Instituto Federal do Rio Grande do Norte
Campus Natal - Central
Diretoria de Gestão e Tecnologia da Informação
Tecnologia em Análise e Desenvolvimento de Sistemas

Uma estrutura de controle de caminhada em um robô humanoide em C++

Jamillo G. da S. Santos

Natal-RN
2017

Jamillo G. da S. Santos

Uma estrutura de controle de caminhada em um robô humanoide em C++

Trabalho de conclusão de curso de graduação do curso de Tecnologia e Análise em Desenvolvimento de Sistemas da Diretoria de Gestão e Tecnologia de Informação do Instituto Federal do Rio Grande do Norte como requisito parcial para a obtenção do grau de Tecnologo em Análise e Desenvolvimento de Sistemas.

Linha de pesquisa:
Nome da linha de pesquisa

Orientador

Prof. Dr. Eduardo Braulio

TADS – Curso de Tecnologia em Análise e Desenvolvimento de Sistemas
DIATINF – Diretoria Acadêmica de gestão e Tecnologia da Informação
CNAT – Campus Natal - Central
IFRN – Instituto Federal do Rio Grande do Norte

Natal-RN

2017

Trabalho de Conclusão de Curso de Graduação sob o título *Uma estrutura de controle de caminhada em um robô humanoide em C++* apresentada por Jamillo G. da S. Santose aceita pelo Diretoria de Gestão e Tecnologia da Informação do Instituto Federal do Rio Grande do Norte, sendo aprovada por todos os membros da banca examinadora abaixo especificada:

Nome completo do orientador e titulação
Presidente
DIATINF – Diretoria Acadêmica de Gestão e Tecnologia da Informação
IFRN – Instituto Federal do Rio Grande do Norte

Nome completo do examinador e titulação
Examinador
Diretoria/Departamento
Instituto

Nome completo do examinador e titulação
Examinador
Diretoria/Departamento
Universidade

Natal-RN, data da defesa (dia, mês e ano).

Dedico este trabalho à minha filha Iasmin, por ser a minha maior motivação e inspiração para ser melhor a cada dia.

Agradecimentos

À minha esposa, filha, irmão e meus pais, pelo apoio incomensurável.

Ao meu orientador, pela paciência, liberdade e confiança.

Ao Sr. Mojtaba Karimi, por compartilhar seus *insights* e seu trabalho.

Ao Sr. Hosseinmemar, pela sua sincera amizade e o apoio viabilizando alguns testes remotos no robô real.

Ao Prof. Phd. Jacky Baltes, pelas valiosas lições, competições internacionais e as diversas portas abertas.

Aos meus amigos Nirvana, Adolfo, Duarte, e outros, que me ajudaram no processo da escrita deste trabalho, corrigindo desde pequenos erros de português até dicas valiosas no entendimento geral do trabalho.

Aos professores Plácido Neto e Cláudia Ribeiro pelas afortunadas dicas durante a escrita deste trabalho.

A satisfação reside no esforço, não no resultado obtido. O esforço total é a plena vitória.

Mahatma Gandhi

Uma estrutura de controle de caminhada em um robô humanoide em C++

Autor: Jamillo G. da S. Santos
Orientador(a): Prof. Dr. Eduardo Braulio

Resumo

A robótica está cada vez mais avançada e acessível. Seja em um pequeno robô aspirador de pó autônomo ou nos sofisticados robôs que automatizam a fabricação de carros, aumentando a produção e diminuindo seu custo de produção. Entretanto, em suas formas muito variadas, nem sempre se adequam aos ambientes projetados para os humano. Por isso, a pesquisa com robôs humanoides tem significativa importância para nossos dias. Nesse trabalho foi utilizado o robô Arash, humanoide com 1m de altura e 7,5 kg de peso. Esse robô funciona a partir de controles em software, implementados em paradigma estruturado. O objetivo desse trabalho é reimplementar uma parte desse controle, o *walking gait*, em linguagem orientada a objetos. O novo Walking gait possui um esquema de configurações baseado em objetos JSON salvos em arquivos. Desta forma, é possível inicializar o componente com diversos parâmetros diferentes. Também é possível o versionamento dos arquivos de configuração, para que alterações sejam mantidas de forma rastreável. Para demonstrar a viabilidade dessa abordagem um conjunto de verificações foram realizadas e mostrou a eficácia do novo *walking gait*.

Palavras-chave: Robô, Humanoide, Controle, Walking Gait, Caminhada.

Título do trabalho (em língua estrangeira)

Author: Jamillo G. da S. Santos
Supervisor: Prof. Doc. Eduardo Braulio

Abstract

The robotics field is increasingly advanced and accessible. Whether it is a small standalone vacuum cleaner robot or the sophisticated robots which automate car manufacturing, increasing production and lowering your production cost. However, in their very varied forms, they do not always suit the environments designed for humans. Therefore, research with humanoid robots has significant importance for our days. In this work the Arash robot was used, a humanoid robot 1m tall and 7.5 kg of weight. This robot controlled using software implemented in a structured paradigm. The goal of this work is to reimplement one part of this control, the walking gait, in an object oriented language. The new Walking gait has a configuration scheme based on JSON objects saved into files. In this way, it is possible to initialize the component with several different parameters. It is also possible to version the configuration files so that changes are kept traceable. To demonstrate the feasibility of this approach a set of checks were performed and showed the effectiveness of the new walking gait.

Keywords: Robot, Humanoid, Control, Walking Gait, Walking.

Lista de figuras

Lista de abreviaturas e siglas

CoM – Centro de massa, do inglês *Center of Mass*

ZMP – do inglês *zero moment point*

ARASH – *Anthropomorphic Robot Augmented with Sense of Human*, em livre tradução Robô Antropomórfico Aumentado com Sentido de Ser Humano

DOF – Do inglês *Degrees Of Freedom*, ou graus de liberdade

IK – conhecido como *Inverse Kinematics*, ou cinemática inversa

IA – Inteligência artificial

IMU – do inglês *Inertia Measurement Unit*

IDE – do inglês *integrated development environment*

PID – do inglês *Porportional Integral Derivative*

TTL – Do inglês *transistor-transistor logic*

JSON – *Java Script Object Notation*

Sumário

1 Introdução

A robótica está cada vez mais avançada e acessível. Seja em um pequeno robô aspirador de pó autônomo - que até mesmo retorna à estação de recarregamento quando sua bateria está fraca - ou nos sofisticados robôs que automatizam a fabricação de carros, aumentando a produção e diminuindo o custo. E eles vem em todos os tamanhos e formas.

Vivemos em um mundo feito e projetado para humanos. De escadas, portas, e até ferramentas como furadeiras, ou até o mouse são pensados para o uso diário de ser humano. Desta forma, nada mais obvio que projetar um robô humanoide que adapte-se de forma natural a este ambiente. Esta forma, porém, impõe diversos desafios ao controle.

Em um esforço mútuo, as universidades *Amirkabir University of Technology* (AUT), do Irã, e a *University of Manitoba* (UofM), do Canadá, trabalham juntas para participar da Robocup - competição internacional de futebol de robôs cujo o objetivo é derrotar a seleção campeã mundial na copa de 2050. Nas edições 2015 e 2016, realizada em Hefei (China) e Leipzig (Alemanhã), respectivamente, o sistema que controlou a caminhada de *Arash* – um robô humanóide de 100cm de Altura – rendeu o 3º lugar na modalidade de futebol em ambos os anos. Também, os 2º e 1º lugar no modalidade do desafio técnico em 2015 e 2016.

Desenvolvido utilizando placa microcontroladora *OpenCM 9.04*, compatível com Arduino, e batizado de *AUT-UofM-Walk-Engine*, o sistema de caminhada, ou *walking gait*, foi um dos grandes responsáveis pelo bom desempenho nas competições. Entretando, o fato de ser executado na *OpenCM9.04* traz uma série de consequências não desejáveis ao palco.

Inicialmente, sendo um sistema complexo, a caminhada deve ser configurada para cada tipo de superfície. As configurações de uma superfície lisa e áspera, como concreto, podem ser bastante diferentes das configurações para caminhar num terreno como a grama artificial utilizada na Robocup. Essas mudanças nas configurações são determinantes para o bom equilíbrio do robô. Com mais de 100 parâmetros, esse processo de configuração

requer muito tempo e paciência.

O primeiro problema com a implementação atual é que a configuração se dá através de alterações diretas dos valores dos parâmetros no código-fonte. Em seguida, a nova versão é compilada e enviada à *OpenCM9.04*. Na sequência, inicializa-se o robô e verifica-se a caminhada.

Além de massante, este esquema de atualização não favorece uma forma razoável de manter as alterações dos parâmetros organizada. Há uma frequente perda na determinação de quais valores de parâmetros são melhores para qual tipo de situação. Uma solução seria criar subversões do sistema para cada cenário, o que desfavorece a correção de *BUGs* e melhorias, já que cada modificação deve ser refletida em várias versões. Uma segunda possibilidade seria criar uma interface de comunicação entre um computador, que funcionaria como um gerenciador, e o *walking gait* via USB. Assim, o computador poderia guardar arquivos de configurações que seriam enviados à placa microcontroladora a medida que fossem necessários, o que seria uma saída simples e eficaz, mas não definitiva, tendo em vista os desafios que serão citados adiante nesse texto.

O segundo problema com o *walking gait* atual é a dificuldade de implementação de simulações, testes e depuração. A única forma de obter visualização da saída gerada é através do *console serial* da IDE de programação – o que torna o processo de depuração um exercício de abstração com visualização de ângulos em 3 dimensões – ou arriscar-se aplicando a saída de teste direto aos motores de alto custo, confiando que tudo sairá bem. A ativação da simulação neste cenário é possível. Porém, haveria a dependência do dispositivo *OpenCM9.04*, implicando também em algumas dificuldades técnicas na hora de testes de distúrbio em um ambiente simulado, sendo que a placa microcontroladora está ligada aos sensores reais.

O terceiro problema é a forma em que o paradigma estruturado foi utilizado. Já que o processo realizado pelo *walking gait* já não é trivial, uma má organização do código implica em uma baixa curva de assimilação do seu funcionamento. Como solução é proposta a aplicação de um paradigma orientado a objetos, dividindo cada subprocesso em diferentes componentes encapsulados.

