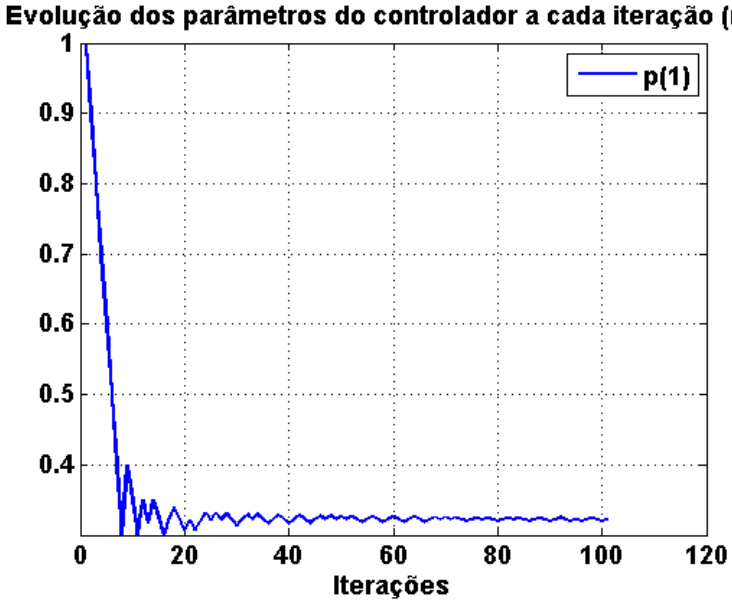

Figura 35. Evolução dos parâmetros do controlador a cada iteração, sendo ρ_1 a linha azul.

A evolução da função custo ao longo das iterações é mostrada na Figura 36.

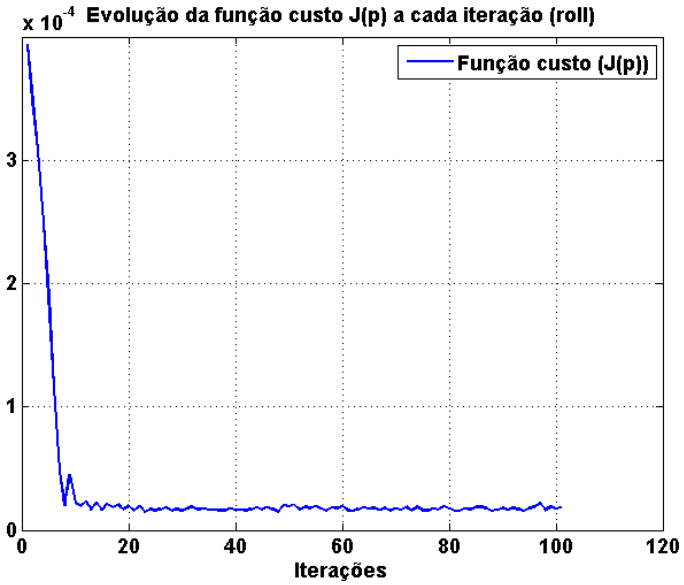

Figura 36. Evolução da função custo a cada iteração.

A função custo $J(\rho)$ calculada com as saídas desejada e real da Figura 37, representando a última iteração, é igual a $1{,}73x10^{-5}$.

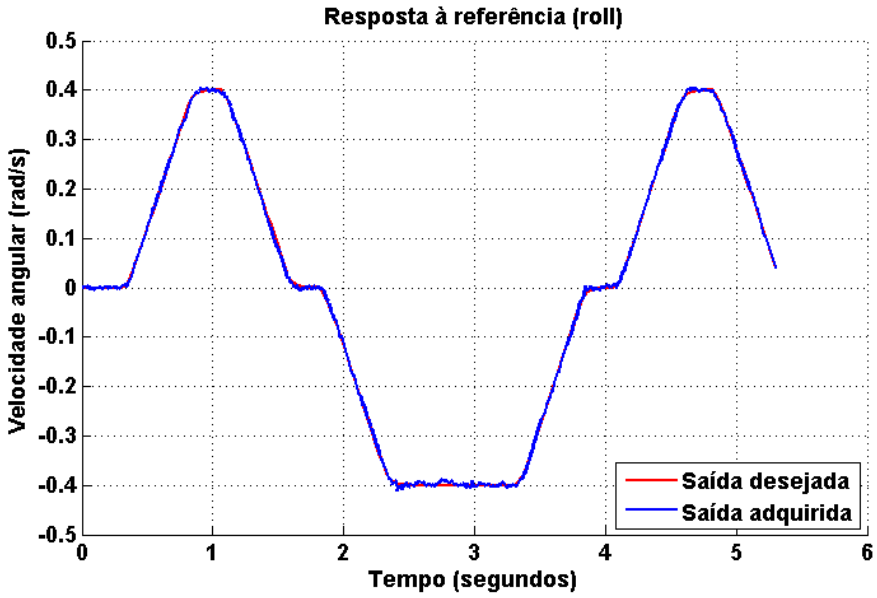

Figura 37. Resposta à referência utilizada no *loop* do IFT, sendo a saída desejada a linha vermelha e a saída adquirida a linha azul.

É notável, desta forma, a evolução mais rápida do parâmetro nos passo ρ_1 ao longo das iterações iniciais, assim como a redução da função custo $J(\rho)$. Nos passos finais, entretanto, o tamanho do passo é reduzido, permitindo uma aproximação mais precisa ao parâmetro ideal.

O código de MATLAB utilizado nas simulações deste capítulo se encontra no ANEXO C, ao qual pequenas variações são efetuadas para cada um dos diferentes testes.

7 IMPLEMENTAÇÃO DO IFT AO QUADRICÓPTERO E RESULTADOS

Para que fosse possível aplicar o método estudado neste trabalho ao quadricóptero, o código explicado no capítulo 4 foi modificado de forma a incluir um modo de ajuste, no qual o funcionamento é o mesmo em relação à obtenção dos ângulos e às malhas de controle, por exemplo, mas as referências não são obtidas do controle remoto, e sim pré-programadas. Além disso, há a inclusão da lógica que calcula o novo parâmetro do controlador baseado nos dois experimentos que são executados a cada iteração do algoritmo.

A programação foi feita com o *Win-AVR*, que é um conjunto de ferramentas de desenvolvimento de *software* para a série de microcontroladores Atmel AVR. O compilador utilizado é o AVR GNU Compiler Collection (GCC) 4.3.3. A gravação do código no microcontrolador (Atmel ATmega328) foi realizada através de comunicação serial com o computador.

Pelo fato do controle do quadricóptero já estar sendo feito separadamente para os ângulos *roll*, *pitch* e *yaw*, um deles, o *roll*, foi escolhido para ser utilizado nos testes deste projeto. Para permitir que isto fosse feito, foi necessário fixar um dos eixos de forma que só esse ângulo pudesse variar, amarrando duas extremidades opostas (e desativando os motores correspondentes às hélices destas extremidades), dentro de uma caixa que estava disponível no laboratório.

Além disso, desta forma foi possível realizar os testes com mais segurança (seria inconveniente que o quadricóptero voasse de fato dentro do laboratório) e conectar o quadricóptero ao computador, aproveitando a comunicação serial já citada para enviar comandos e receber dados relevantes.

Como já mostrado na seção 2.4 , para o ângulo em questão, há uma malha interna que controla sua velocidade angular e uma malha externa que controla sua posição angular. O controlador utilizado para cada uma destas malhas é do tipo proporcional, sendo que para a malha interna o ganho é fixado com valor *0,3*, que já era utilizado no quadricóptero, e para a malha externa o ganho é recalculado de acordo com o algoritmo IFT, através da atualização do parâmetro ρ_1 (equação (5)) a cada iteração.

A forma de operação neste modo de ajuste foi concebida na programação da seguinte maneira: ao receber um comando específico pela serial, o quadricóptero é "armado" e começa a receber uma referência pré-programada relativa ao primeiro experimento do algoritmo, sendo que a cada amostra a saída (ângulo) é gravada numa variável para uso posterior. Ao fim deste primeiro experimento, o quadricóptero é forçado a retornar à sua posição inicial (ângulo 0), para assim começar o segundo

experimento. Neste, a referência utilizada é o erro (como já explicado anteriormente e expresso na equação (15)). Além disso, ao longo deste segundo experimento, é calculado o $\frac{\partial y(t,\rho)}{\partial \rho}$ para cada amostra, conforme a equação (17), e calculado o $\frac{\partial J_T(\rho)}{\partial \rho}$ segundo a equação (13) (um fator do somatório é calculado a cada amostra e somado a uma variável, para que ao final do experimento o resultado de $\frac{\partial J_T(\rho)}{\partial \rho}$ seja obtido). Após esta sequência, de posse dos valores previamente calculados, um novo parâmetro para o controlador é obtido de acordo com a equação (12). O código que implementa este algoritmo se encontra no ANEXO D.

É importante ressaltar que, como anteriormente, a parte principal do código é executada a cada 10ms, sendo este tempo considerado o período de amostragem do sistema. Além disso, a saída desejada é também pré-programada no microcontrolador para cada referência, sendo que a função de transferência de malha-fechada $T_d(z)$ (desejada para o sistema) utilizada para o cálculo desta saída é expressa na equação (32).

$$T_d(z) = \frac{0{,}1}{z - 0{,}9} \quad \textbf{(32)}$$

Vale lembrar também que o processo $G(z)$, como anteriormente, é desconhecido, sendo esta uma das principais características do método IFT. Sendo assim, o algoritmo visa à minimização da função custo, como explicado nos capítulos anteriores.

O primeiro teste, que tem seus resultados mostrados a seguir, foi feito com uma referência angular do tipo salto (saindo de 0º para 20º). O número de amostras para cada experimento foi 300. O passo inicial (γ) teve valor 0,2, sendo que era possível alterá-lo via comunicação serial, o que foi feito após algumas iterações, sendo este reduzido para 0,1. Foram feitas 20 iterações neste teste.

O parâmetro adquirido neste teste foi $\rho_1 = 1{,}9$, o qual tem sua evolução ao longo das iterações mostrada na Figura 38.

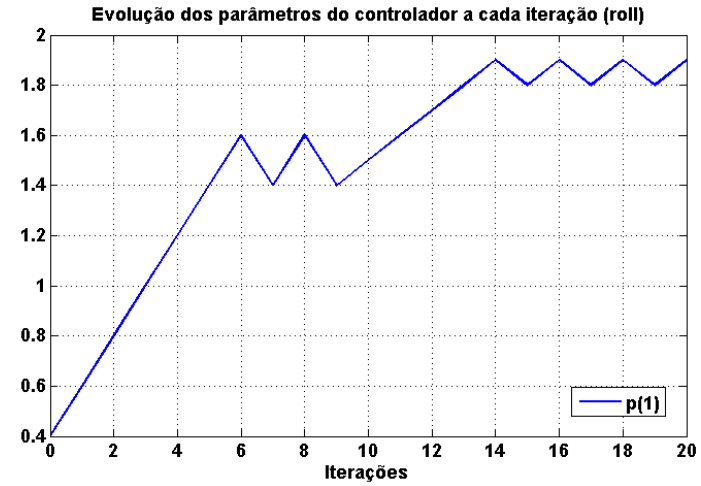

Figura 38. Evolução dos parâmetros do controlador a cada iteração, sendo ρ_1 a linha azul.

A evolução da função custo $J(\rho)$ também é mostrada, na Figura 39, com seu valor caindo de *65,56* para *21,58*.