O quarto problema é a natureza multi-processada do sistema. Ao mesmo tempo em que ângulos devem ser enviados aos motores, a leitura dos sensores de orientação devem ser processadas e levadas em consideração para as próximas iterações. No *OpenCM0.04*, esse paralelismo é habilitado através da biblioteca *MapleFreeRTOS*. O fato de já haver muito processamento para rodar o sistema atual limita possíveis futuras melhorias a serem

desenvolvidas, melhorias estas que poderia levar à próxima geração de robôs ainda mais estáveis.

Por fim, o quinto problema – que também limita o desenvolvimento da próxima geração da caminhada – é o tamanho da memória *ROM* da placa microcontroladora ondeo programa compilado do *walking gait* é guardado. Atualmente, o *firmware* da *OpenCM9.04* já foi modificado pela equipe da AUTUofM, via remoção de funções e bibliotecas não utilizadas, para diminuir seu tamanho e assim liberar espaço para a versão atual do sistema.

Dados os problemas encontrados no funcionamento do sistema atual e as considerações listadas até agora, este trabalho propõe mover a execução do componente *walking gait* do microcontrolador *OpenCM09.04* para o controlador principal, um computador Linux rodando Ubuntu 14.04. Para o projeto da solução, foi levada em consideração dois aspectos importantes: o desempenho e a interface de comunicação com os motores, decidindo-se assim pela adoção da linguagem *C++* versão 14.

Para sumarizar, este trabalho tem como objetivo a implementação de uma versão funcional do *walking gait* usando o trabalho realizado por Karimi *et al* (KARIMI; SADEGHNEJAD; BALTES, 2016), melhorando o código atual, de forma a diminuir a curva de aprendizado e aumentando as possibilidades para a implementação de futuras melhorias.

1.1 Organização do trabalho

Este trabalho está organizado, além da introdução, em mais três capítulos, Organização do onde são abordadas a Matemática envolvida no movimento do robô Arash (capítulo 2), a trabalho Arquitetura do sistema computacional e a proposta da nova arquitetura (Capítulo 3) e as conclusões (Capítulo 5).

2 A caminhada de Arash

A caminhada de robôs com pernas vem sendo desenvolvida desde da década de 70, quando Kato e Vukobratovic praticamente iniciaram este campo de estudo (KAJITA; ESPIAU, 2008).

E no escopo do desenvolvimento de um *walking gait* é possível duas estratégia de equilíbrio: estática ou dinâmica. Cada estratégia tem suas vantagens e desvantagens. Todavia, o equilíbrio dinâmico vem sendo mais aceito dentre os pesquisadores.

O equilíbrio estático mantém o centro de massa, ou CoM – do inglês *Center of Mass* – sempre sob a superfície de apoio do robô. Uma desvantagem desse modelo é que as velocidades alcançadas são bem lentas, tendo em vista que o movimento produzido para a caminhada não deve gerar inercia suficiente para deslocar o CoM fora da área esperada. Geralmente observa-se pés maiores em robôs que adotam este tipo de caminhada (MILLER, 1994).

Ainda, segundo Miller, o equilíbrio dinâmico, considera a dinâmica do movimento da caminhada, onde o ZMP, do inglês *zero moment point*, sempre fica sob a superfície de apoio. Desta forma, habilita-se o CoM a movimentar-se para fora do ponto de apoio, mantendo o robô em uma espécie de "queda controlada". Assim, ela provê maiores velocidades de caminhada com maior eficiência. O ZMP é o ponto de apoio com o chão onde nenhum momento é criado no eixo horizontal (KAJITA; ESPIAU, 2008).

Durante qualquer tarefa envolvendo locomoção é necessária uma maneira de controlar o *walking gait* de forma a direcionar o robô ao destino escolhido. Uma das abordagens utilizadas são movimentos pré-definidos que executam ações específicas. Por exemplo, dar um passo a frente, um passo lateral à esquerda, girar 30°à direita. Assim, essas ações são sequenciadas de forma a levar o robô até seu destino. A desvantagem deste método é cada ação deve ser iniciada e finalizada antes que a próxima seja acionada e isto diminui a agilidade do robô.

Uma segunda forma de controle é a chamada caminhada omnidirecional que consiste

em controlar o robô por velocidades em diferentes eixos. Essas mudanças podem ser realizadas com o robô em movimento, sem a necessidade de encadeamento de ações. Além disso, existe a possibilidade de combinar movimentos em diferentes eixos ao mesmo tempo criando novas possibilidades de locomoção.

Neste trabalho, a abordagem de Karimi *et al* é usada como base para o desenvolvimento do novo *walking gait*. Eles descrevem seu método como uma caminhada omnidirecional com equilíbrio dinâmico.

3 A matemática dos movimentos

3.1 Arash e sua estrutura

ARASH (*Anthropomorphic Robot Augmented with Sense of Human*, em português Robô Antropomórfico Aumentado com Sentido de Ser Humano), é um robô de 7,5 Kg e 1 metro de altura.

Como controlador principal, um computador *mini-box* é utilizado em conjunto com uma placa microcontroladora *OpenCM9.04*, que serve de interface entre o controlador principal e os motores (SADEGHNEJAD et al., 2016). Adicionalmente, um sensor *IMU InvenSense MEMS MPU* integrado com 9 eixos (KARIMI; SADEGHNEJAD; BALTES, 2016) está anexado em suas costas para fornecer informações sobre a orientação do torso diretamente à placa microcontroladora.

As partes que ligam os motores de Arash foram esculpidas de blocos sólidos de alumínio e em suas juntas estão presentes motores de alto desempenho projetados para robôs pela empresa sul-coreana Robotis Co.

Possuindo 20 graus de liberdade (ou DOF , do inglês *degrees of freedom*), ele tenta imitar a forma humana, tanto em suas juntas quanto em suas proporções. Os DOFs são o número de parâmetros de posições independentes que definem a configuração de uma sistema mecânico (CRAIG, 1989) e tão importante quanto sua configuração é a sua orientação para que os cálculos funcionem como o esperado. A inversão de qualquer orientação entre o modelo matemático e o modelo real causará uma inversão durante a caminhada, causando reações inesperadas e possíveis danos aos motores.

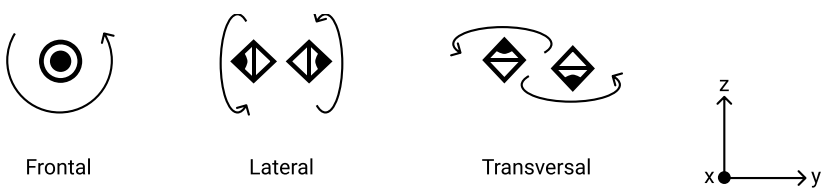

Figura 1: Orientação dos movimentos dos atuadores

A figura 1 exibe as possíveis orientações dos atuadores que são referenciadas ao longo do trabalho como frontal, lateral e transversal.

Atuadores com orientação frontal – ou simplesmente, atuadores frontais – propiciam rotação orientada pelo eixo X, que está aponta para o leitor (para fora do papel). Esse tipo de rotação também é referida como *roll*.

Atuadores laterais apontam para direita ou esquerda. Eles apresentam rotação com base no eixo Y (plano XZ), também conhecida como rotação *pitch*. Analogamente, atuadores transversais apontam para cima ou baixo, com base no eixo Z (plano XY), com movimento de rotação conhecido como *yaw*.

Na figura 2, observa-se o esquema de orientação dos atuadores aplicada no diagrama das juntas de Arash. Ainda na mesma figura, os "triângulos pretos preenchidos" representam apenas o fim de uma cadeia de atuadores. Assim, estes símbolos representam as mãos, que não possuem atuadores, e a ponta da cabeça, onde encontra-se a câmera.

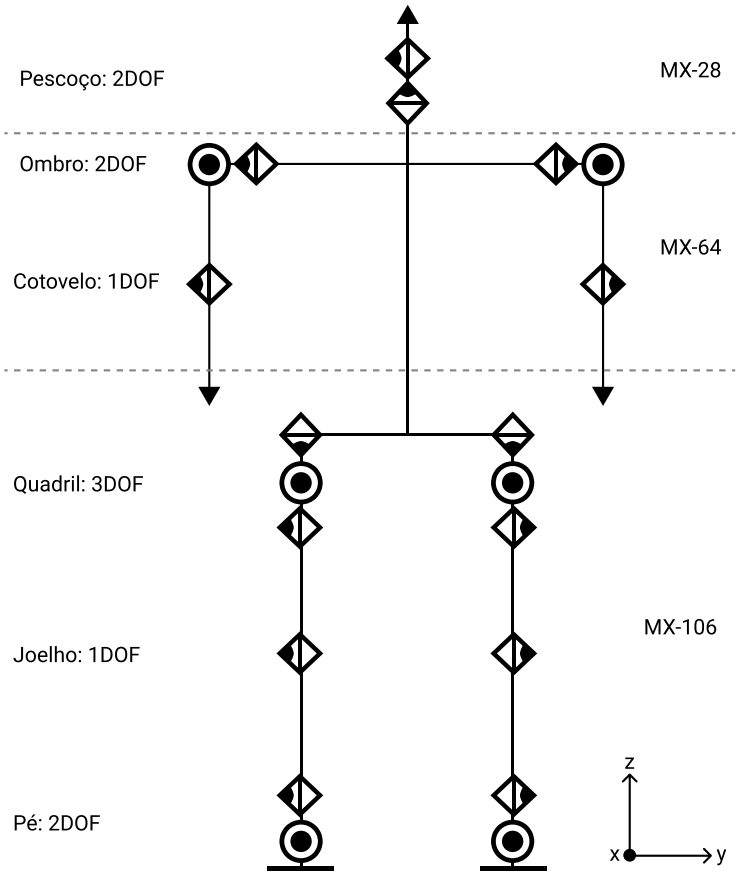

Figura 2: Diagrama de orientação dos atuadores de Arash

Em Arash, todos os atuadores utilizados são produzidos pela Robotics.Co. Porém, devido a variação de carga nas diversas juntas, foram utilizados diferentes modelos da série MX. Esta decisão de projeto diminui bastante o custo final do robô.

No pescoço, onde a carga é bem leve, "MX-28" são suficientes. Dois atuadores, um na posição transversal (o atuador mais baixo) que fornece o movimento panorâmico a cabeça e outro na posição horizontal, fornecendo o movimento de inclinação vertical da cabeça.

Nos braços, que podem sofrer uma carga maior, atuadores "MX-64" são utilizados. Isso é importante já que existem modalidades de competições, como o levantamento de peso, que testam a capacidade do carregamento de cargas.

Para as pernas, foram utilizados atuadores "MX-106" que são mais poderosos que os anteriores. Entretanto, na fase de projeto, simulações mostraram que durante o movimento de levantar-se do chão (em caso da recuperação de uma possível queda), o torque nas juntas do joelho era levado ao máximo suportado pelo "MX-106", podendo assim levar este motor à falha. Desta forma, 2 atuadores sincronizados passaram a formar esta junta, afim de compartilhar a carga sofrida entre dois motores. Esta junta dupla não oferece nenhum impacto na implementação, já que os motores da série "MX-106" oferecem a capacidade de serem ligados e sincronizados via *hardware*. Assim, a nível de software controla-se apenas 1 único atuador. Desta forma, apesar das 20 DOF, Arash possui 22 motores.

3.2 Visão geral do processo

Para caminhar, Arash move suas juntas de uma forma sincronizada mantendo o equilíbrio. É um processo contínuo onde cada passo é planejado e possíveis perturbações são compensadas, o mais breve possível, afim de preservar a trajetória e evitar quedas.

O processo de caminhada é dividido em ciclos. Cada ciclo é dado pelo movimentos de passo da perna direita seguido pelo movimento da perna esquerda. O caminho percorrido pelos pés, suas posição e rotação, é predeterminado pelo *walking gait* e é referido como trajetória.

A cada momento no tempo do ciclo – levando em consideração também outras variáveis – a trajetória fornece a posição e rotação de ambos os pés de Arash. O tempo de cada ciclo é definida por um relógio central definido no intervalo exibido na equação 3.1, sendo t o tempo do ciclo atual da caminhada.

$$0 \leq t < 2\pi \tag{3.1}$$

Dentro do tempo do relógio central, existe a frequência que os movimentos são gerados, identificado por $Gait_{frequency}$. Este parâmetro indica a quantidade de posições geradas por

segundo.

No conceito de caminhada omnidirecional são utilizadas 3 velocidades afim de controlar a direção e velocidade do robô. V_x é a velocidade em direção à frente, com valores positivos para a frente e negativos para caminhar de costas. V_y é a velocidade lateral com onde passos para o lado são definidos com valores positivos implicando em movimentos laterais à direita. V_θ é a velocidade de rotação por passo com valores positivos indicando rotação também à direita.

Na figura 3 é possível observar o diagrama com as posições dos pés e torso de Arash. Em azul estão ambos os pés e torso de Arash no início do ciclo de caminhada, quando t é 0. Em cinza, vê-se o pé direito e sua posição projetada quando t vale π (ou seja, metade do ciclo). Nota-se o deslocamento frontal em V_x, um deslocamento lateral em V_y e um deslocamento angular em V_θ do torso, bem como do pé direito (afetado pela primeira metade do ciclo).

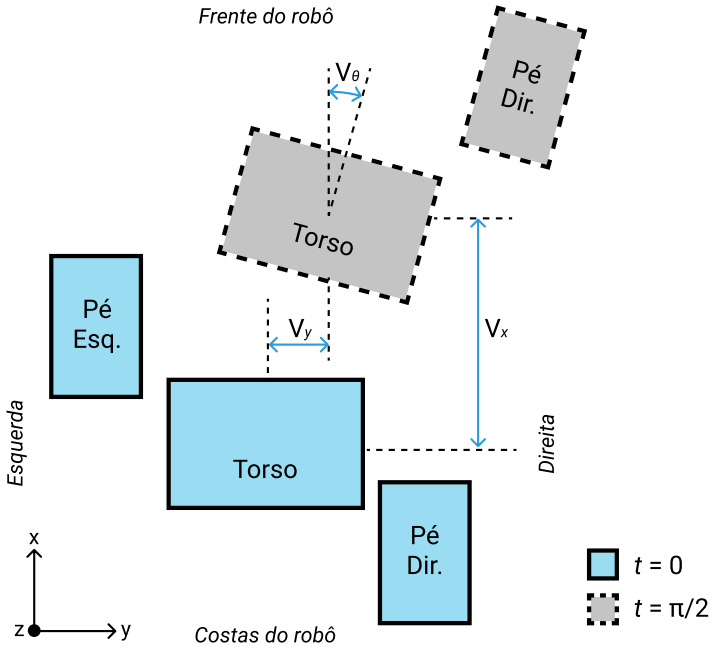

Figura 3: Diagrama da direção das velocidades da caminhada omnidirecional

Como mostra as equações 3.2, valores normalizados entre -1.0 e 1.0 são utilizados para indicar as velocidades. Adotando-se 0.0 como velocidade nula, ou sem movimento, 1.0 como valor de velocidade máxima e -1.0 como valor de velocidade máxima no sentido oposto. Valores normalizados foram adotados para abstrair valores de velocidades absolutos já que futuras aplicações deste trabalho podem utilizar diferentes configurações de robôs. Desta forma, o controlador de comportamento pode abstrair a configuração do robô que está sendo controlado e enviando velocidades relativas. Adicionalmente, valo-

res normalizados provém a integração perfeita entre as funções trigonométricas para a definição das trajetórias senoidais adotadas por *Karimi et al.*

$$-1.0 \leq V_x \leq 1.0 \quad , \quad -1.0 \leq V_y \leq 1.0 \quad , \quad -1.0 \leq V_\theta \leq 1.0 \tag{3.2}$$

A cada iteração dois vetores T, de transferência, e R, de rotação, de cada pé são obtidos. Para o pé direito, utiliza-se o t corrente no relógio central, já à perna esquerda um atraso de fase de $\pi/2$ é inserido.

A equação 3.3 mostra a definição de dos vetores T e R. Cada um de seus elementos representam um eixo dentro deles e são gerados pelas trajetórias que são detalhadas na seção 3.4.

$$T = [Foot_x, \quad Foot_y, \quad Foot_z], \quad R = [Foot_{roll}, \quad Foot_{pitch}, \quad Foot_{yaw}] \tag{3.3}$$

Em poder desses vetores, utiliza-se os conceitos da cinemática inversa para gerar os ângulos de cada junta. Finalmente, os ângulos gerados são enviados aos motores – como especificado na subseção 4.3.3 – e, por fim, a iteração é finalizada.

3.3 Orientação

Durante a implementação deste trabalho, nenhuma modificação foi realizada no componente de orientação originalmente desenvolvido por Karimi *et al.* Entretanto, para a implementação do cálculo da trajetórias é necessário algumas informações sobre a saída deste componente.

No final do processamento, obtém-se como saída um vetor com aceleração linear nos eixos x e y, como mostra a equação 3.4. Então, o vetor *Accel* é utilizado para o cálculo da detecção de "empurrões", ou distúrbios, como visto nas equações 3.5 e 3.6, que representação a variação na aceleração linear entre a iteração atual e a anterior.

$$Accel = [Accel_x \quad Accel_y] \tag{3.4}$$

$$Push_x = Accel[x]_t - Accel[x]_{t-1} \tag{3.5}$$

$$Push_y = Accel[y]_t - Accel[y]_{t-1} \tag{3.6}$$

3.4 Geração da trajetória

Como afirmado anteriormente, a trajetória define a posição e rotação de ambos os pés de Arash durante cada momento do ciclo de caminhada, gerando como saída dois vetores T e R, como definido na equação 3.3. Karimi *et al* definem as funções que determinam cada eixo dos vetores, da equação 3.7 até 3.12.

$$Foot_x = \left(\frac{-cos(t)+1}{2}\right) \times \left(\frac{tanh(V_x + Push_x) + H_{leg}}{\pi}\right) \tag{3.7}$$

$$Foot_y = \big((-sin(t) \times B_{swing}) + (cos(t)+1)\big) \times \left(\frac{tanh(V_y + Push_y) + H_{leg}}{2\pi}\right) \tag{3.8}$$

$$Foot_z = \left(\frac{sin(t)}{t+1/2}\right) \times \left(\frac{tanh(V_x + V_y) \times H_{leg}}{\pi}\right) \tag{3.9}$$

$$Foot_{roll} = 0 \tag{3.10}$$

$$Foot_{pitch} = 0 \tag{3.11}$$

$$Foot_{yaw} = \left(\frac{sin(V_\theta)+1}{2}\right) \times \left(\frac{-cos(t)+1}{2} \times V_\theta\right) \tag{3.12}$$

Onde t representa o tempo atual normalizado em cada ciclo; B_{swing} representa a quantidade de compensação de balanço lateral que o corpo deve efetuar para compensar a dinâmica do movimento das pernas; $Push$ – definido na seção 3.3 – é a quantidade de distúrbio causado por desvios na trajetória; por fim, H_{leg} é o valor do tamanho da perna, extraído da estrutura do robô.

Nota-se que as funções $Foot_{roll}$ e $Foot_{pitch}$ são definidas como 0 pois, em resultados experimentais, os efeitos durante a caminhada mostraram-se desprezíveis (KARIMI; SADEGHNEJAD; BALTES, 2016).

3.5 Cinemática Inversa

Descobrir a posição de um parte do robô a partir dos ângulos de suas juntas, *forward kinematics*, é uma tarefa fácil e pode ser realizada eficientemente. Porém, o oposto – dado um ponto e calcular os ângulos das juntas que o alcancem; processo conhecido como IK (do inglês *inverse kinematics* ou cinemática inversa) – não é uma tarefa tão simples.

Enquanto a problemas envolvendo a *forward kinematics* possuem uma solução única, na *IK* podem haver múltiplas soluções ou nenhuma (SPONG; HUTCHINSON; VIDYASAGAR, 2005). Pode-se observar este fato ao tocar a ponta do próprio nariz, já que existem diversas combinações de posições do ombro, cotovelo, pulso e falanges que resolvem este problema; Entretando, usando as configurações de braço de um Tiranossauro o problema não teria solução, uma vez que o braço é demasiado curto para efetuar tal tarefa.

Aplicando a teoria da cinemática inversa ao problema proposto, Karimi *et al* apresenta uma solução direta aproveitando-se das limitações impostas pela configuração das juntas de Arash.

$$P = [Hip_{yaw}, Hip_{roll}, Hip_{pitch}, Knee, Foot_{pitch}, Foot_{roll}] = iK(T, R) \tag{3.13}$$

Na equação 3.13, vemos a definição do vetor P que mantém os valores dos ângulos de todas as juntas. Ele é a saída da função iK que recebe como parâmetros os vetores T e R já definidos na equação 3.3.

O processo de formulação das equações inicia-se a partir do nó principal do quadril descendo para o fim da cadeia, nos pés; onde encontra-se o fim da cadéia e o ponto de destino.