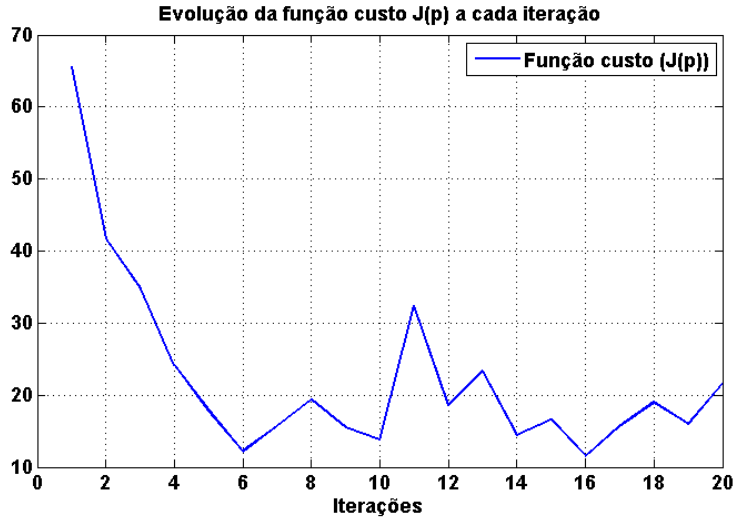

Figura 39. Evolução da função custo a cada iteração.

É mostrado, a seguir, um comparativo entre a resposta na condição inicial do sistema, com $\rho_1 = 0,4$ (Figura 40), e na condição final, com $\rho_1 = 1,9$ (Figura 41).

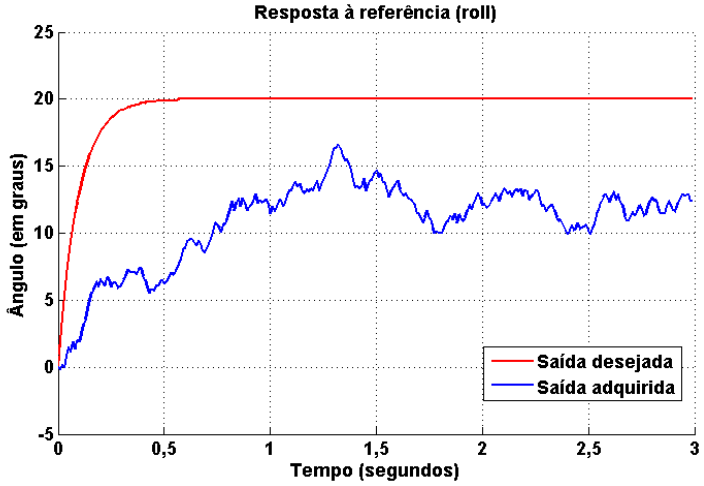

Figura 40. Resposta à referência, sendo a saída desejada a linha vermelha e a saída adquirida a linha azul.

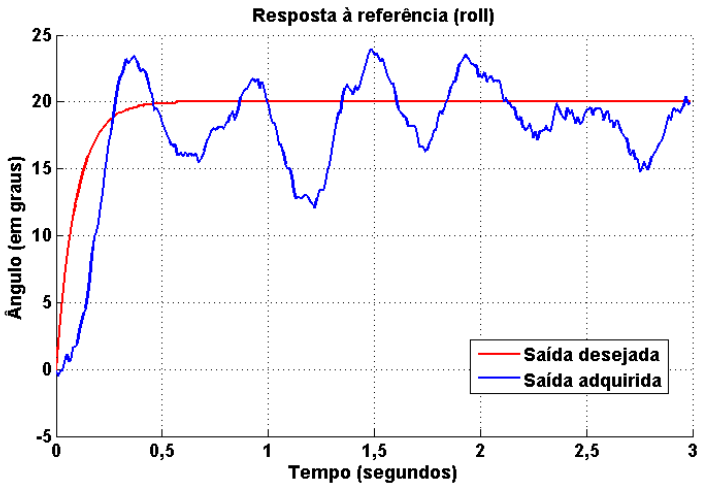

Figura 41. Resposta à referência, sendo a saída desejada a linha vermelha e a saída adquirida a linha azul.

Este comportamento excessivamente oscilatório é significativamente explicado pelo fato do quadricóptero estar amarrado dentro da caixa e próximo ao chão, pois o vento gerado pelas hélices (para baixo) não segue seu fluxo natural, sendo forçado a se expandir de forma radial à medida que se aproxima do solo, como explicado em (Prasad Pulla, 2006).

Este efeito é chamado de "efeito solo", ou *ground effect*, e provoca uma mudança nas características aerodinâmicas quando o veículo está próximo ao solo, aumentando sua sustentação. Nesse tópico, dois aspectos importantes são observados, um é o fato do ar recircular e aumentar o fluxo no rotor, e o outro é a criação de vórtices próximos ao rotor, que causam um desbalanceamento devido a um fluxo lateral significante, levando a certa instabilidade no sistema. Isto é demonstrado em (Ganesh & Komerath, 2004). Além disso, quando o *roll* é diferente de zero, as duas hélices opostas estão em alturas diferentes, fazendo com que o efeito seja mais intenso numa delas, causando oscilação.

Entretanto, é possível mesmo assim notar uma melhora na performance, bem como um comportamento correto do algoritmo, que ao modificar o parâmetro ρ_1 e fazê-lo convergir após algumas iterações, leva a função custo a níveis inferiores em relação ao início de sua execução.

No segundo teste, mostrado na sequência, foi utilizada como referência uma onda quadrada de frequência 0,4Hz e amplitude 20º. O número de amostras para cada experimento foi 500. O passo inicial (γ) teve valor 0,2, sendo reduzido ao longo das iterações até 0,01. Foram realizadas 50 iterações.

O parâmetro adquirido neste teste foi $\rho_1 = 1{,}78$, o qual tem sua evolução ao longo das iterações mostrada na Figura 42.

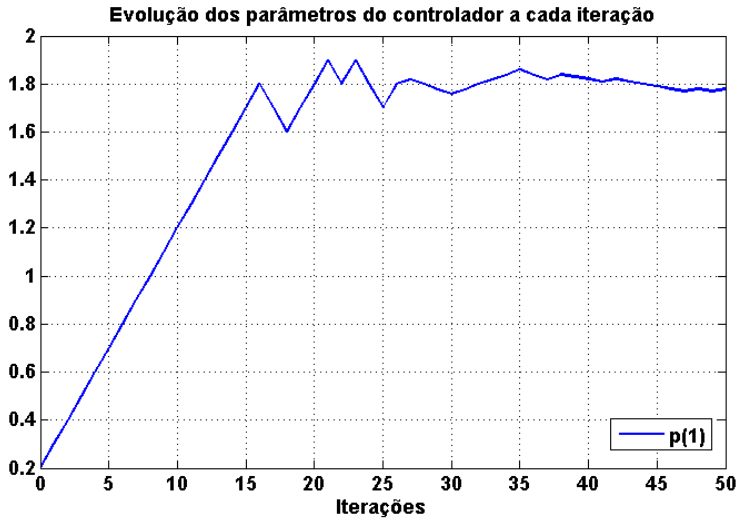

Figura 42. Evolução dos parâmetros do controlador a cada iteração, sendo ρ_1 a linha azul.

A evolução da função custo $J(\rho)$ é mostrada na Figura 43, sendo seu valor inicial igual a *172,96* e seu valor final igual a *40,33*.

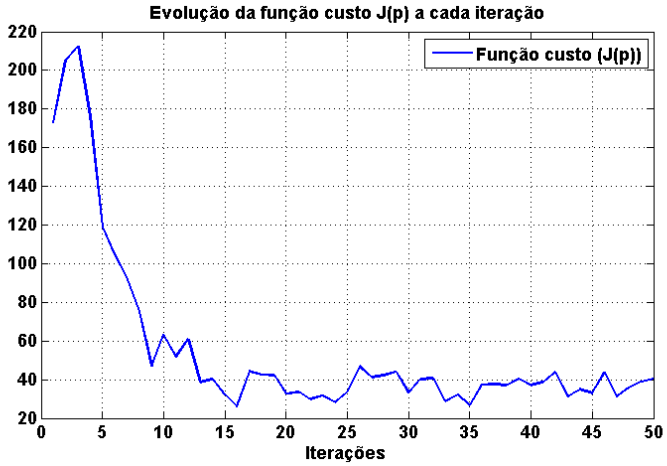

Figura 43. Evolução da função custo a cada iteração.

A Figura 44 mostra a saída real em relação à saída desejada no início da execução do algoritmo, com $\rho_1 = 0{,}2$, enquanto a Figura 45 mostra a saída real em relação à saída desejada ao final, com $\rho_1 = 1{,}78$.

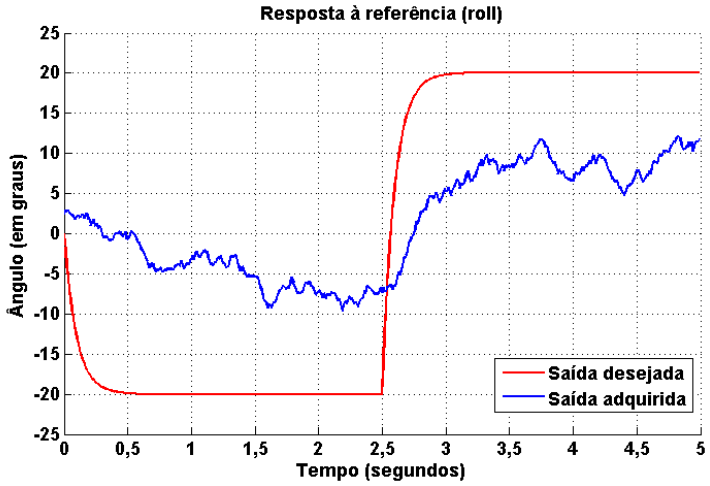

Figura 44. Resposta à referência, sendo a saída desejada a linha vermelha e a saída adquirida a linha azul.

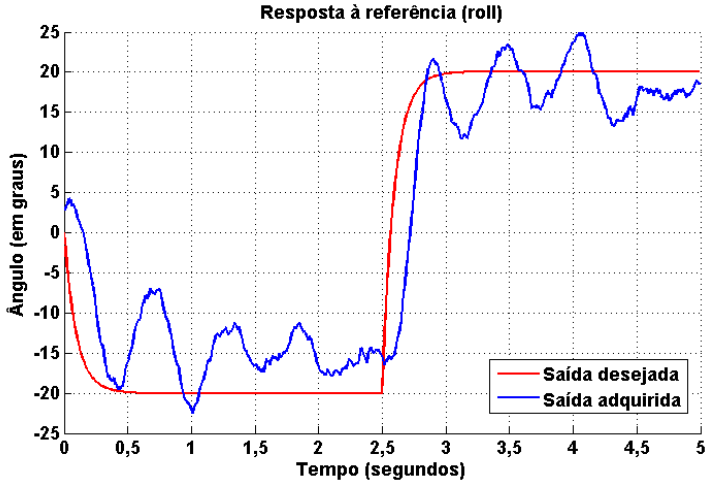

Figura 45. Resposta à referência, sendo a saída desejada a linha vermelha e a saída adquirida a linha azul.

É importante notar, na evolução da função custo, que nas primeiras iterações há uma variação mais significativa em relação à variação existente nas últimas iterações. Sendo assim, após algumas poucas iterações, neste caso 13, a função custo já está próxima a seu mínimo.

O terceiro e último teste foi realizado com a referência da Figura 46, que foi elaborada de forma que, caso o algoritmo fosse executado com o quadricóptero livre, ele voltasse, em teoria, à sua posição inicial.