Na primeira etapa, a figura 4 mostra uma tomada de cima de Arash auxiliando na visualização da equação 3.14. Nota-se que, embora os pés de Arash não possuam rotação no eixo *yaw* (figura 2) a rotação *yaw* aplicada no quadril, afeta toda a cadeia.

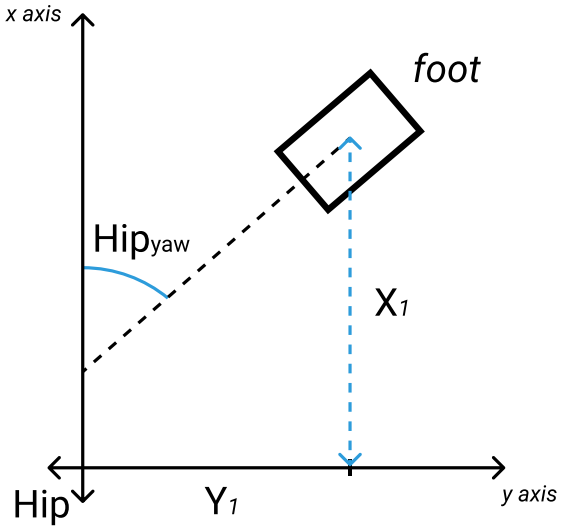

Figura 4: Diagrama da visão superior da estrutura de Arash que representa da equação 3.14 até 3.17

Fonte: (KARIMI; SADEGHNEJAD; BALTES, 2016)

$$P[Hip_{yaw}] = R_{yaw} \tag{3.14}$$

$$X_2 = T_x cos(R_{yaw}) + T_y sin(R_{yaw}) \tag{3.15}$$

$$Y_2 = -T_x sin(R_{yaw}) + T_y cos(R_{yaw}) \tag{3.16}$$

$$Z_2 = T_z \tag{3.17}$$

Seguindo o processo de parametrização dos ângulos das juntas, observa-se na figura 5 a visão frontal de Arash provendo uma visualização geométrica das equações 3.18 até 3.22. Nesta figura observa-se, também, o movimento no eixo *roll* do quadril sendo compensado pelo movimento no sentido oposto (mas no mesmo eixo) do pé, afim de compensar o alinhamento do pé com o piso.

$$P[Hip_{roll}] = atan\left(\frac{Y_2}{Z_2}\right) \tag{3.18}$$

$$P[Foot_{roll}] = -P[Hip_{roll}] + R_{roll} \tag{3.19}$$

$$X_3 = X_2 \tag{3.20}$$

$$Y_3 = Y_2 \tag{3.21}$$

$$Z_3 = \sqrt{{Y_2}^2 + {Z_2}^2} \tag{3.22}$$

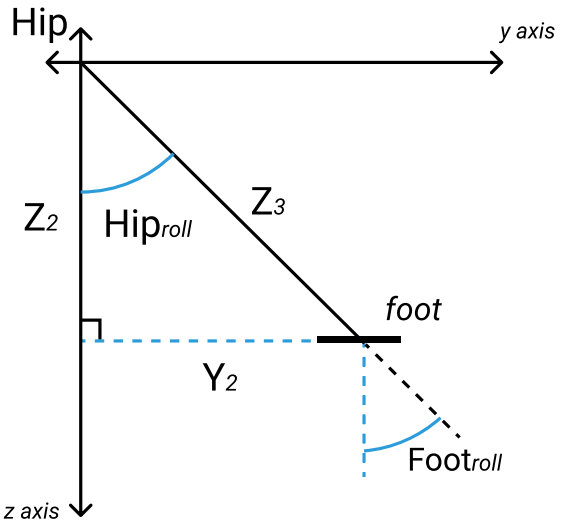

Figura 5: Diagrama da visão frontal de Arash que representa da equação 3.18 até 3.22

Fonte: (KARIMI; SADEGHNEJAD; BALTES, 2016)

Na terceira etapa, a figura 5 mostra a visão frontal de Arash, também, provendo uma visualização geométrica das equações 3.18 até 3.22.

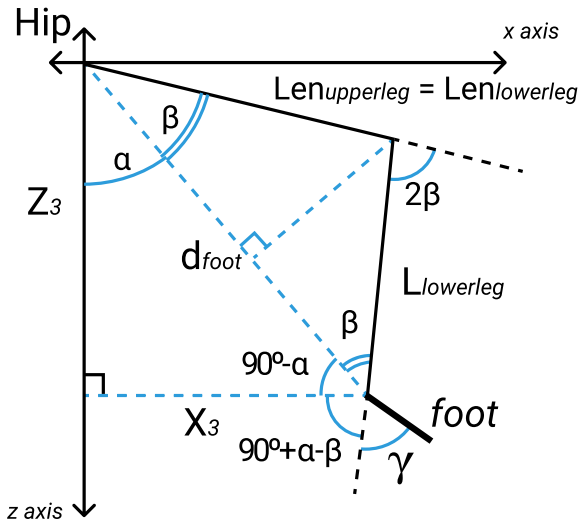

Figura 6: Diagrama da visão lateral de Arash que representa da equação 3.18 até 3.22

Fonte: (KARIMI; SADEGHNEJAD; BALTES, 2016)

$$d_{foot} = \sqrt{{Z_3}^2 + {X_3}^2} \tag{3.23}$$

$$\alpha = atan\left(\frac{X_3}{Z_3}\right) \tag{3.24}$$

$$\beta = acos\left(\frac{d_{foot}}{2 \times Leg_{len}}\right) \tag{3.25}$$

$$P[Hip_{pitch}] = \alpha + \beta \tag{3.26}$$

$$P[Knee] = -2\beta \tag{3.27}$$

$$P[Foot_{pitch}] = \gamma = -\alpha + \beta + R_{pitch} \tag{3.28}$$

Adicionalmente, a cada junta é somado um parâmetro de deslocamento – não incluso nas equações de *IK* –, também conhecido como *offset*, para compensar possíveis erros provenientes de desalinhamentos no momento de montagem do robô, ou folgas nas engrenagens dos atuadores.

Finalmente, com as equações de *IK* definidas, o *walking gait*, com o auxílio dos componentes de orientação e *proxy*, é apto a fazer os cálculos necessários para determinar os movimentos de caminhada.

4 Arquitetura

A robótica tenta solucionar problemas envolvendo diversas áreas da computação. Desde o controle de motores, usando programação de baixo nível, até soluções sofisticas utilizando IA (Inteligência artificial) aplicada a visão e tomada de decisões, entre outros. Assim, o desenvolvimento de uma única ferramenta, em uma única linguagem, que solucione todos os problemas pode tornar-se impraticável o estágio atual da tecnologia, já que as linguagens são projetadas com intenções e problemas diferentes em mente.

Para solucionar este problema, a arquitetura adotada pelo time AUT-UofM é distribuída, baseada no *framework* ROS. Onde cada componente tem uma função diferente dentro do sistema total. O ROS é uma coleção de bibliotecas e convenções para simplificar a tarefa de criar comportamentos complexos na grande variedade de robôs e suas plataformas (ROS, 2017).

Desta forma, este capítulo divide cada subsistema dentro do robô como componentes apresentando suas funções antes e após implementadas as mudanças propostas pelo trabalho em suas diferentes camadas de execução. Alguns deles são mais detalhados e descritos em seus módulos, apresentando alguns fluxogramas e diagramas afim de expressar melhor seu funcionamento.

4.1 Componentes e arquitetura atual

A arquitetura do sistema pode ser abstraída em 3 camadas: A camada de *hardware*, a de software de baixo nível, e a de software.

Na camada de *hardware*, a mais inferior na figura 7, estão contidos os componentes físicos do sistema. Uma câmera *Webcam Logintech C920 Pro HD 15MP Full HD1080p* é usada para fornecer imagens para o sistema de visão. Uma *IMU* , do inglês *Inertia Measurement Unit*, é usada para fornecer informações para o posterior cálculo da orientação do corpo do robô em relação ao eixo de gravidade. E os motores, também chamados de

atuadores, são utilizados nas juntas para os movimentos às partes do robô.

A camada de software de baixo nível é responsável pelo processamento de dados da *IMU*, juntamente com a execução do *walking gait*. Esta camada roda dentro da placa microcontroladora *Robotis.Co OpenCM9.04*, ou simplesmente *OpenCM9.04*.

A placa controladora conta com um processador *32bit ARM Cortex-M3*, memória *flash* (que mantém os dados mesmo quando desligada) de 128Kb e 20Kb de *SRAM* (ROBOTIS, 2016). A placa tem seu projeto e código fonte abertos e foi utiliza a *Arduino IDE* como IDE de desenvolvimento. Ela conecta-se ao controlador principal via USB, por onde há troca de dados. Adicionalmente, devido ser produzida pela mesma empresa, a placa integra-se perfeitamente com os motores utilizados no projeto de Arash.

Os motores, todos pertencentes a categoria *Dynamixel*, são atuadores inteligentes otimizados para robôs. Eles possuem controle PID integrado e comunicar-se em uma rede de comunicação TTL (*transistor-transistor logic*) com *baudrate* de até 4.5 Mbps (ROBOTIS, 2017).

A camada de software é executada dentro de um computador, doravante referido como controlador principal, *mini-box PC MAXData QutePC-3001* com um processador 64 bits de dois núcleos Intel© Celeron© 847E (1,1GHz e 2MB de *cache*), rodando o sistema operacional Linux Ubuntu versão 14.04. Esta camada é responsável por rodar os componentes de software, de alto nível, e enviar comandos de controle para a camada inferior, de baixo nível – figura 7.

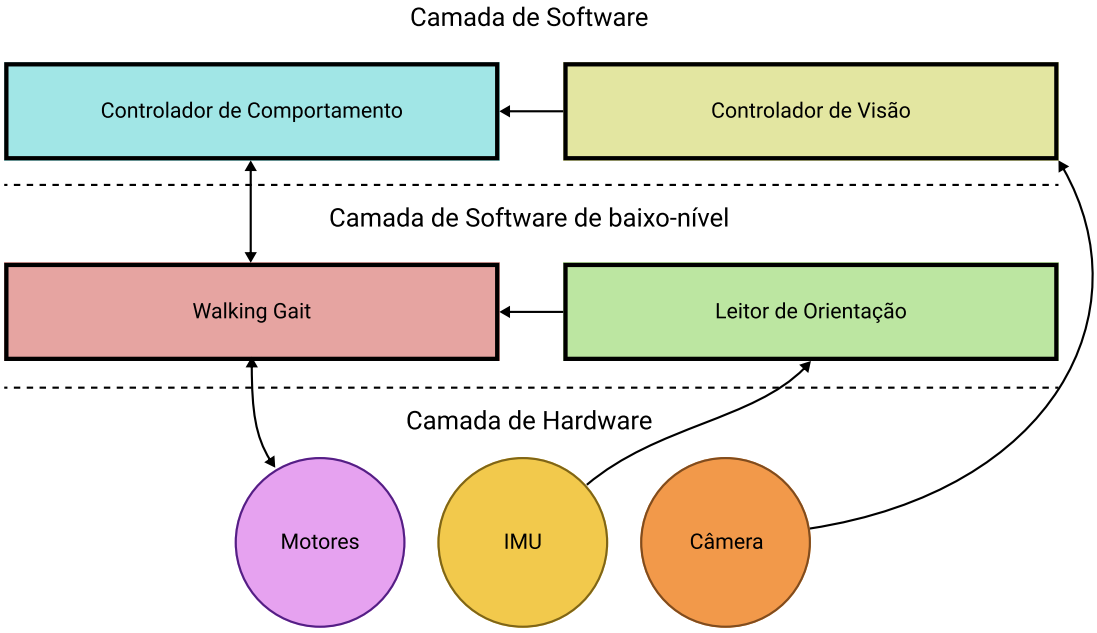

Figura 7: Fluxograma simplificado da arquitetura atual e seus fluxos.

A figura 7 mostra uma visão simplificada dos principais componentes – e o sentido do fluxo de dados entre eles – envolvidos no processo de uma tarefa genérica hipotética. O funcionamento de cada componente individual será detalhado nas próximas seções.

4.1.1 Visão

Visão não é o foco deste trabalho. Entretanto, uma explanação geral é necessária afim de melhor entender o funcionamento geral do sistema.

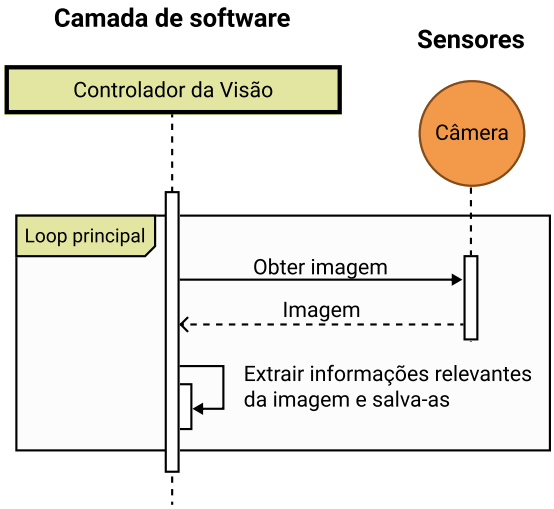

Figura 8: Diagrama de sequência simplificando do *loop* principal do componente de visão.

O controle de visão é um software que conecta-se via USB com a câmera, analisando as imagens e extraindo informações relevantes à execução da tarefa do robô. De forma geral, podemos abstrair seu funcionamento em duas *threads* distintas:

A primeira é o *loop* principal, representada no diagrama de sequência na figura 8. Em um primeiro passo, o controlador obtém a imagem da câmera. Em seguida, a extração de informações é realizada. Essas informações dependem da tarefa que o robô deve realizar. Por exemplo, detectar a bola, os adversários e as linhas do campo, em caso de da tarefa ser uma partida de futebol; ou detectar as linhas guia da pista de corrida no caso da tarefa ser uma corrida. Em seguida, tudo o que foi encontrado é salvo para futuras consultas até que a próxima iteração ocorra e os dados sejam atualizados.

A segunda *thread*, cuja a interação pode ser visualizada na figura 9, do controle de visão executa o sistema de comunicação que pode ser implementado de várias maneiras, desde objetos *JSON* sendo transmitidos via *UDP* até um *message broker* com filas de consumo. Ao receber a mensagem de solicitação das informações, responde serializando os dados que ficaram salvos na primeira *thread*, fornecendo as informações com atraso

mínimo.

Neste trabalho, nenhuma implementação relacionada a visão foi produzida.

4.1.2 Controlador de Comportamento

O controlador de comportamento roda na camada de *software* e implementa a tarefa que deve ser realizada pelo robô. Ele funciona como um gerente que coordena outros componentes, recebendo informações e enviando comandos com ações específicas serem a executadas. Na figura 9 é possível observar a sequência de passos realizados pelo controlador de comportamento.

Usando como exemplo uma partida de futebol, na figura 9 observa-se o controlador de comportamento "jogador" consultando o componente de visão, que responde a posição da bola, previamente obtida durante o processo da figura 8. Em seguida, baseado nessas informações o comportamento envia o comando ao *walking gait* para caminhar na direção correta.

Normalmente este processo é contínuo. A detecção é realizada, as informações são processadas, a decisão é tomada, o comando é enviado e o sistema aguarda o início da próxima iteração, onde tudo será repetido.

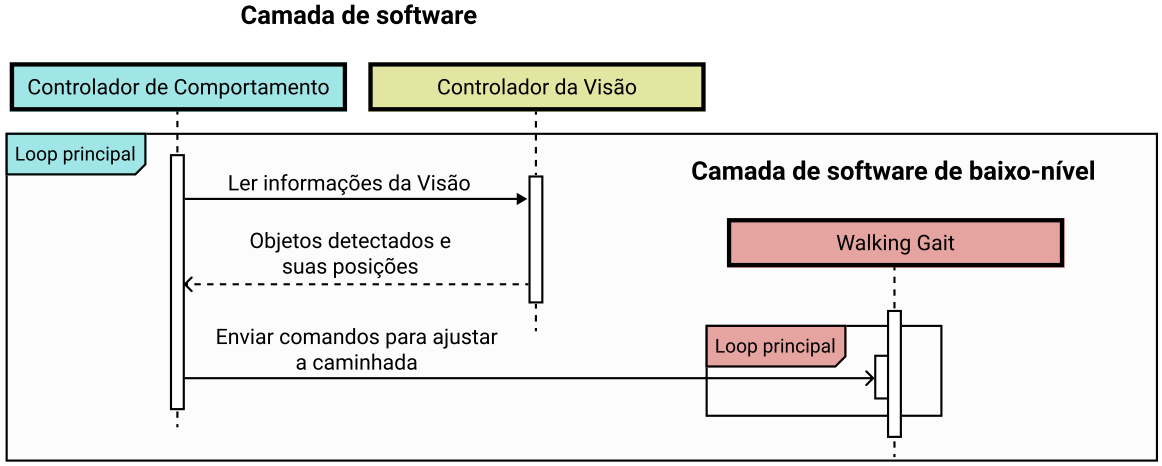

Figura 9: Diagrama de sequência do controlador de comportamento e suas dependências.

4.1.3 *Walking Gait* e Leitor de Orientação

O componente *walking gait* coordena e executa a caminhada. Ele monitora a USB aguardando os comandos de controle que contém as velocidades da caminhada, enviados a partir do controlador de comportamento.

O componente leitor de orientação desempenha um papel importante ao lado do *walking gait*. Ele é o responsável pela verificação da orientação da rotação do torso – uma vez que a *IMU* está situada nas costas de Arash – assim fornecendo dados para o *walking gait* fazer correções durante a caminhada afim de compensar um eventual distúrbio.

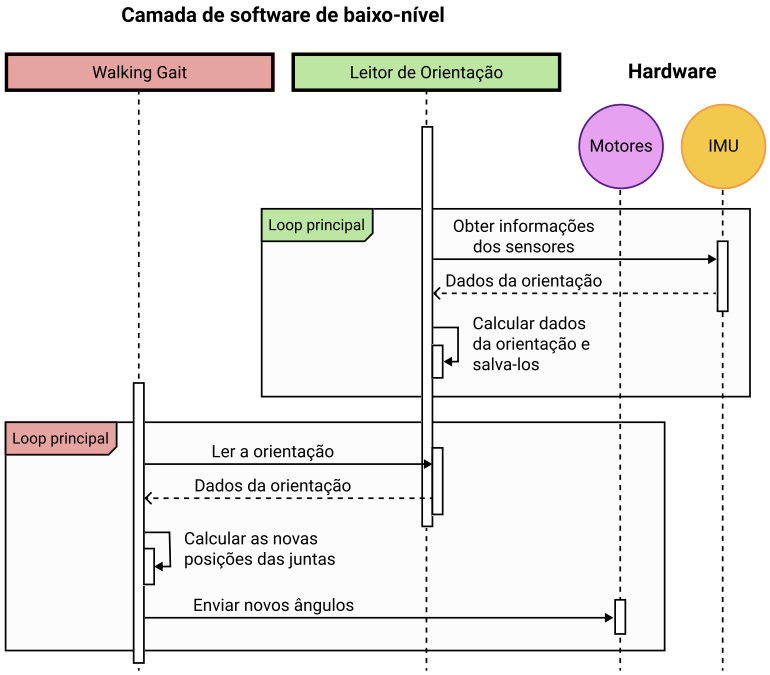

Figura 10: Diagrama de sequência simplificando do *walking gait* e do leitor de orientação.

Ainda no leitor de orientação, os dados são coletados da *IMU* e um processo de utilizando filtro de *Kalman* é utilizado para obter uma estimativa mais acurada das orientações que são salvas em variáveis. Já na *thread* que roda o *walking gait*, a orientação é continuamente consultada e utilizada para que os ângulos das juntas sejam atualizados. O filtro de Kalman é um algoritmo que – dada uma séria de observações no tempo e suas incertezas – é capaz de, estatisticamente, calcular um valor que tende a ser mais próximo ao real (GREWAL; ANDREWS, 2014).

Dentro da placa microcontroladora *OpenCM9.04*, o *walking gait* e o leitor de orientação rodam em *threads* dinstintas, recurso este habilitado pela biblioteca *MapleFreeRTOS*. Na figura 10 observa-se a execução de ambas a *threads* separadamente e a forma de interação entre elas. Iniciando a partir do *walking gait*, onde os dados de orientação são lidos diretamente do leitor de orientação, em seguida calcula-se os ângulos das juntas que são enviadas aos motores. Em paralelo, o leitor de orientação continua recebendo os dados da *IMU*, processando e salvando-os para que fiquem disponíveis ao *walking gait*. Nota-se que, em ambas as *threads*, as sequências de ações são executadas dentro de *loops* principais.