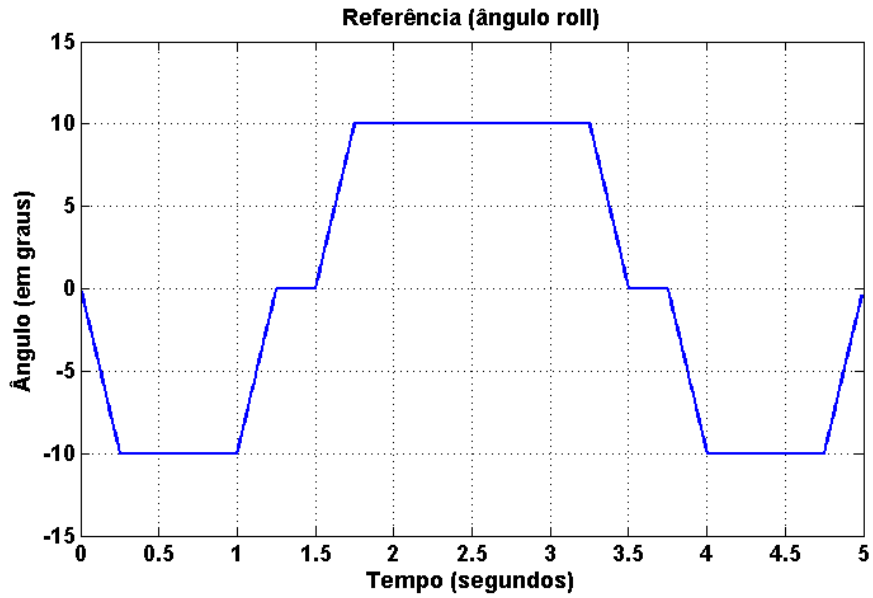

Figura 46. Referência de ângulo em graus.

O número de amostras para este teste foi 500. O passo inicial (γ) teve valor 0,2, sendo reduzido ao longo das iterações até 0,01. Foram realizadas 41 iterações.

O parâmetro ρ_1 inicial foi *0,4* e o encontrado foi *1,95*, sendo que sua evolução ao longo das iterações é mostrada na Figura 47.

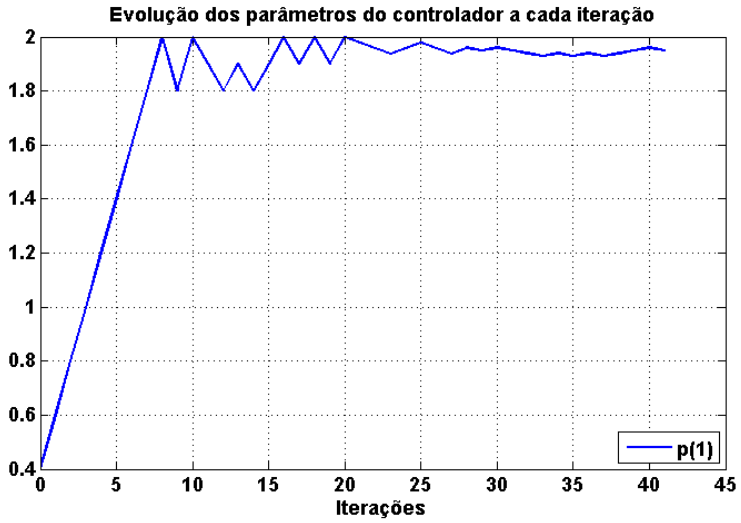

Figura 47. Evolução dos parâmetros do controlador a cada iteração, sendo ρ_1 a linha azul.

A evolução da função custo $J(\rho)$ é mostrada na Figura 48, com valor inicial *40,13* e valor final *10,89*.

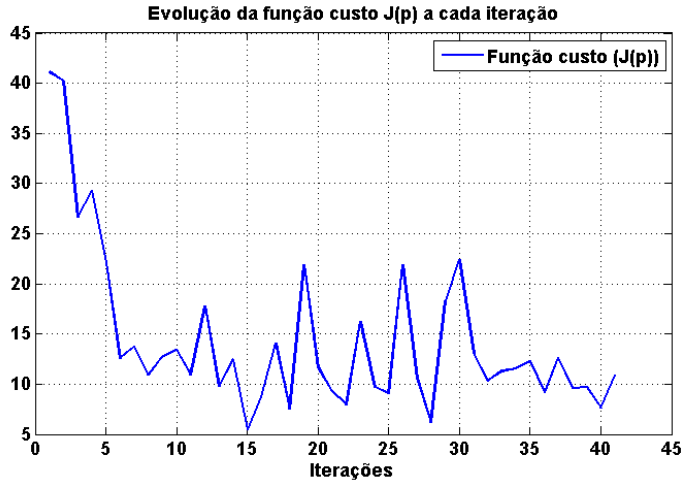

Figura 48. Evolução da função custo a cada iteração.

A Figura 49 mostra a saída obtida inicialmente, com $\rho_1 = 0{,}4$, e a Figura 50 mostra a saída obtida ao final da execução do algoritmo, com $\rho_1 = 1{,}95$.

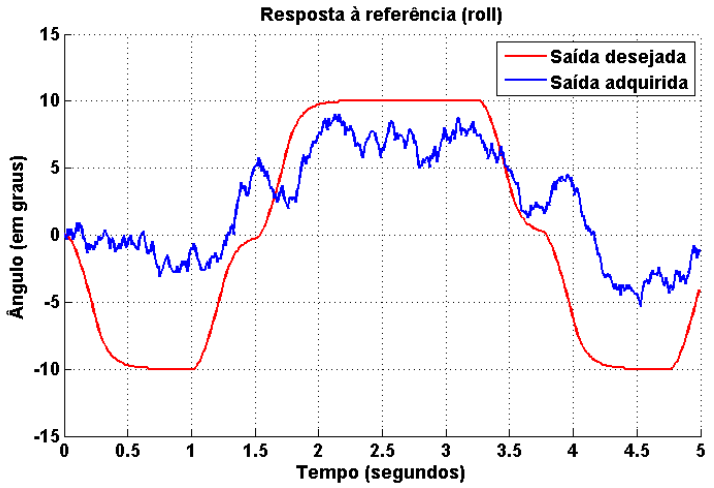

Figura 49. Resposta à referência, sendo a saída desejada a linha vermelha e a saída adquirida a linha azul.

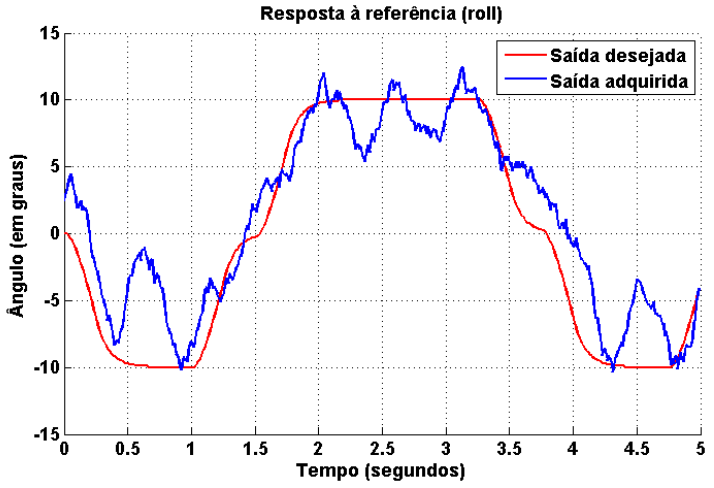

Figura 50. Resposta à referência, sendo a saída desejada a linha vermelha e a saída adquirida a linha azul.

8 CONCLUSÕES

Após a realização de diversas simulações do algoritmo do IFT, incluindo sua aplicação a um modelo teórico de quadricóptero, foi possível implementar com êxito este método na prática, através da programação do microcontrolador para realizar tal função, como mostrado no capítulo 7 .

O método em questão se mostrou eficiente para tal propósito pelo fato de não necessitar da modelagem do processo (que é naturalmente complexo) para seu controle, sendo possível chegar a um ajuste satisfatório de parâmetros baseando os cálculos apenas nas entradas e saídas do sistema. Além disso, tal vantagem é amplificada pois o algoritmo desenvolvido pode, em teoria, ser aplicado a qualquer quadricóptero que utilize um dos tipos de controlador discutidos neste trabalho.

Os resultados práticos, por sua vez, confirmam o comportamento observado nas simulações, mesmo que com níveis maiores de ruído e oscilações. Isto é constatado porque o algoritmo realiza o que é proposto, ou seja, otimiza um critério de desempenho específico, minimizando uma função custo de saídas adquiridas em relação a saídas desejadas (definidas de acordo com um modelo de referência para o sistema).

Uma limitação encontrada na aplicação prática é o tipo de controlador utilizado (apenas proporcional), de forma que o sistema poderia ter maior performance caso um PID, por exemplo, fosse aplicado. Entretanto, considerando as condições de operação, o algoritmo busca eficientemente os novos parâmetros, convergindo sempre para valores que reduzem significativamente a função custo.

Como o método é aplicável a outros controladores, com mais parâmetros a serem ajustados, uma possível sequência deste trabalho seria desenvolver o código do microcontrolador de forma a suportar um controlador PI ou PID. Tal situação foi inclusive considerada neste projeto, mas apenas em simulações do IFT. Além disso, variantes do método utilizado poderiam também ser exploradas, como através da inclusão do *"Newton-Raphson"* (explicado na seção 2.3.1) à direção de busca do algoritmo.

Outro aspecto que reforça a eficiência do algoritmo aplicado ao quadricóptero é o fato de, mesmo sendo utilizadas diferentes referências em cada teste, os parâmetros encontrados para o controlador têm valor próximo, dentro de uma margem aceitável.

A contribuição do presente trabalho se dá, então, no sentido de demonstrar, por meio de testes práticos e resultados que condizem com o que é esperado do método, ser viável o auto-ajuste de parâmetros de controle de um quadricóptero através do IFT, seja para modificar a resposta do sistema de acordo com a necessidade do usuário, seja para

reajustar o controle após possíveis variações na função de transferência do processo, sem a necessidade de exaustivos cálculos adicionais.

REFERÊNCIAS

Atmel, 2009. ATmega328P datasheet. *ATMEL 8-BIT MICROCONTROLLER WITH 4/8/16/32KBYTES IN-SYSTEM PROGRAMMABLE FLASH*, Dec, p. Revised Oct. 2010.

Bazanella, A. S., Campestrini, L. & Eckhard, D., 2012. *Data-Driven Controller Design: the H2 Approach.* Amsterdam: Springer.

Bazanella, A. S. & Silva Jr, J. M. G. d., 2005. *Sistemas de Controle - princípios e métodos de projeto.* Porto Alegre: UFRGS.

Bosch, 2010. BMA180 data sheet Version 2.5. 7 Dec.

Bouabdallah, S., Murrieri, P. & Siegwart, R., 2004. *Design and Control of an Indoor Micro Quadrotor*. New Orleans, LA, United States, s.n.

De Villeneuve, D. A., 1982. *DC Motor Speed Controller*. United States, Patente Nº 4309645.

Deshmukh, R. et al., 2015. ARDUINO BASED UAV QUADCOPTER. *International Journal for Engineering Applications and Technology*.

Ganesh, B. & Komerath, N., 2004. *Unsteady Aerodynamics of Rotorcraft in Ground Effect.* Portland, OR, s.n.

Gibiansky, A., 2012. *Quadcopter Dynamics and Simulation.* [Online] Available at: http://andrew.gibiansky.com/downloads/pdf/Quadcopter%20Dynamics,%20Simulation,%20and%20Control.pdf

[Accessed 14 April 2015].

Hoffman, G. M., Huang, H., Waslander, S. L. & Tomlin, C. J., 2007. *Quadrotor Helicopter Flight Dynamics and Control: Theory and Experiment.* Hilton Head, South Carolina, s.n.

InvenSense, 2010. ITG-3200 Product Specification Revision 1.4. 30 Mar.

Khan, M., 2014. Quadcopter Flight Dynamics. *INTERNATIONAL JOURNAL OF SCIENTIFIC & TECHNOLOGY RESEARCH*, Agosto, 3(8), pp. 130-135.

Mahony, R., Pounds, P., Roberts, J. & Hynes, P., 2002. *Design of a Four-Rotor Aerial Robot.* Auckland, s.n., pp. 145-150.

Microchip Technology Inc., 2003. Brushless DC (BLDC) Motor Fundamentals.