Para sumarizar, foram apresentados os componentes, da arquitetura atual, e o seu

papel dentro do sistema completo. Seus fluxos de dados e uma base de como seu processamento ocorre também foi mostrado.

4.2 A nova arquitetura

De acordo com os problemas e solução já discutidos em seções anteriores, esta seção irá apresentar as mudanças realizadas na arquitetura original apresentada na seção 4.1.

Uma das principais mudanças realizadas neste trabalho é a re-implementação do *walking gait*. Nessas modificações ainda faz-se necessário a a utilização da *OpenCM9.04*. Entretanto, ela assume o papel de reencaminhar os dados dos ângulos aos motores e, ainda, fornecer os dados de orientação ao *walking gait*.

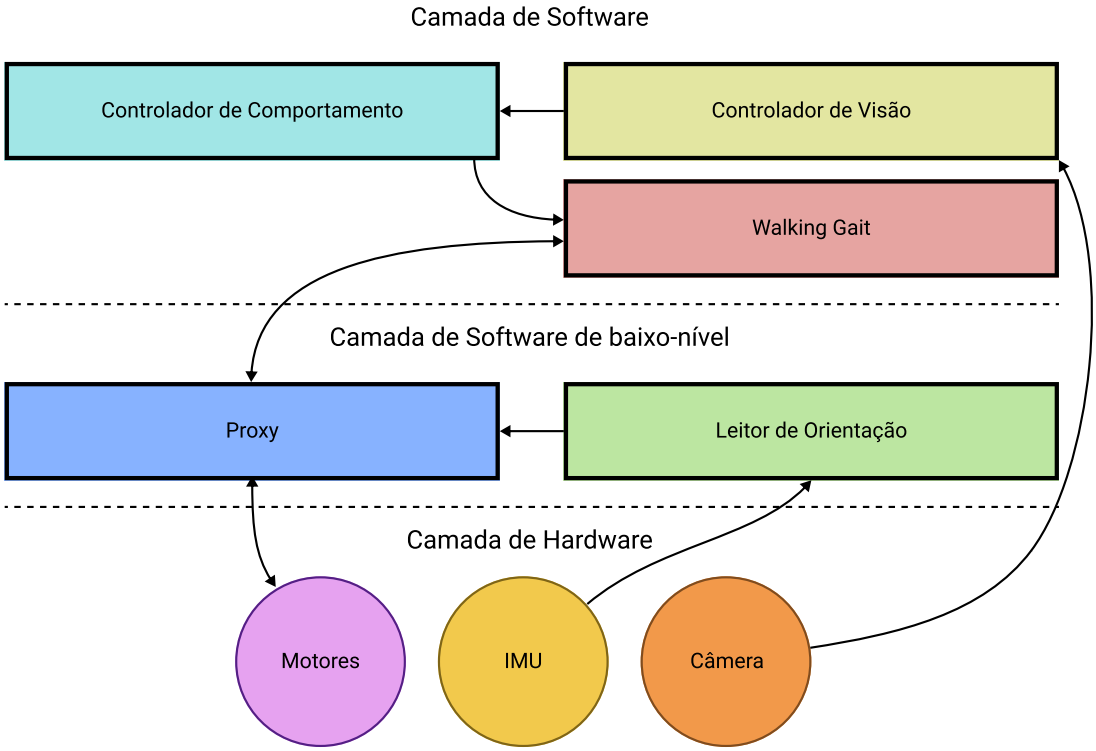

Figura 11: Fluxograma simplificado da nova arquitetura e seus fluxos de dados.

A figura 11 mostra a visão simplificada da nova arquitetura. Nela observa-se o *walking gait* – agora na camada de *software* – funcionando dentro do controlador principal, e um novo componente chamado *proxy* é apresentado como interface os componentes de *hardware* e o *walking gait*. O papel do novo componente é de servir realmente como um *proxy* enviando comandos recebidos do controlador principal aos motores e também receber a orientação do componente leitor de orientação e fornecê-la ao controlador principal.

O controlador de comportamento continua a enviar os comandos de controle ao *walking*

gait. Porém, ao invés de usar comandos binários via USB, agora utiliza objetos serializados via JSON por UDP (seção 4.3.1).

Entre o *walking gait* e o *Proxy* a comunicação ocorre via USB @ 1 Mbps. Comandos de controle utilizando o *dynamixel protocol 2.0* são gerados e enviados diretamente do *walking gait* ao *Proxy*, que apenas os encaminha aos motores. Esta estratégia foi adotada para habilitar a possibilidade da utilização de motores de diferentes tipos, não apenas da Robotis Co. Assim, apenas o *walking gait* é responsável pela geração de destes comandos específicos de cada motor enquanto o *Proxy* apenas os encaminha.

Na camada de *software* de baixo nível, o leitor de orientação continua a fornecer os dados de orientação. Porém, ao invés do *walking gait* acessá-los diretamente, o *Proxy* encarrega-se de disponibiliza-los ao *walking gait*.

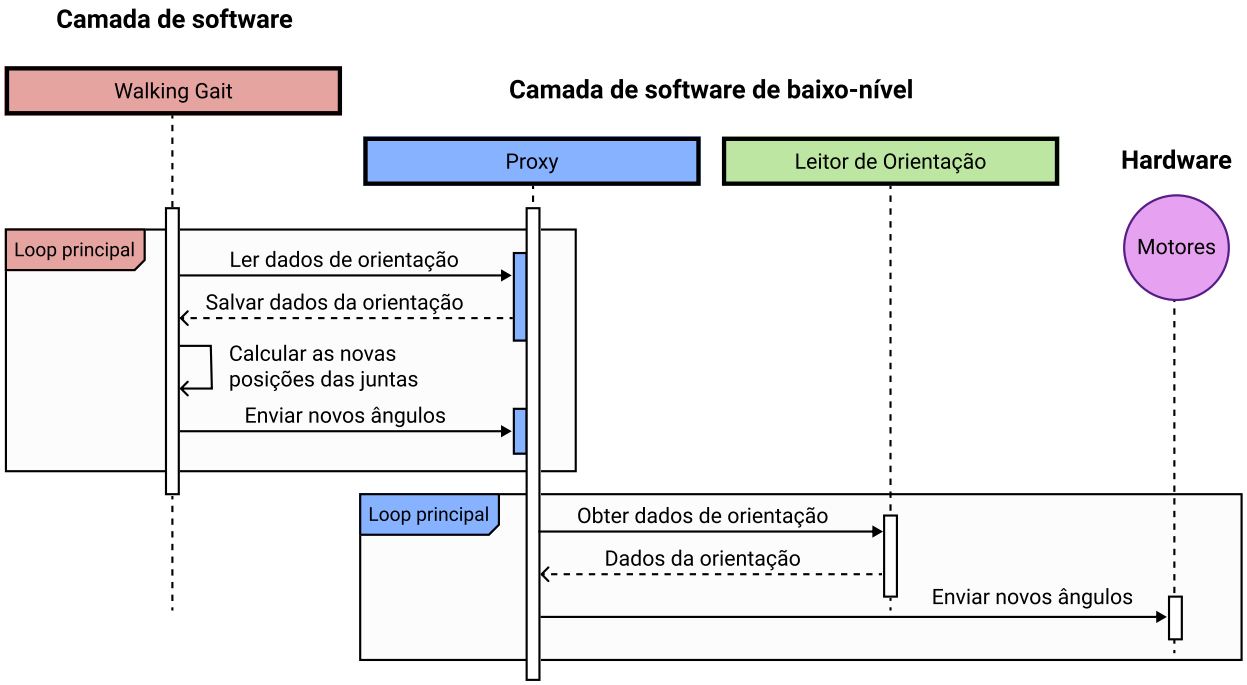

Figura 12: Diagrama de sequência simplificado mostrando a interação entre o *walking gait* e os componentes de baixo nível.

Na figura 12, é possível observar o diagrama de sequência com os componentes que foram atingidos pela mudança na arquitetura. O *walking gait* lê os dados de orientação, calcula as novas posições das juntas e envia os novos ângulos ao *Proxy* e processo volta ao início. Enquanto isso, o *Proxy* obtém os dados de orientação diretamente do Leitor de orientação, em seguida os fornece ao *walking gait*.

4.3 *Walking gait*

A nova implementação do *walking gait*, uma mudança completa paradigma estruturado ao orientado a objetos foi realizada. Afim de facilitar o desenvolvimento, o projeto foi divido em 3 submódulos: o servidor de controle, o motor de caminhada e o atualizador de juntas.

Nesta seção, os três submódulos são apresentados e, por fim, um software de apoio chamado visualizador 3D é apresentado.

Adicionalmente, todos os parâmetros de configuração – tais como as portas do de controle do visualizador 3D e servidor de controle (descritos nas próximas seções), deslocamentos físicos das juntas (*offsets*), peso de compensação do processo de *swing* do corpo (B_{swing}), a frequência de geração dos moimentos ($Gait_{frequency}$), o *walking gait* possui um esquema de configurações baseado em objetos JSON salvos em arquivos. Desta forma, é possível inicializar o componente com diversas parâmetros diferentes. Assim, também é possível o versionamento dos arquivos de configuração, para que alterações sejam mantidas de forma rastreável.

4.3.1 Servidor de controle

O servidor de controle é um submódulo que é implementa a interface de comunicação do *walking gait* a outros componentes do sistema. De fato, todo o gerenciamento da caminhada pode ser realizada a partir de um único comando que atualizam as velocidades V_x, V_y e V_θ.

```
{
  "command": "omniwalk",
  "params":
  {
    "x": 0.3, // Opcional: Velocidade no eixo X
    "y": 0.6, // Opcional: Velocidade no eixo Y
    "theta": 0.0  // Opcional: Velocidade no eixo teta
  }
}
```

Figura 13: Comando de controle exemplo, em JSON.

Para tal tarefa, adotou-se uma solução simplista que consiste em receber objetos serializados *JSON* através de um servidor *UDP*, principalmente por causa de sua rápida implementação.

Sabe-se que o protocolo UDP não garante a integridade, nem a ordem, da entrega dos dados. Todavia, a aplicação inteira roda dentro da rede de *loopback* dispensa essa preocupação. Também, como o processo é iterativo, caso alguma falha ocorra em uma iteração, ela logo será corrigida na próxima iteração. Adicionalmente, para esta aplicação, a não verificação dos pacotes pelo protocolo UDP, reverte-se em vantagem, já que não há o atraso de tempo necessário para sincronizar as duas pontas da comunicação. Além disso, esta abordagem aproveita-se da filosofia *zero-copy* adotada pelo *kernel* do Linux (TIANHUA et al., 2008) que evita fazer cópia dos *buffers* das mensagens recebidas via rede, desde o *driver* de rede até a entrega na aplicação.