MICROMO, 2014. *DC Motor Calculations.* [Online] Available at: http://www.micromo.com/technical-library/dc-motor-tutorials/motor-calculations [Acesso em 29 05 2015].

Prasad Pulla, D., 2006. *A Study of Helicopter Aerodynamics in Ground Effect,* Columbus, OH: The Ohio State University.

Sam, R., Tan, M. N. M. & Ismail, M. S., 2013. *Quad-Copter using ATmega328 Microcontroller*. Busan, Korea, s.n.

ANEXO A

main.c

```
#include <stdio.h>
#include <avr/io.h>
#include <avr/interrupt.h>
#include <math.h>
#include <string.h>
#include <stdlib.h>
#include "mi2c.h"
#include "uart.h"
#define ROLL 0
#define PITCH 1
#define YAW 2
#define ACC_ORIENTATION(X, Y, Z)  {accADC[ROLL]  = Y; accADC[PITCH]  = X;
accADC[YAW]  =  Z;}
#define GYRO_ORIENTATION(X, Y, Z) {gyroADC[ROLL] =  X; gyroADC[PITCH] = -Y;
gyroADC[YAW] = -Z;}
#define ITG3200_ADDRESS 0X68
// #define ITG3200_SMPLRT_DIV 7  //8000Hz
//#define ITG3200_DLPF_CFG   0
#define ITG3200_SMPLRT_DIV 0  //8000Hz
#define ITG3200_DLPF_CFG   4
#define BMA180_ADDRESS 0x40
#define FOSC 16000000 // Clock Speed
#define BAUD 57600
#define MYUBRR FOSC/16/BAUD-1
#define MASK_P2 (1<<2)
#define MASK_P4 (1<<4)
#define MASK_P5 (1<<5)
#define MASK_P6 (1<<6)
#define MASK_P7 (1<<7)
uint8_t P2 = 0;//Status of pin 2 H or L
uint8_t P4 = 0;//Status of pin 4 H or L
uint8_t P5 = 0;//Status of pin 5 H or L
uint8_t P6 = 0;//Status of pin 6 H or L
uint8_t P7 = 0;//Status of pin 7 H or L
uint16_t dados;
float anglef[3];
int angle[2];
float calibra_giro[3];
float calibra_giro_i[3];
uint8_t armed=0;  //Armado-desarmado
uint8_t carmed=0; //Contador para armar
int tempo=0;  //Tempo usado no timer
uint8_t roda=0; //Se 1 roda loop se zero espera
int ocioso=0; //Tempo ocioso entre loops
int pr=0;
int motor[4];
int ref[4]={0,0,0,0};
int pino[4]={0,0,0,0};
int pid[3]={0,0,0};
float erro1[3]={0,0,0};
float erro2[3]={0,0,0};
float erro3[3]={0,0,0};
uint8_t rawADC[6];
int16_t gyroADC[3];
int16_t accADC[3];

int uart_putchar(char c, FILE *stream) {
    /*
     if (c == '\n') {
     uart_putchar('\r', stream);
     }
     loop_until_bit_is_set(UCSR0A, UDRE0);
     UDR0 = c;
     */
    uart0_putc(c);
    return 0;
```

```
}
FILE uart_output = FDEV_SETUP_STREAM(uart_putchar,NULL,_FDEV_SETUP_WRITE);

/* FUNCOES GYRO */

void Gyro_init(void) {
    i2c_writeReg(ITG3200_ADDRESS, 0x3E, 0x80); //register: Power Management  -
-  value: reset device
    i2c_writeReg(ITG3200_ADDRESS,   0x15,   ITG3200_SMPLRT_DIV);   //register:
Sample Rate Divider  -- default value = 0: OK
    i2c_writeReg(ITG3200_ADDRESS, 0x16, 0x18 + ITG3200_DLPF_CFG); //register:
DLPF_CFG - low pass filter configuration
    i2c_writeReg(ITG3200_ADDRESS, 0x3E, 0x03); //register: Power Management  -
-  value: PLL with Z Gyro reference
}

void Gyro_getADC(void) {
    TWBR = ((F_CPU / 400000L) - 16) / 2; // change the I2C clock rate to
400kHz
    i2c_getSixRawADC(ITG3200_ADDRESS,0X1D);
    GYRO_ORIENTATION( ((rawADC[0]<<8) | rawADC[1])>>2 , // range: +/- 8192;
+/- 2000 deg/sec
                      ((rawADC[2]<<8) | rawADC[3])>>2 ,
                      ((rawADC[4]<<8) | rawADC[5])>>2 );

    gyroADC[0]=gyroADC[0]-calibra_giro_i[0];
    gyroADC[1]=gyroADC[1]-calibra_giro_i[1];
    gyroADC[2]=gyroADC[2]-calibra_giro_i[2];
}

/* FUNCOES ACC */

void ACC_init(void) {
    //default range 2G: 1G = 4096 unit.
    i2c_writeReg(BMA180_ADDRESS,0x0D,1<<4); // register: ctrl_reg0   -- value:
set bit ee_w to 1 to enable writing
    uint8_t control = i2c_readReg(BMA180_ADDRESS, 0x20);
    control = control & 0x0F;        // save tcs register
    //control = control | (0x01 << 4); // register: bw_tcs reg: bits 4-7 to
set bw -- value: set low pass filter to 20Hz
    control = control | (0x07 << 4); // set low pass filter to 10Hz (bits
value = 0000xxxx)
    i2c_writeReg(BMA180_ADDRESS, 0x20, control);

    control = i2c_readReg(BMA180_ADDRESS, 0x30);
    control = control & 0xFC;        // save tco_z register
    control = control | 0x00;        // set mode_config to 0
    i2c_writeReg(BMA180_ADDRESS, 0x30, control);

    control = i2c_readReg(BMA180_ADDRESS, 0x35);
    control = control & 0xF1;        // save offset_x and smp_skip register
    control = control | (0x02 << 1); // set range to 8G
    i2c_writeReg(BMA180_ADDRESS, 0x35, control);
}

void ACC_getADC (void) {
    TWBR = ((F_CPU / 400000L) - 16) / 2;  // Optional line.  Sensor is good
for it in the spec.
    i2c_getSixRawADC(BMA180_ADDRESS,0x02);
    //usefull info is on the 14 bits  [2-15] bits  /4 => [0-13] bits  /4 => 12
bit resolution
    ACC_ORIENTATION( ((rawADC[1]<<8) | rawADC[0])>>4 ,
                     ((rawADC[3]<<8) | rawADC[2])>>4 ,
                     ((rawADC[5]<<8) | rawADC[4])>>4 );
}

/* FUNCOES MATEMATICA */

int16_t _atan2(int32_t y, int32_t x){
    float z = (float)y / x;
```

```
    int16_t a;
    if ( abs(y) < abs(x) ){
        a = 573 * z / (1.0f + 0.28f * z * z);
        if (x<0) {
            if (y<0) a -= 1800;
            else a += 1800;
        }
    } else {
        a = 900 - 573 * z / (z * z + 0.28f);
        if (y<0) a -= 1800;
    }
    return a;
}

/* FUNCOES MOTOR */

void angulos(void) {
    float t[2];
    t[0]=sqrt(square((float)accADC[2])+square((float)accADC[1]));
    t[1]=sqrt(square((float)accADC[2])+square((float)accADC[2]));
    if(accADC[2]<0){
        t[1]=-t[1];
    }
    angle[0]=_atan2(accADC[0],t[0]);
    angle[1]=_atan2(accADC[1],t[1]);

    t[0]=(float)gyroADC[0]*cos(anglef[1]/572.9)/sqrt(1-
square(sin(anglef[0]/572.9)*sin(anglef[1]/572.9)));
    t[1]=(float)gyroADC[2]*sin(anglef[1]/572.9)/sqrt(1-
square(sin(anglef[0]/572.9)*cos(anglef[1]/572.9)));
    t[0]=t[0]+t[1];

    anglef[0]=0.01*(float)angle[0]+0.99*anglef[0]+0.02754*(t[0]);

t[0]=(float)gyroADC[1]*cos(anglef[0]/572.9)/sqrt(1+square(sin(anglef[1]/572.9)
*sin(anglef[0]/572.9)));
    t[1]=(float)gyroADC[2]*sin(anglef[0]/572.9)/sqrt(1-
square(sin(anglef[1]/572.9)*cos(anglef[0]/572.9)));
    t[0]=t[0]-t[1];

    anglef[1]=0.01*(float)angle[1]+0.99*anglef[1]+0.02754*(t[0]);
}

void controle(void) {
    pid[0]=((ref[0]-(int)anglef[0])*2-gyroADC[0])*0.2;
    pid[1]=((ref[1]-(int)anglef[1])*2-gyroADC[1])*0.2;
    pid[2]=ref[3]-gyroADC[2];
}

void atualiza_pid(void) {
    int j;

    motor[0]=ref[2]+pid[0]-pid[1]-pid[2];
    motor[1]=ref[2]-pid[0]+pid[1]-pid[2];
    motor[2]=ref[2]-pid[0]-pid[1]+pid[2];
    motor[3]=ref[2]+pid[0]+pid[1]+pid[2];

    for(j=0;j<4;j++)
    {
        if(motor[j]>1000) motor[j]=1000;
        if(motor[j]<90) motor[j]=90;
        if(ref[2]<10) motor[j]=0;
    }
}

void atualiza_motor(void) {

    OCR2A = (motor[0]>>3)+125;
    OCR2B = (motor[1]>>3)+125;
    OCR1A = (motor[2]>>2)+250;
```

```
    OCR1B = (motor[3]>>2)+250;
}

void setup(void) {
    DDRB |= 1<<1; // pinMode(9,  OUTPUT);  //Saida PWM
    DDRB |= 1<<2; // pinMode(10, OUTPUT);  //Saida PWM
    DDRB |= 1<<3; // pinMode(11, OUTPUT);  //Saida PWM
    DDRD |= 1<<3; // pinMode(3, OUTPUT);  //Saida PWM

    DDRB |= 1<<5; // pinMode(13, OUTPUT);  //LED

    /* ENTRADAS NÃO PRECISA - Eh Padrao
     pinMode(2, INPUT);  //Saida PWM
     pinMode(4, INPUT);  //Saida PWM
     pinMode(5, INPUT);  //Saida PWM
     pinMode(6, INPUT);  //Saida PWM
     pinMode(7, INPUT);  //Saida PWM
     */

    cli(); // disable all interrupts

    //Timer 0
    TCCR0A = 0;
    TCCR0B = 0;
    TCNT0  = 0;
    OCR0A = 159;            //10ms
    TCCR0A |= (1 << WGM01);   // CTC mode
    TCCR0B |= (1 << CS00);    // prescaler
    TIMSK0 |= (1 << OCIE0A);  // enable timer compare interrupt

    //Interrupcao RC
    PCICR |= 0x04;  //Interrupt2 Portas
    PCMSK2 |= 0xF0;  //Interrupt2 Pinos

    sei(); // enable interrupts

    //Timer 1 - PWM - dividido por 64
    TCCR1A = 0;
    TCCR1B = 0;
    TCNT1  = 0;
    OCR1A = 250;
    OCR1B = 250;

    TCCR1A |= (1 << COM1A1);   // Normal mode A (non-inverted)
    TCCR1A |= (1 << COM1B1);   // Normal mode B (non-inverted)
    TCCR1A |= (1 << WGM11);   // Fast PWM 9 Bits
    TCCR1B |= (1 << WGM12);   // Fast PWM 9 Bits
    TCCR1B |= (1 << CS11);    // 64 prescaler
    TCCR1B |= (1 << CS10);    // 64 prescaler

    //Timer 2 - PWM - dividido por 128
    TCCR2A = 0;
    TCCR2B = 0;
    TCNT2  = 0;
    OCR2A = 125;
    OCR2B = 125;
    TCCR2A |= (1 << COM2A1);   // Normal mode A (non-inverted)
    TCCR2A |= (1 << COM2B1);   // Normal mode B (non-inverted)
    TCCR2A |= (1 << WGM21);   // Fast PWM
    TCCR2A |= (1 << WGM20);   // Fast PWM
    TCCR2B |= (1 << CS22);    // 128 prescaler
    TCCR2B |= (1 << CS20);    // 128 prescaler

    //No timer 1 deve ser colocado o DOBRO do valor do timer 2 para mesma
resposta pois PRESCALER eh diferente
    //Timer 1 entre 250 e 500
    //Timer 2 entre 125 e 250

    uart0_init(MYUBRR);
    stdout = &uart_output;
```

```
    TWBR = ((F_CPU / 400000L) - 16) / 2; // change the I2C clock rate to
400kHz
    Gyro_init();
    ACC_init();
}

ISR(TIMER0_COMPA_vect)          // timer compare interrupt service routine
{
    tempo++;
    if(tempo>1000)
    {
        roda=1;
        tempo=0;
    }
}

ISR(PCINT2_vect)
{
    //Interrupcao Pinos para PC

    if ((PIND & MASK_P5) != P5)//if digital PIN5 has changed
    {
        if(P5)
        {
            pino[0]=tempo-pino[0];
            if (pino[0]<0)
            {
                pino[0]=pino[0]+1000;
            }
            ref[0]=(pino[0]-150)*10;

        }else{
            pino[0]=tempo;
        }
        P5 = PIND & MASK_P5;//Set new status
    }
    if ((PIND & MASK_P4) != P4)//if digital PIN4 has changed
    {
        if(P4)
        {
            pino[1]=tempo-pino[1];
            if (pino[1]<0)
            {
                pino[1]=pino[1]+1000;
            }
            ref[1]=(pino[1]-150)*10;

        }else{
            pino[1]=tempo;
        }
        P4 = PIND & MASK_P4;//Set new status
    }
    if ((PIND & MASK_P6) != P6)//if digital PIN6 has changed
    {
        if(P6)
        {
            pino[2]=tempo-pino[2];
            if (pino[2]<0)
            {
                pino[2]=pino[2]+1000;
            }
            ref[2]=(pino[2]-110)*10;

        }else{
            pino[2]=tempo;
        }
        P6 = PIND & MASK_P6;//Set new status
    }
    if ((PIND & MASK_P7) != P7)//if digital PIN7 has changed
    {
```

```
        if(P7)
        {
            pino[3]=tempo-pino[3];
            if (pino[3]<0)
            {
                pino[3]=pino[3]+1000;
            }
            //ref[3]=(pino[3]-150)*10;
            ref[3]=-(pino[3]-150)*10;

        }else{
            pino[3]=tempo;
        }
        P7 = PIND & MASK_P7;//Set new status
    }
}

void loop(void) {
    ocioso++;
    if(roda==1){
        roda=0;
        printf("%d\t",ocioso);
        ocioso=0;
        Gyro_getADC();
        ACC_getADC();
        if(armed==0)
        {
            if((ref[2]<10) & (ref[3]>450))      carmed++;

            if(carmed==50){
                armed=1;
                calibra_giro_i[0]=(int)calibra_giro[0];
                calibra_giro_i[1]=(int)calibra_giro[1];
                calibra_giro_i[2]=(int)calibra_giro[2];
            }

            motor[0]=0;
            motor[1]=0;
            motor[2]=0;
            motor[3]=0;

            PORTB &= ~(1<<5); //Desliga LED

            calibra_giro[0]=0.1*gyroADC[0]+0.9*calibra_giro[0];
            calibra_giro[1]=0.1*gyroADC[1]+0.9*calibra_giro[1];
            calibra_giro[2]=0.1*gyroADC[2]+0.9*calibra_giro[2];
        }
        else
        {
            if((ref[2]<10) & (ref[3]<-450))    carmed--;

            if(carmed==0)
            {
                armed=0;
                calibra_giro_i[0]=0;
                calibra_giro_i[1]=0;
                calibra_giro_i[2]=0;
            }

            PORTB |= 1<<5; //Liga LED

            angulos();
            controle();
            atualiza_pid();
        }

        atualiza_motor();

        dados=uart0_getc();
```

```
        printf("%d\t",dados);
        printf("%d\t",ref[2]);
        printf("%d\t",ref[0]);
        printf("%d\t",ref[1]);
        printf("%d\t",(int)anglef[0]);
        printf("%d\t",(int)anglef[1]);
        printf("%d\t",gyroADC[0]);
        printf("%d\n",gyroADC[1]);
    }
}

int main(void){
    setup();

    for(;;)
    {
        loop();
    }

    return 0;
}
```

ANEXO B

IFT_TCC.m

```
% IFT_TCC.m: Implementation of IFT algorithm for test purposes.
% Author: Maurício Spiazzi Richter
% Date: 05/05/2015
% The control penaly is not being applied (lambda = 0).
close all,clear,clc % Variables and command-line clearing.
Ts = 0.01; % Sampling time.
z = tf('z', Ts); % Definition of z as a discrete frequency-domain
variable.
% Definition of G(z) coefficients, according to:
%         b3*z^3 + b2*z^2 + b1*z + b0
% G(z) = ---------------------------
%         a3*z^3 + a2*z^2 + a1*z + a0
b3 = 0; b2 = 0; b1 = 0; b0 = 2; B = [b3 b2 b1 b0];
a3 = 0; a2 = 0; a1 = 1; a0 = -0.3; A = [a3 a2 a1 a0];
% Definition of noise variance, where e(t) is white noise in the
following
% equation:
% y(t) = G(z)*u(t) + e(t)
noise_var = 0.0001; % Noise variance.
% Definition of Td(z) coefficients, according to:
%         w3*z^3 + w2*z^2 + w1*z + w0
% Td(z) = ---------------------------
%         v3*z^3 + v2*z^2 + v1*z + v0
w3 = 0; w2 = 0; w1 = 0; w0 = 0.1; W = [w3 w2 w1 w0];
v3 = 0; v2 = 0; v1 = 1; v0 = -0.9; V = [v3 v2 v1 v0];
% Initial controller coefficients and controller parametrization:
p1 = 0.2; p2 = 0.1;
% p0 = p1; C_type = tf(1,1,Ts); % P controller.
p0 = [p1; p2]; C_type = [z/(z-1); 1/(z-1)]; % PI controller.
dC_dp = C_type; % C(z) = p'*C_type, so dC(z)/dp = C_type.
% IFT parameters:
samples = 100; % Total number of samples per iteration.
iterations = 200; % Number of IFT iterations.
init_step = 0.05; % Step size.
% Control blocks definition:
G = tf(B,A,Ts); % G(z).
Td = tf(W,V,Ts); % Td(z).
Cd = Td/(G*(1-Td)); % Cd(z).
t = 0:Ts:samples*Ts; t(length(t)) = []; % Time vector.
ref = square(2*pi*t/Ts/100); % Time-domain initial reference signal
(r(t)).
yd = lsim(Td,ref); % yd(t).
p = zeros(iterations+1,length(p0)); p(1,:) = p0'; % Initialization of
p matrix.
step_size = zeros(iterations,1); step_size(1) = init_step; %
Initialization of the step size matrix.
step_factor = 1;
dJT_dp = zeros(iterations,length(p0)); % Initialization of JT matrix.
r = zeros(samples,2); r(:,1) = ref; % Initialization of the input
matrix.
y = zeros(samples,2); % Initialization of the output matrix.
u = zeros(samples,2); % Initialization of the control signal matrix.
% IFT loop:
for i = 1:iterations
    C(i) = p(i,:)*C_type; % Calculation of C(z).
    T = feedback(C(i)*G,1); % Calculation of T(z).
    S = 1/(1+C(i)*G); % Calculation of the feedback loop for the
noise.
    noise = sqrt(noise_var)*randn(1,samples); % Noise signal (e(t)).
```

```
    y(:,1) = lsim(T,r(:,1)) + lsim(S,noise); % Acquisition of output
data (first experiment).
    u(:,1) = lsim(C(i),r(:,1)-y(:,1)); % Acquisition of the control
signal data (first experiment).
    r(:,2) = r(:,1)-y(:,1); % Calculation of the next input.
    noise = sqrt(noise_var)*randn(1,samples); % Noise signal (e(t)).
    y(:,2) = lsim(T,r(:,2)) + lsim(S,noise); % Acquisition of output
data (second experiment).
    u(:,2) = lsim(C(i),r(:,2)-y(:,2)); % Acquisition of the control
signal data (second experiment).
    Q = 1/C(i)*dC_dp; % Calculation of the vector filter Q(z).
    dy_dp = lsim(Q,y(:,2)); % Calculation of dy(t)/dp.
    dJT_dp(i,:) = 2/samples*(y(:,1)-yd)'*dy_dp; % Calculation of the
derivative of the cost function.
    J(i) = 1/samples*sum((y(:,1)-yd).^2); % Calculation of the cost
function.
%     if i>1
%         if J(i)>J(i-1)
%             step_factor = step_factor+1;
%         end
%     end
    step_size(i) = init_step/step_factor;
    p(i+1,:) = p(i,:)-step_size(i)*(dJT_dp(i,:)/norm(dJT_dp(i,:))); %
Calculation of the controller parameters after each iteration.
end
C(i) = p(end,:)*C_type; % Calculation of the final C(z).
T = feedback(C(i)*G,1); % Calculation of the final T(z).
S = 1/(1+C(i)*G); % Calculation of the final S(z).
noise = sqrt(noise_var)*randn(1,samples); % Noise signal (e(t)).
y(:,1) = lsim(T,r(:,1)) + lsim(S,noise); % Acquisition of the
resultant output data.
final_cont = p(end,:) % Final controller parameters.
J(i) = 1/samples*sum((y(:,1)-yd).^2); % Calculation of the resultant
cost function.
J(end)
figure
hold on
plot(t,yd,'red')
plot(t,y(:,1))
title('Resposta à referência utilizada no IFT')
xlabel('Tempo (segundos)'),ylabel('Amplitude')
grid
yd_step = step(Td,Ts*100);
y_step = step(T,Ts*100);
J2 = 1/length(y_step)*sum((y_step-yd_step).^2) % Calculation of the
cost function for the step responses.
figure
hold on
step(Td,Ts*100,'r')
step(T,Ts*100)
grid
figure
plot(p)
title('Evolução dos parâmetros do controlador a cada iteração')
xlabel('Iterações')
grid
figure
plot(J)
title('Evolução da função custo a cada iteração')
xlabel('Iterações')
grid
```