Na figura 13 observa-se um comando de controle JSON. A propriedade *command* que identifica a intenção do comando de alterar os parâmetros da caminhada. A propriedade *params* indica um sub-objeto *JSON* com as propriedades x, y e *theta* representanto, respectivamente, V_x, V_y e V_θ. Todas as propriedades são opcionais, e caso não sejam informadas, nenhuma modificação será feita naquele eixo faltante.

Em caso de intenção de parada, o mesmo comando *omniwalk* deverá ser enviado porém com todas as velocidades com valor 0.0.

4.3.2 Motor de caminhada

O motor de caminhada é o submódulo onde os processos dos cálculo das trajetórias, e da IK serão executados e guardados para, posteriormente, serem aplicados aos motores pelo submódulo atualizador de juntas (subseção 4.3.3).

A classe *Humanoid* é usada para a manter o estado das juntas, tanto posição quanto angulações. Ela também é usada para manter os estados das juntas durante os cálculos da *IK*.

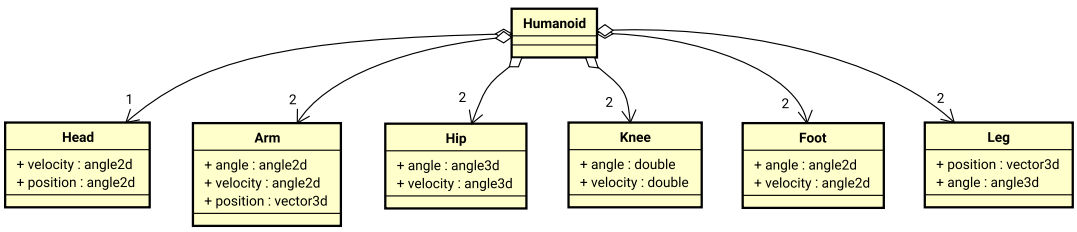

Figura 14: Diagrama de domínio do *walking gait*.

A figura 14 mostra o diagrama de domínio das classes mais relevantes aos dados que o componente manipula. Observa-se todas as composições envolvidas na definição da classe

Humanoid, que representa todo o conjunto das juntas de Arash. Também, nota-se que nas classes *Head*, *Arm*, *Hip*, *Knee*, *Foot* existem dados de ângulo e velocidade. Isso dá-se por causa que componente utiliza essas informações para calcular o próximo estado do sistema em caso de algum distúrbio. Também, a classe *Leg* aparece de forma ambígua, já que existem as definições de *Hip*, *Knee* e *Foot*. Porém, a classe *Leg*, representa os dados da cinemática inversa de uma perna, enquanto as demais classes guardam o estado atual das partes que representam.

Ainda na figura 14, a notação dos dados utilizados: *angle*2*d*, *angle*3*d* e *vector*3*d*. Estes tipos possuem o sufixo numérico, que indica a quantidade de dimensões aquela classe guarda, e a letra, que indica o tipo de dado, no caso *d* é referente a *double*. Para ângulos, 2 dimensões significa que os eixos de rotação *pitch* e *roll* são guardados; já os ângulos com 3 eixos, as orientações *roll*, *pitch* e *yaw* são guardadas. Para vetores, os valores nas dimensões x, y e z são mantidos.

O processo de caminhada inicia-se no momento em que qualquer velocidade é diferente de zero. Quando isso ocorre (ou seja, um comando de controle é recebido), a instância da classe *Robot*, doravante referida apenas como *robot*, através de seu método *omniGait*, faz uma interpolação linear entre a velocidade atual e a nova velocidade. A medida que a velocidade é gradualmente alterada para seu novo valor, o relógio central é iniciado e o método *updateTrajectories* é chamado, calculando a trajetória (de acordo com a seção 3.4) e guardando-a na variável de instância *humanoid*, do tipo *Humanoid*. Enfim, o método *update* de *Ik* é chamado para que execute os cálculos da cinemática inversa. O diagrama de sequência simplificado na figura 15 mostra este processo.

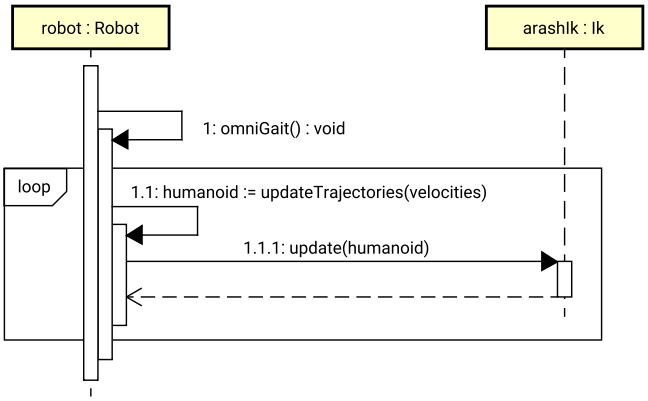

Figura 15: Diagrama de sequência simplificado do motor de caminhada.

4.3.3 Atualizador de juntas

Este módulo é responsável pela atualização dos ângulos dos motores. Ele roda em sua própria *thread*, afim de não bloquear o processamento dos demais componentes do *walking gait*.

A implementação base do atualizador de juntas, usa uma abstração de motores afim de prover a possibilidade do uso de motores de fabricantes diferentes em futuras aplicações.

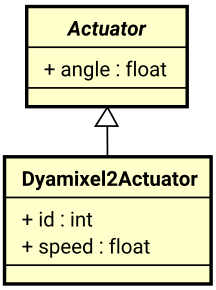

Figura 16: Fragmento do diagrama de classe da hierarquia dos atuadores.

A classe abstrata *Actuator*, base da implementação dos atuadores, define a propriedade de posição, referida na classe como *angle*, comum a todos os atuadores. Cada tipo de atuador, pode ter diferentes características e parâmetros de acordo com seu fabricante, embora todos tenham o mesmo objetivo.

Construída sobre a classe *Actuator* está a classe *Dynamixel2Actuator*. Ela adiciona as propriedades *id* e *speed*. O *id* é a identificação numérica de cada junta, solução adotada pela Robotis para identificar cada atuador dentro da rede TTL. Já a *speed* é a velocidade que a posição é alcançada. Todos os atuadores utilizados em Arash funcionam usando o mesmo protocolo, apesar das diferentes versões. Assim, todos são representados por instâncias desta classe.

A figura 16 mostra o diagrama de classe com a hierarquia e os principais métodos e propriedades das classes *Actuator* e *Dynamixel2Actuator*.

A classe *ActuatorUpdater* implementa uma iteração completa do processo de atualização das juntas e a figura 18 mostra este processo. Para mante-lo rodando continuamente, uma *thread* mantém o método *update* sendo chamando continuamente. A frequência de atualização é sincronizada com a geração dos dados pelo módulo do motor de caminhada.

Na figura 17 é possível observar as principais classes envolvendo o processo de atualização. A classe *ActuatorUpdater* é responsável por enviar os dados calculados pelo motor de caminhada aos atuadores. Porém, ela é uma classe abstrata e serve apenas como im-

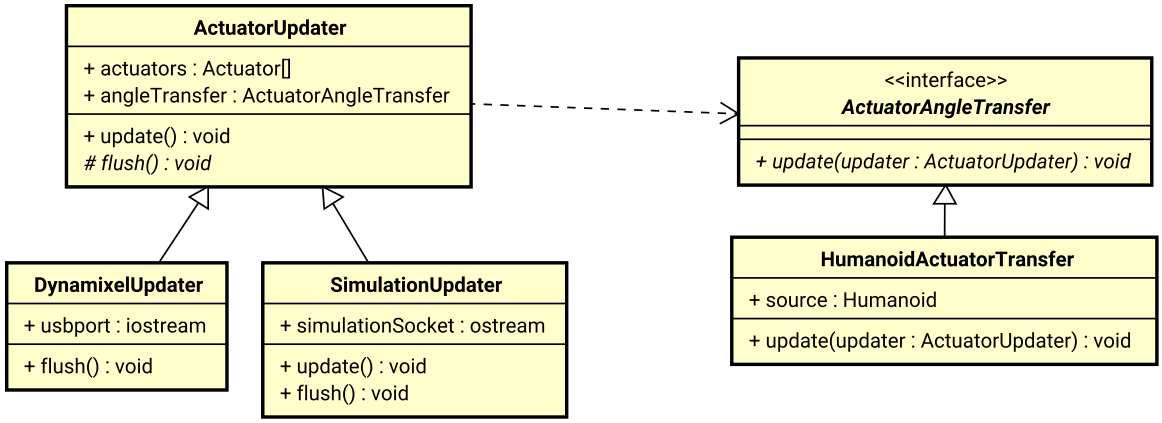

Figura 17: Fragmento do diagrama de classe da hierarquia dos atualizadores de atuadores.

plementação básica para as reais implementações. A classe concreta *DynamixelUpdater*, filha de *ActuatorUpdater*, define o método $flush$ implementando os comandos *Dynamixel 2.0* (ROBOTIS, 2017) para os motores reais. A classe concreta *SimulationUpdater*, também filha de *ActuatorUpdater*, implementa a comunicação para um sistema de visualização, descrito nas próximas seções, que habilita a visualização dos movimentos do robô usando um ambiente 3D facilitando a visualização dos movimentos.

Pensando em futuras implementações, criou-se a classe abstrata $ActuatorAngleTransfer$ que é responsável por conhecer a estrutura do robô e quais atuadores são as equivalentes as juntas. Assim, quando seu método *update* é chamado, este transfere todos os ângulos das juntas, situados em sua propriedade *source*, às suas respectivas instâncias de *Actuator* correspondentes. A propriedade *source* é uma instância da classe $Humanoid$ que é atualizada pelo motor de caminhada ao final do processamento da sua iteração.

Acompanhando o diagrama de sequência da figura 18, podemos ver o método *update* de *ActuatorUpdater* sendo chamado. Na sequência, o método *update* de *humanoidAngleTransfer* é chamado passando a própria instância de *ActuatorUpdater*, onde todos os dados das juntas são transferidos de *source* (propriedade de *humanoidAngleTransfer*) para os atuadores contidos na propriedade *actuators* de *updater*. Assim, finalmente, sendo chamado o método $flush$ de *ActuatorUpdater* que implementa a verdadeira comunicação com o motor.