ANEXO C

IFT_simulacao.m

```
% IFT_simulacao.m: Implementation of IFT algorithm for test purposes.
% Author: Maurício Spiazzi Richter
% Date: 22/05/2015
% This file is an implementation of the data-driven control method
% called Iterative Feedback Tuning (IFT).

% The control penaly is not being applied (lambda = 0).
close all, clear, clc % Variables and command-line clearing.
run('param.m')
ts = 0.01; % Sampling time.
var = 0.00001; % Noise variance.
z = tf('z', ts); % Definition of z as a discrete frequency-domain
variable.
% Definition of Td(z) coefficients, according to:
%        w3*z^3 + w2*z^2 + w1*z + w0
% Td(z) = --------------------------
%        v3*z^3 + v2*z^2 + v1*z + v0
w3 = 0; w2 = 0; w1 = 0; w0 = 0.2; W = [w3 w2 w1 w0];
v3 = 0; v2 = 0; v1 = 1; v0 = -0.8; V = [v3 v2 v1 v0];
% Initial controller coefficients and controller parametrization:
p1_roll = 1; p2_roll = 0; p3_roll = 0;
p0_roll = [p1_roll]; C_type_roll = [tf(1,1,ts)]; % P controller.
% p0_roll = [p1_roll; p2_roll]; C_type_roll = [z/(z-1); 1/(z-1)]; % PI
controller.
% p0_roll = [p1_roll; p2_roll; p3_roll]'; C_type_roll = [1; z/(z-1);
(z-1)/z]; % PID controller.
dC_dp_roll = C_type_roll; % C(z) = p'*C_type, so dC(z)/dp = C_type.
p1_pitch = 0.2; p2_pitch = 0; p3_pitch = 0;
p0_pitch = [p1_pitch]; C_type_pitch = [tf(1,1,ts)]; % P controller.
% p0_pitch = [p1_pitch; p2_pitch]; C_type_pitch = [z/(z-1); 1/(z-1)];
% PI controller.
% p0_pitch = [p1_pitch; p2_pitch; p3_pitch]'; C_type_pitch = [1; z/(z-
1); (z-1)/z]; % PID controller.
dC_dp_pitch = C_type_pitch; % C(z) = p'*C_type, so dC(z)/dp = C_type.

% IFT parameters:
samples = 500; % Total number of samples per iteration.
Tsim = samples*ts; % Total simulation time.
iterations = 70; % Number of IFT iterations.
init_step_roll = 0.01; % Initial step size.
init_step_pitch = 0; % Initial step size.

% Control blocks definition:
Td = tf(W,V,ts); % Td(z).

t = 0:ts:samples*ts; % t(length(t)) = []; % Time vector.
ref_roll = 0.02*[40*t(1:51) 20*ones(1,25) -40*t(2:51)+20
zeros(1,25)...
            -40*t(2:51) -20*ones(1,100) 40*t(2:51)-20 zeros(1,25)...
            40*t(2:51) 20*ones(1,25) -40*t(2:51)+20]; % Time-domain
initial reference signal (r(t)) for the roll angle.
ref_pitch = zeros(length(t),1); % Time-domain initial reference signal
(r(t)) for the pitch angle.
yd_roll = lsim(Td,ref_roll); % yd(t) for the roll angle.
yd_pitch = lsim(Td,ref_pitch); % yd(t) for the pitch angle.

test_roll = 1; % Variable to carry out tests.
a_roll = 0; % Variable to carry out tests.
```

```
test_pitch = 1; % Variable to carry out tests.
a_pitch = 0; % Variable to carry out tests.

p_roll = zeros(iterations+1,length(p0_roll)); p_roll(1,:) = p0_roll';
% Initialization of p matrix.
p_pitch = zeros(iterations+1,length(p0_pitch)); p_pitch(1,:) =
p0_pitch'; % Initialization of p matrix.
step_size_roll = zeros(iterations+1,1); step_size_roll(1) =
init_step_roll; % Initialization of the step size matrix.
step_size_pitch = zeros(iterations+1,1); step_size_pitch(1) =
init_step_pitch; % Initialization of the step size matrix.
R_roll = ones(iterations+1,1); % Initialization of the search
direction (1 for 'steepest descent' method).
R_pitch = ones(iterations+1,1); % Initialization of the search
direction (1 for 'steepest descent' method).
dJT_dp_roll = zeros(iterations,length(p0_roll)); % Initialization of
JT matrix.
dJT_dp_pitch = zeros(iterations,length(p0_pitch)); % Initialization of
JT matrix.
J_roll = zeros(iterations,1); % Initialization of the cost function
J(p).
J_pitch = zeros(iterations,1); % Initialization of the cost function
J(p).
r_roll(:,1) = ref_roll; % Initialization of the input matrix.
r_pitch(:,1) = ref_pitch; % Initialization of the input matrix.

% IFT loop:
for i = 1:iterations
    C_roll(i) = p_roll(i,:)*C_type_roll; % Calculation of C(z).
    C_pitch(i) = p_pitch(i,:)*C_type_pitch; % Calculation of C(z).
    ref_r = r_roll(:,1); % Definition of the simulation roll reference
(first experiment).
    ref_p = r_pitch(:,1); % Definition of the simulation pitch
reference (first experiment).
    sim('Sim_teste2copia')
    y_roll(:,1) = y(:,1); % Acquisition of output data (first
experiment).
    y_pitch(:,1) = y(:,2); % Acquisition of output data (first
experiment).
    u_roll(:,1) = u(:,1); % Acquisition of the control signal data
(first experiment).
    u_pitch(:,1) = u(:,2); % Acquisition of the control signal data
(first experiment).
    J_roll(i) = 1/samples*sum((y_roll(:,1)-yd_roll).^2); % Calculation
of the cost function J(p).
    J_pitch(i) = 1/samples*sum((y_pitch(:,1)-yd_pitch).^2); %
Calculation of the cost function J(p).
    r_roll(:,2) = r_roll(:,1)-y_roll(:,1); % Calculation of the next
input.
    r_pitch(:,2) = r_pitch(:,1)-y_pitch(:,1); % Calculation of the
next input.
    ref_r = r_roll(:,2); % Definition of the simulation roll reference
(second experiment).
    ref_p = r_pitch(:,2); % Definition of the simulation pitch
reference (second experiment).
    sim('Sim_teste2copia')
    y_roll(:,2) = y(:,1); % Acquisition of output data (second
experiment).
    y_pitch(:,2) = y(:,2); % Acquisition of output data (second
experiment).
    u_roll(:,2) = u(:,1); % Acquisition of the control signal data
(second experiment).
```

```
    u_pitch(:,2) = u(:,2); % Acquisition of the control signal data
(second experiment).
    Q_roll = 1/C_roll(i)*dC_dp_roll; % Calculation of the vector
filter Q(z).
    Q_pitch = 1/C_pitch(i)*dC_dp_pitch; % Calculation of the vector
filter Q(z).
    dy_dp_roll = lsim(Q_roll,y_roll(:,2)); % Calculation of dy(t)/dp.
    dy_dp_pitch = lsim(Q_pitch,y_pitch(:,2)); % Calculation of
dy(t)/dp.
    dJT_dp_roll(i,:) = 2/samples*(y_roll(:,1)-yd_roll)'*dy_dp_roll; %
Calculation of the derivative of the cost function.
    dJT_dp_pitch(i,:) = 2/samples*(y_pitch(:,1)-
yd_pitch)'*dy_dp_pitch; % Calculation of the derivative of the cost
function.
    if i>1
        if J_roll(i)>J_roll(i-1)
            a_roll(i) = i;
            test_roll = test_roll+1;
        end
        if J_pitch(i)>J_pitch(i-1)
            a_pitch(i) = i;
            test_pitch = test_pitch+1;
        end
        step_size_roll(i) = init_step_roll; % /test_roll;
        step_size_pitch(i) = 0;
    end
    p_roll(i+1,:) = p_roll(i,:)-
step_size_roll(i)*R_roll(i)*(dJT_dp_roll(i,:)/norm(dJT_dp_roll(i,:)));
% Calculation of the controller parameters after each iteration.
    p_pitch(i+1,:) = p_pitch(i,:)-
step_size_pitch(i)*R_pitch(i)*(dJT_dp_pitch(i,:)/norm(dJT_dp_pitch(i,:
))); % Calculation of the controller parameters after each iteration.
    i
end
C_roll(i) = p_roll(end,:)*C_type_roll; % Calculation of the final
C(z).
C_pitch(i) = p_pitch(end,:)*C_type_pitch; % Calculation of the final
C(z).
ref_r = r_roll(:,1); % Definition of the simulation roll reference.
ref_p = r_pitch(:,1); % Definition of the simulation pitch reference.
sim('Sim_teste2copia')
y_roll(:,1) = y(:,1); % Acquisition of the resultant output data.
y_pitch(:,1) = y(:,2); % Acquisition of the resultant output data.
u_roll(:,1) = u(:,1); % Acquisition of the final control signal data.
u_pitch(:,1) = u(:,2); % Acquisition of the final control signal data.

% Display of the results.
final_cont_roll = p_roll(end,:) % Final roll controller parameters.
J_roll(i+1) = 1/samples*sum((y_roll(:,1)-yd_roll).^2) % Calculation of
the resultant cost function.
figure
hold on
plot(t,yd_roll,'red')
plot(t,y_roll(:,1))
title('Resposta à referência (roll)')
xlabel('Tempo (segundos)'),ylabel('Velocidade angular (rad/s)')
legend('Saída desejada','Saída adquirida')
grid
figure
plot(p_roll)
title('Evolução dos parâmetros do controlador a cada iteração (roll)')
xlabel('Iterações')
legend('p(1)','p(2)')
```

```
grid
figure
plot(J_roll)
title('Evolução da função custo J(p) a cada iteração (roll)')
xlabel('Iterações')
legend('Função custo (J(p))')
grid
```

ANEXO D

main.c

```
#include <stdio.h>
#include <avr/io.h>
#include <avr/interrupt.h>
#include <math.h>
#include <string.h>
#include <stdlib.h>
#include "mi2c.h"
#include "uart.h"
#include <avr/pgmspace.h>
#define ROLL 0
#define PITCH 1
#define YAW 2
#define ACC_ORIENTATION(X, Y, Z)  {accADC[ROLL]  = Y; accADC[PITCH]  = X;
accADC[YAW]  =  Z;}
#define GYRO_ORIENTATION(X, Y, Z) {gyroADC[ROLL] =  X; gyroADC[PITCH] = -Y;
gyroADC[YAW] = -Z;}
#define ITG3200_ADDRESS 0X68
// #define ITG3200_SMPLRT_DIV 7  //8000Hz
//#define ITG3200_DLPF_CFG   0
#define ITG3200_SMPLRT_DIV 0  //8000Hz
#define ITG3200_DLPF_CFG   4
#define BMA180_ADDRESS 0x40
#define FOSC 16000000 // Clock Speed
#define BAUD 57600
#define MYUBRR FOSC/16/BAUD-1
#define MASK_P2 (1<<2)
#define MASK_P4 (1<<4)
#define MASK_P5 (1<<5)
#define MASK_P6 (1<<6)
#define MASK_P7 (1<<7)
/* Definições para IFT */
#define TOTAL_AMOSTRAS 550 // Total de amostras em cada experimento.
/* ------------------- */
uint8_t P2 = 0;//Status of pin 2 H or L
uint8_t P4 = 0;//Status of pin 4 H or L
uint8_t P5 = 0;//Status of pin 5 H or L
uint8_t P6 = 0;//Status of pin 6 H or L
uint8_t P7 = 0;//Status of pin 7 H or L
uint16_t dados;
float anglef[3];
int angle[2];
float calibra_giro[3];
float calibra_giro_i[3];
uint8_t armed=0;  //Armado-desarmado
uint8_t carmed=0; //Contador para armar
int tempo=0;  //Tempo usado no timer
uint8_t roda=0; //Se 1 roda loop se zero espera
int ocioso=0; //Tempo ocioso entre loops
/* Variáveis para IFT */
uint8_t ajuste = 0; // Se ajuste = 1, roda o IFT.
int amostra = 0; // Número da amostra sendo executada.
uint8_t exp_n = 0; // Número do experimento sesndo executado.
int iter = 0; // Número da iteração sendo executada.
const int ref_roll[] PROGMEM =
{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,\
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,\
0,4,8,12,16,20,24,28,32,36,40,44,48,52,56,60,64,68,72,76,80,84,88,92,96,\
100,100,100,100,100,100,100,100,100,100,100,100,100,100,100,100,100,100,100,10
0,100,100,100,100,100,\
100,100,100,100,100,100,100,100,100,100,100,100,100,100,100,100,100,100,100,10
0,100,100,100,100,100,\
100,100,100,100,100,100,100,100,100,100,100,100,100,100,100,100,100,100,100,10
0,100,100,100,100,100,\
100,96,92,88,84,80,76,72,68,64,60,56,52,48,44,40,36,32,28,24,20,16,12,8,4,\
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,\
```

```
0,-4,-8,-12,-16,-20,-24,-28,-32,-36,-40,-44,-48,-52,-56,-60,-64,-68,-72,-76,-
80,-84,-88,-92,-96,\
-100,-100,-100,-100,-100,-100,-100,-100,-100,-100,-100,-100,-100,-100,-100,-
100,-100,-100,-100,-100,-100,-100,-100,-100,-100,\
-100,-100,-100,-100,-100,-100,-100,-100,-100,-100,-100,-100,-100,-100,-100,-
100,-100,-100,-100,-100,-100,-100,-100,-100,-100,\
-100,-100,-100,-100,-100,-100,-100,-100,-100,-100,-100,-100,-100,-100,-100,-
100,-100,-100,-100,-100,-100,-100,-100,-100,-100,\
-100,-100,-100,-100,-100,-100,-100,-100,-100,-100,-100,-100,-100,-100,-100,-
100,-100,-100,-100,-100,-100,-100,-100,-100,-100,\
-100,-100,-100,-100,-100,-100,-100,-100,-100,-100,-100,-100,-100,-100,-100,-
100,-100,-100,-100,-100,-100,-100,-100,-100,-100,\
-100,-100,-100,-100,-100,-100,-100,-100,-100,-100,-100,-100,-100,-100,-100,-
100,-100,-100,-100,-100,-100,-100,-100,-100,-100,\
-100,-96,-92,-88,-84,-80,-76,-72,-68,-64,-60,-56,-52,-48,-44,-40,-36,-32,-28,-
24,-20,-16,-12,-8,-4,\
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,\
0,4,8,12,16,20,24,28,32,36,40,44,48,52,56,60,64,68,72,76,80,84,88,92,96,\
100,100,100,100,100,100,100,100,100,100,100,100,100,100,100,100,100,100,100,10
0,100,100,100,100,100,\
100,100,100,100,100,100,100,100,100,100,100,100,100,100,100,100,100,100,100,10
0,100,100,100,100,100,\
100,100,100,100,100,100,100,100,100,100,100,100,100,100,100,100,100,100,100,10
0,100,100,100,100,100,\
100,96,92,88,84,80,76,72,68,64,60,56,52,48,44,40,36,32,28,24,20,16,12,8,4}; //
Vetor de referências.
const int saida_ideal[] PROGMEM =
{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,\
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,\
0,0,0,1,2,4,5,7,9,11,14,17,19,22,25,28,31,35,38,41,45,48,52,56,59,\
63,67,70,73,76,78,80,82,84,86,87,88,90,91,92,92,93,94,94,95,95,96,96,97,97,\
97,98,98,98,98,98,99,99,99,99,99,99,99,99,99,99,100,100,100,100,100,100,100,10
0,100,\
100,100,100,100,100,100,100,100,100,100,100,100,100,100,100,100,100,100,100,10
0,100,100,100,100,100,\
100,100,100,99,98,96,95,93,91,88,86,83,81,78,75,72,69,65,62,59,55,52,48,44,41,
\
37,33,30,27,24,22,20,18,16,14,13,12,10,9,8,8,7,6,6,5,5,4,4,3,3,\
3,2,2,1,0,-2,-4,-6,-8,-10,-13,-16,-19,-21,-25,-28,-31,-34,-38,-41,-45,-48,-
52,-55,-59,\
-63,-66,-70,-73,-76,-78,-80,-82,-84,-86,-87,-88,-89,-91,-91,-92,-93,-94,-94,-
95,-95,-96,-96,-97,-97,\
-97,-98,-98,-98,-98,-98,-99,-99,-99,-99,-99,-99,-99,-99,-99,-99,-100,-100,-
100,-100,-100,-100,-100,-100,-100,\
-100,-100,-100,-100,-100,-100,-100,-100,-100,-100,-100,-100,-100,-100,-100,-
100,-100,-100,-100,-100,-100,-100,-100,-100,-100,\
-100,-100,-100,-100,-100,-100,-100,-100,-100,-100,-100,-100,-100,-100,-100,-
100,-100,-100,-100,-100,-100,-100,-100,-100,-100,\
-100,-100,-100,-100,-100,-100,-100,-100,-100,-100,-100,-100,-100,-100,-100,-
100,-100,-100,-100,-100,-100,-100,-100,-100,-100,\
-100,-100,-100,-100,-100,-100,-100,-100,-100,-100,-100,-100,-100,-100,-100,-
100,-100,-100,-100,-100,-100,-100,-100,-100,-100,\
-100,-100,-100,-99,-98,-96,-95,-93,-91,-89,-86,-83,-81,-78,-75,-72,-69,-65,-
62,-59,-55,-52,-48,-44,-41,\
-37,-33,-30,-27,-24,-22,-20,-18,-16,-14,-13,-12,-10,-9,-8,-8,-7,-6,-6,-5,-5,-
4,-4,-3,-3,\
-3,-2,-2,-1,0,2,4,6,8,10,13,16,19,21,25,28,31,34,38,41,45,48,52,55,59,\
63,66,70,73,76,78,80,82,84,86,87,88,89,91,91,92,93,94,94,95,95,96,96,97,97,\
97,98,98,98,98,98,99,99,99,99,99,99,99,99,99,99,100,100,100,100,100,100,100,10
0,100,\
100,100,100,100,100,100,100,100,100,100,100,100,100,100,100,100,100,100,100,10
0,100,100,100,100,100,\
100,100,100,99,98,96,95,93,91,88,86,83,81,78,75,72,69,65,62,59,55,52,48,44,41}
; // Vetor com a saída ideal para esta referência, de acordo com a FT
desejada.
int saida[TOTAL_AMOSTRAS]; // Variável onde a saída do primeiro experimento é
salva.
float dy_dp; // Variável para salvar a derivada da saída em relação a rho.
double dJT_dp = 0;
double J = 0;
```

```
float passo = 0.2;
int k; // Variável auxiliar para 'for'.
int sem_comando = 0; // Temporizador que desarma o quad caso a letra 'c' não
seja recebida pela serial por 3s.
int transicao = 0; // Variável para definir a transição entre os experimentos
1 e 2.
float K_int = 0.3; // Controlador proporcional interno (para a velocidade
angular).
float K_int_inv; // Variável auxiliar para evitar o cálculo recorrente de
(1/K_int).
float K_ext = 1.95; // Controlador proporcional externo (para o ângulo).
float K_ext_inv; // Variável auxiliar para evitar o cálculo recorrente de
(1/K_ext).
/* ------------------ */
/* Variáveis para testes */
int aux1 = 0;
long int erro;
/* ---------------------- */
int pr=0;
int motor[4];
int ref[4]={0,0,0,0};
int pino[4]={0,0,0,0};
int pid[3]={0,0,0};
float erro1[3]={0,0,0};
float erro2[3]={0,0,0};
float erro3[3]={0,0,0};
uint8_t rawADC[6];
int16_t gyroADC[3];
int16_t accADC[3];
int uart_putchar(char c, FILE *stream) {
    /*
     if (c == '\n') {
     uart_putchar('\r', stream);
     }
     loop_until_bit_is_set(UCSR0A, UDRE0);
     UDR0 = c;
     */
    uart0_putc(c);
    return 0;
}
FILE uart_output = FDEV_SETUP_STREAM(uart_putchar,NULL,_FDEV_SETUP_WRITE);

/* FUNCOES GYRO */

void Gyro_init(void) {
    i2c_writeReg(ITG3200_ADDRESS, 0x3E, 0x80); //register: Power Management  -
-  value: reset device
    i2c_writeReg(ITG3200_ADDRESS, 0x15, ITG3200_SMPLRT_DIV); //register:
Sample Rate Divider  -- default value = 0: OK
    i2c_writeReg(ITG3200_ADDRESS, 0x16, 0x18 + ITG3200_DLPF_CFG); //register:
DLPF_CFG - low pass filter configuration
    i2c_writeReg(ITG3200_ADDRESS, 0x3E, 0x03); //register: Power Management  -
-  value: PLL with Z Gyro reference
}
void Gyro_getADC(void) {
    TWBR = ((F_CPU / 400000L) - 16) / 2; // change the I2C clock rate to
400kHz
    i2c_getSixRawADC(ITG3200_ADDRESS,0X1D);
    GYRO_ORIENTATION( ((rawADC[0]<<8) | rawADC[1])>>2 , // range: +/- 8192;
+/- 2000 deg/sec
                      ((rawADC[2]<<8) | rawADC[3])>>2 ,
                      ((rawADC[4]<<8) | rawADC[5])>>2 );
    gyroADC[0]=gyroADC[0]-calibra_giro_i[0];
    gyroADC[1]=gyroADC[1]-calibra_giro_i[1];
    gyroADC[2]=gyroADC[2]-calibra_giro_i[2];
}

/* FUNCOES ACC */

void ACC_init(void) {
```

```
    //default range 2G: 1G = 4096 unit.
    i2c_writeReg(BMA180_ADDRESS,0x0D,1<<4); // register: ctrl_reg0  -- value:
set bit ee_w to 1 to enable writing
    uint8_t control = i2c_readReg(BMA180_ADDRESS, 0x20);
    control = control & 0x0F;        // save tcs register
    //control = control | (0x01 << 4); // register: bw_tcs reg: bits 4-7 to
set bw -- value: set low pass filter to 20Hz
    control = control | (0x07 << 4); // set low pass filter to 10Hz (bits
value = 0000xxxx)
    i2c_writeReg(BMA180_ADDRESS, 0x20, control);

    control = i2c_readReg(BMA180_ADDRESS, 0x30);
    control = control & 0xFC;        // save tco_z register
    control = control | 0x00;        // set mode_config to 0
    i2c_writeReg(BMA180_ADDRESS, 0x30, control);

    control = i2c_readReg(BMA180_ADDRESS, 0x35);
    control = control & 0xF1;        // save offset_x and smp_skip register
    control = control | (0x02 << 1); // set range to 8G
    i2c_writeReg(BMA180_ADDRESS, 0x35, control);
}

void ACC_getADC (void) {
    TWBR = ((F_CPU / 400000L) - 16) / 2;  // Optional line.  Sensor is good
for it in the spec.
    i2c_getSixRawADC(BMA180_ADDRESS,0x02);
    //usefull info is on the 14 bits  [2-15] bits  /4 => [0-13] bits  /4 => 12
bit resolution
    ACC_ORIENTATION( ((rawADC[1]<<8) | rawADC[0])>>4 ,
                     ((rawADC[3]<<8) | rawADC[2])>>4 ,
                     ((rawADC[5]<<8) | rawADC[4])>>4 );
}

/* FUNCOES MATEMATICA */
int16_t _atan2(int32_t y, int32_t x){
    float z = (float)y / x;
    int16_t a;
    if ( abs(y) < abs(x) ){
        a = 573 * z / (1.0f + 0.28f * z * z);
        if (x<0) {
            if (y<0) a -= 1800;
            else a += 1800;
        }
    } else {
        a = 900 - 573 * z / (z * z + 0.28f);
        if (y<0) a -= 1800;
    }
    return a;
}

/* FUNCOES MOTOR */
void angulos(void) {
    float t[2];
    t[0]=sqrt(square((float)accADC[2])+square((float)accADC[1]));
    t[1]=sqrt(square((float)accADC[2])+square((float)accADC[2]));
    if(accADC[2]<0){
        t[1]=-t[1];
    }
    angle[0]=_atan2(accADC[0],t[0]);
    angle[1]=_atan2(accADC[1],t[1]);
    t[0]=(float)gyroADC[0]*cos(anglef[1]/572.9)/sqrt(1-
square(sin(anglef[0]/572.9)*sin(anglef[1]/572.9)));
    t[1]=(float)gyroADC[2]*sin(anglef[1]/572.9)/sqrt(1-
square(sin(anglef[0]/572.9)*cos(anglef[1]/572.9)));
    t[0]=t[0]+t[1];
    anglef[0]=0.01*(float)angle[0]+0.99*anglef[0]+0.02754*(t[0]);

t[0]=(float)gyroADC[1]*cos(anglef[0]/572.9)/sqrt(1+square(sin(anglef[1]/572.9)
*sin(anglef[0]/572.9)));
```

```
    t[1]=(float)gyroADC[2]*sin(anglef[0]/572.9)/sqrt(1-
square(sin(anglef[1]/572.9)*cos(anglef[0]/572.9)));
    t[0]=t[0]-t[1];

    anglef[1]=0.01*(float)angle[1]+0.99*anglef[1]+0.02754*(t[0]);

void controle(void) {
      if (ajuste==1) {
      if (exp_n==1)
      {
            if (transicao==0) {
                  ref[0] = (int)pgm_read_word(&ref_roll[amostra]);
                  saida[amostra] = (int)anglef[0];
            }
            else {
                  ref[0] = 0;
            }
      }
      else
      {
            ref[0] = (int)pgm_read_word(&ref_roll[amostra])-saida[amostra];
            dy_dp = K_ext_inv*(int)anglef[0];
            //printf("%ld\t",(long int)(100*dy_dp));
            dJT_dp += ((saida[amostra]-
(int)pgm_read_word(&saida_ideal[amostra]))*dy_dp);
            //printf("%ld",(long int)(dJT_dp));
      }
      ref[1] = 0;
      ref[2] = 250;
      ref[3] = 0;
      if (transicao==0) {amostra++;}
      }
    pid[0]=((ref[0]-(int)anglef[0])*K_ext-gyroADC[0])*K_int;
    pid[1]=((ref[1]-(int)anglef[1])*2-gyroADC[1])*0.2;
    pid[2]=ref[3]-gyroADC[2];
}

void atualiza_pid(void) {
    int j;
    motor[0]=ref[2]+pid[0]; //-pid[1]-pid[2];
    motor[1]=ref[2]-pid[0]; //+pid[1]-pid[2];
    motor[2]=ref[2]-pid[0]; //-pid[1]+pid[2];
    motor[3]=ref[2]+pid[0]; //+pid[1]+pid[2];

    for(j=0;j<4;j++)
    {
        if(motor[j]>1000) motor[j]=1000;
        if(motor[j]<90) motor[j]=90;
        if(ref[2]<10) motor[j]=0;
    }
}

void atualiza_motor(void) {
    OCR2A = (motor[0]>>3)+125;
    OCR2B = (motor[1]>>3)+125;
    OCR1A = (motor[2]>>2)+250;
    OCR1B = (motor[3]>>2)+250;
}

void setup(void) {

    DDRB |= 1<<1; // pinMode(9,  OUTPUT);  //Saida PWM
    DDRB |= 1<<2; // pinMode(10, OUTPUT);  //Saida PWM
    DDRB |= 1<<3; // pinMode(11, OUTPUT);  //Saida PWM
    DDRD |= 1<<3; // pinMode(3, OUTPUT);  //Saida PWM

    DDRB |= 1<<5; // pinMode(13, OUTPUT);  //LED

    /* ENTRADAS NÃO PRECISA - Eh Padrao
     pinMode(2, INPUT);  //Saida PWM
```

```
     pinMode(4, INPUT);  //Saida PWM
     pinMode(5, INPUT);  //Saida PWM
     pinMode(6, INPUT);  //Saida PWM
     pinMode(7, INPUT);  //Saida PWM
     */

    //  Serial.begin(115200);

    cli(); //noInterrupts();           // disable all interrupts

    //Timer 0
    TCCR0A = 0;
    TCCR0B = 0;
    TCNT0  = 0;
    OCR0A = 159;            //10ms
    TCCR0A |= (1 << WGM01);   // CTC mode
    TCCR0B |= (1 << CS00);    // prescaler
    TIMSK0 |= (1 << OCIE0A);  // enable timer compare interrupt

    //Interrupcao RC
    PCICR |= 0x00;  // 0x04 para ativar entradas do controle remoto.
//Interrupt2 Portas
    PCMSK2 |= 0xF0;  //Interrupt2 Pinos


    sei(); // interrupts();

    //Timer 1 - PWM - dividido por 64
    TCCR1A = 0;
    TCCR1B = 0;
    TCNT1  = 0;
    OCR1A = 250;
    OCR1B = 250;

    TCCR1A |= (1 << COM1A1);   // Normal mode A (non-inverted)
    TCCR1A |= (1 << COM1B1);   // Normal mode B (non-inverted)
    TCCR1A |= (1 << WGM11);   // Fast PWM 9 Bits
    TCCR1B |= (1 << WGM12);   // Fast PWM 9 Bits
    TCCR1B |= (1 << CS11);    // 64 prescaler
    TCCR1B |= (1 << CS10);    // 64 prescaler

    //Timer 2 - PWM - dividido por 128
    TCCR2A = 0;
    TCCR2B = 0;
    TCNT2  = 0;
    OCR2A = 125;
    OCR2B = 125;
    TCCR2A |= (1 << COM2A1);   // Normal mode A (non-inverted)
    TCCR2A |= (1 << COM2B1);   // Normal mode B (non-inverted)
    TCCR2A |= (1 << WGM21);   // Fast PWM
    TCCR2A |= (1 << WGM20);   // Fast PWM
    TCCR2B |= (1 << CS22);    // 128 prescaler
    TCCR2B |= (1 << CS20);    // 128 prescaler

    //No timer 1 deve ser colocado o DOBRO do valor do timer 2 para mesma
resposta pois PRESCALER eh diferente
    //Timer 1 entre 250 e 500
    //Timer 2 entre 125 e 250

    uart0_init(MYUBRR);
    stdout = &uart_output;

    TWBR = ((F_CPU / 400000L) - 16) / 2; // change the I2C clock rate to
400kHz
    Gyro_init();
    ACC_init();
}

ISR(TIMER0_COMPA_vect)          // timer compare interrupt service routine
{
```

```
    tempo++;
    if(tempo>1000)
    {
        roda=1;
        tempo=0;
    }
}

ISR(PCINT2_vect)
{
    //Interrupcao Pinos para PC

    if ((PIND & MASK_P5) != P5)//if digital PIN5 has changed
    {
        if(P5)
        {
            pino[0]=tempo-pino[0];
            if (pino[0]<0)
            {
                pino[0]=pino[0]+1000;
            }
            ref[0]=(pino[0]-150)*10;

        }else{
            pino[0]=tempo;
        }
        P5 = PIND & MASK_P5;//Set new status
    }
    if ((PIND & MASK_P4) != P4)//if digital PIN4 has changed
    {
        if(P4)
        {
            pino[1]=tempo-pino[1];
            if (pino[1]<0)
            {
                pino[1]=pino[1]+1000;
            }
            ref[1]=(pino[1]-150)*10;

        }else{
            pino[1]=tempo;
        }
        P4 = PIND & MASK_P4;//Set new status
    }
    if ((PIND & MASK_P6) != P6)//if digital PIN6 has changed
    {
        if(P6)
        {
            pino[2]=tempo-pino[2];
            if (pino[2]<0)
            {
                pino[2]=pino[2]+1000;
            }
            ref[2]=(pino[2]-110)*10;

        }else{
            pino[2]=tempo;
        }
        P6 = PIND & MASK_P6;//Set new status
    }
    if ((PIND & MASK_P7) != P7)//if digital PIN7 has changed
    {
        if(P7)
        {
            pino[3]=tempo-pino[3];
            if (pino[3]<0)
            {
                pino[3]=pino[3]+1000;
            }
            //ref[3]=(pino[3]-150)*10;
```

```
            ref[3]=-(pino[3]-150)*10;

        }else{
            pino[3]=tempo;
        }
        P7 = PIND & MASK_P7;//Set new status
    }
}

void loop(void) {
    ocioso++;
    if(roda==1){
        roda=0;
        Gyro_getADC();
        ACC_getADC();
        if(armed==0)
        {
            if((ref[2]<10) & (ref[3]>450))      carmed+=2;

             if (dados==100) {passo = passo-0.01;
printf("\n%d\n",(int)(100*passo));}
             if (dados==102) {passo = passo+0.01;
printf("\n%d\n",(int)(100*passo));}

             if ((dados==105) & (ajuste==0))
             {
                    aux1++;
                    if(aux1==2) {aux1 = 0; ajuste = 1;}
             }

             if ((ajuste==1) & (exp_n==0)) {carmed = 100; exp_n = 1; amostra =
0; K_ext_inv = 1/K_ext;}

            if(carmed==100){
                armed=1;
                calibra_giro_i[0]=(int)calibra_giro[0];
                calibra_giro_i[1]=(int)calibra_giro[1];
                calibra_giro_i[2]=(int)calibra_giro[2];
            }

            motor[0]=0;
            motor[1]=0;
            motor[2]=0;
            motor[3]=0;
            PORTB &= ~(1<<5); //Desliga LED
            calibra_giro[0]=0.1*gyroADC[0]+0.9*calibra_giro[0];
            calibra_giro[1]=0.1*gyroADC[1]+0.9*calibra_giro[1];
            calibra_giro[2]=0.1*gyroADC[2]+0.9*calibra_giro[2];
        }
        else
        {
            if((ref[2]<10) & (ref[3]<-450))    carmed--;
             // Temporizador para desarmar o quad caso a letra 'c' não seja
recebida pela serial por 3s.
             if (dados==99) {sem_comando = 0;}
             else {sem_comando++;}

             angulos();
             controle();
             atualiza_pid();

             // Desarma caso 'p' seja pressionada OU o ângulo seja maior de
70° OU 'c' não seja recebida por 3s.
             if (((dados==112) & (ajuste==1)) | (anglef[0]>700) | (anglef[0]<-
700) | (sem_comando>=300))
             {
                    carmed = 0; ajuste = 0; exp_n = 0; sem_comando = 0;
transicao = 0;
                    if ((anglef[0]>700) | (anglef[0]<-700)) {anglef[0] = 0;}
                    printf("\nDes\n");
```

```
            }
            if ((amostra==TOTAL_AMOSTRAS))
            {
                  if (transicao==0) {}
                  if (exp_n==2)
                  {
                        ref[2] = 0;
                        motor[0]=0;
                        motor[1]=0;
                        motor[2]=0;
                        motor[3]=0;
                        atualiza_motor();
                        carmed = 0;
                        ajuste = 0;
                        exp_n = 0;
                        dJT_dp = dJT_dp/TOTAL_AMOSTRAS*2;
                        K_ext -= (passo*(dJT_dp/fabs(dJT_dp)));
                        printf("%d\t",(int)(100*K_ext));
                        J = 0;
                        for(k=0;k<TOTAL_AMOSTRAS;k++) {
                              erro = saida[k]-
(int)pgm_read_word(&saida_ideal[k]);
                              J += erro*erro/TOTAL_AMOSTRAS;
                              // printf("\nJ = %ld\terro = %ld\terro*erro =
%ld\n",(long int)J,erro,erro*erro);
                        }
                        printf("%ld\t",(long int)J);
                        dJT_dp = 0;
                        iter++;
                        printf("%d;\n",iter);
                        if ((iter==1) | (iter==5) | (iter==10) | (iter==15)
| (iter==20) | (iter==25) | (iter==30) | (iter==35) | (iter==40))
                        {
                              printf("Ref\tSaida\n");
                              for (k=0;k<TOTAL_AMOSTRAS;k++) {

      printf("%d\t",(int)pgm_read_word(&ref_roll[k]));
                                    printf("%d;\n",saida[k]);
                              }
                        }
                  }
                  if (exp_n==1)
                  {
                        transicao++;
                        if (transicao==500) {transicao = 0; exp_n = 2;
amostra = 0;}
                  }
            }
            if(carmed==0)
           {
               armed=0;
               calibra_giro_i[0]=0;
               calibra_giro_i[1]=0;
               calibra_giro_i[2]=0;
           }
           PORTB |= 1<<5; //Liga LED
        }
        atualiza_motor();

        dados=uart0_getc();
        ocioso=0;
    }
}
int main(void){
    setup();
    printf("\nK\tJ\titer\n");
    for(;;)
    { loop(); }
    return 0;
}
```