Retornando à figura 17 observa-se duas implementações de *ActuatorUpdater*. A primeira implementação, a classe *Dynamixel2Updater*, implementa o protocolo *Dynamixel 2.0* e faz a comunicação via USB com o componente *Proxy*. Após os dados serem enviados ao *Proxy*, este encaminhará os dados aos motores que executam o comando posicionando as juntas.

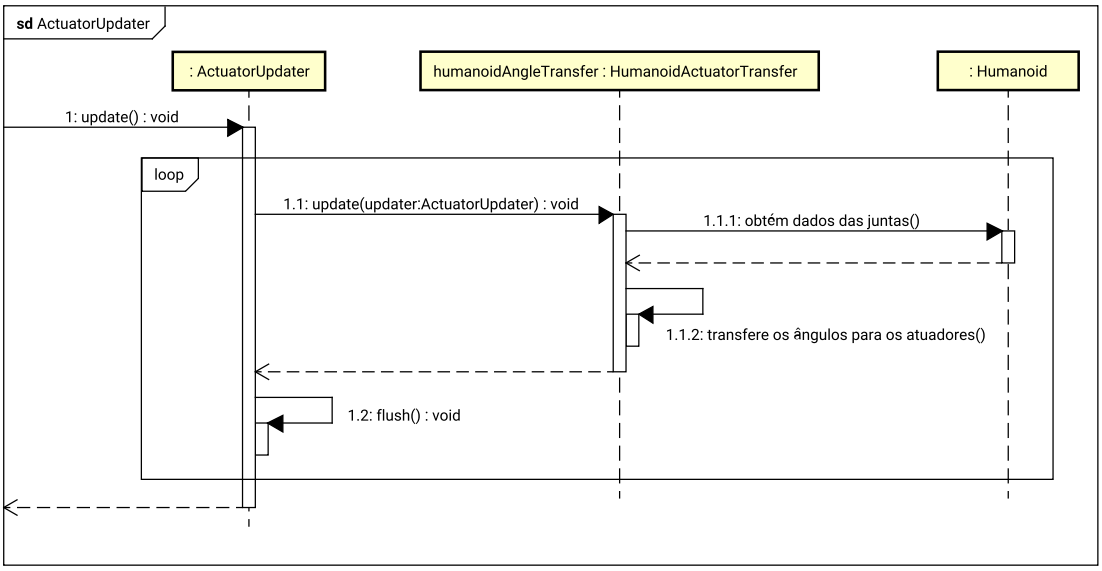

Figura 18: Fragmento do diagrama de classe da hierarquia dos atuadores.

A segunda implementação, a classe *SimulationUpdater*, envia as informações do atuador ao visualizador 3D. Ao envio é realizado através da serialização das informações em um objeto JSON e transmitido via UDP, de forma similar ao servidor de controle implementado no *walking gait*.

A figura 19 mostra um pacote de atualização de exemplo com as propriedades das juntas e seus ângulos. Nota-se a propriedade *params* com um objeto JSON com suas chaves definindo os nomes das juntas recebendo outros objetos com a propriedade *angle* com o valor dos ângulos em radianos.

```
{
  "command": "update",
  "params":
  {
    "RightHipLateral": { "angle": 0.1 },
    "RightHipTransversal": { "angle": 0.2 },
    "RightHipFrontal": { "angle": 0.3 },
    "RightKneeLateral": { "angle": 0.4 },
        ...,
    "LeftFeetLateral": { "angle": 0.5 },
    "LeftFeetFrontal": { "angle": 0.6 }
  }
}
```

Figura 19: Comando de controle recebido pelo visualizador 3D.

4.3.4 Visualizador 3D

As trajetórias geradas, juntamente com os 20 ângulos radianos das juntas, gerados pelo *walking gait* são difíceis de visualizar e validar. Desta forma torna-se necessária uma forma de visualizar, mesmo que de forma preliminar, o resultado; já que o acesso ao robô de forma física é impossível.

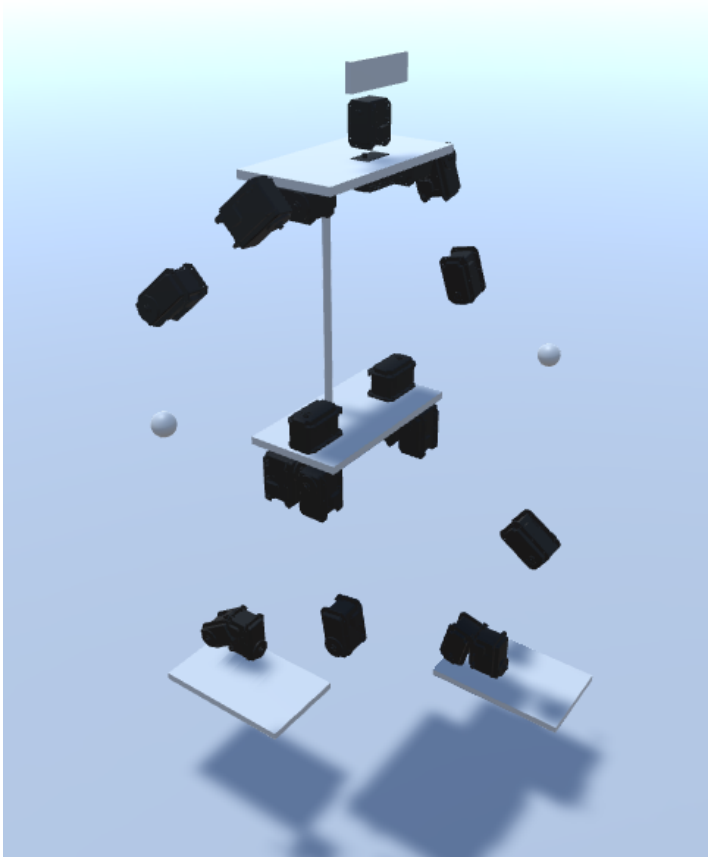

Figura 20: Captura de tela da imagem gerada pelo visualizador 3D.

A melhor solução seria a utilização de um *framework* de simulação, como Gazebo (GAZEBO, 2017). Onde seria possível validar toda o processo de caminhada. Todavia, por restrição de tempo impossibilitaram a adotação desta solução.

Desta forma, o visualizador 3D é uma ferramenta de apoio ao desenvolvimento criada com o objetivo de habilitar a interpretação visual dos ângulos e trajetórias gerados pelo *walking gait*.

Desenvolvido utilizando *framework* Unity 3D 5.1, comumente utilizado para desenvolvimento de jogos, o visualizador 3D possui a estrutura de Arash em uma cadeia de atuadores anexadas para simular as rotações geradas pelo *walking gait*. Entretando, sem efeitos físicos habilitados.

A comunicação entre o visualizador 3D e o *walking gait* é realizada pela troca de *JSON* via UDP, de forma similar ao servidor de controle (subseção 4.3.1). Cada pacote contém o estado completo de todas as juntas de Arash.

Ao receber os dados, o visualizador procura o atuador virtual correspondente e aplica a transformação na cadeia anexada a ele. A sequência de processo no tempo causa o efeito de animação.

5 Conclusão

Este trabalho apresentou o processo de implementação do *walking gait* do time AUTUofM, descrevendo as soluções de software e justificando as decisões tomadas. Apresentou-se a abordagem tomada para mover a implementação da placa microcontroladora *OpenCM9.04* para um *software* dentro do controlador principal.

Para tanto, foram realizadas modificações no *firmware* da *OpenCM9.04* criando um componente *Proxy* que encaminha os comandos recebidos através da porta *USB* aos motores. Comandos estes gerados pela nova implementação do *walking gait*, desta vez utilizando a linguagem C++. O mesmo método usando um padrão de trajetórias senoidais e recuperação de distúrbios proposto por Karimi *et al* foi utilizado para a nova implementação.

Figura 21: Gráficos das trajetórias do ciclo de caminhada nos eixos X, Y e Z gerados pela nova implementação do *walking gait*.

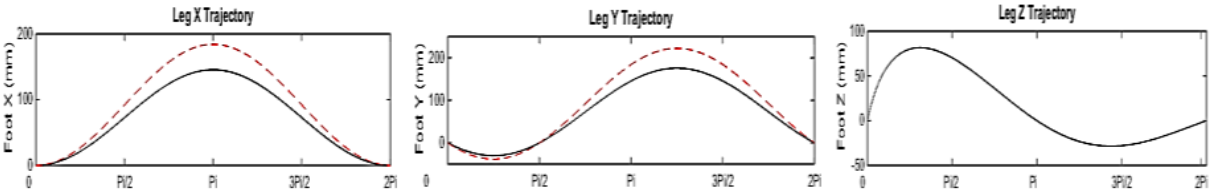

Figura 22: Gráficos das trajetórias do ciclo de caminhada nos eixos X, Y e Z gerados Karimi *et al*.

Devido a falta de dados concretos sobre a geração das trajetórias, foram utilizadas comparações visuais com os gráficos fornecidos por Karimi *et al* em seu trabalho. Nas figuras 21 e 22 é possível observar, respectivamente, os gráficos das trajetórias geradas pelo novo e pelo velho *walking gait*. Em ambas as imagens possível observar a trajetória nos eixos X, Y e Z separados. No eixo vertical do gráfico o trajetória é escalada em milímetros. No eixo horizontal a escala é no ciclo do relógio central.

Ainda na figura 22 observa-se uma linha vermelha pontilhada, que indica a compensação realizada pelo sistema durante um evento de distúrbio realizado no robô físico (KARIMI; SADEGHNEJAD; BALTES, 2016). Tal evento não foi possível de ser reproduzido pela falta de acesso ao robô físico.

Em relação ao movimento da caminhada, foi possível validar a geração da cinemática inversa através das animações, em tempo real, produzidas pelo visualizador 3D. A figura 23 observa-se 4 *capturas de tela* do visualizador em uma caminhada com $V_x = 0.5$ da perspectiva lateral, na primeira linha, e pela perspectiva frontal na segunda linha. Nas primeira coluna, $t = 0$, observa-se o mostra uma posição estável, com ambos os pés no chão. Na segunda, observa-se o momento em que o pé direito é levantado, onde $t = \pi/2$. A terceira coluna mostra o momento em que $t = \pi$, metade do ciclo, em que ambos os pés voltam a tocar no chão. Então, a quarta coluna mostra o momento em que o pé esquerdo sai do chão, $t = 3\pi/2$. Seguindo o ciclo, o relógio é reiniciado e o estado retorna ao da primeira coluna.

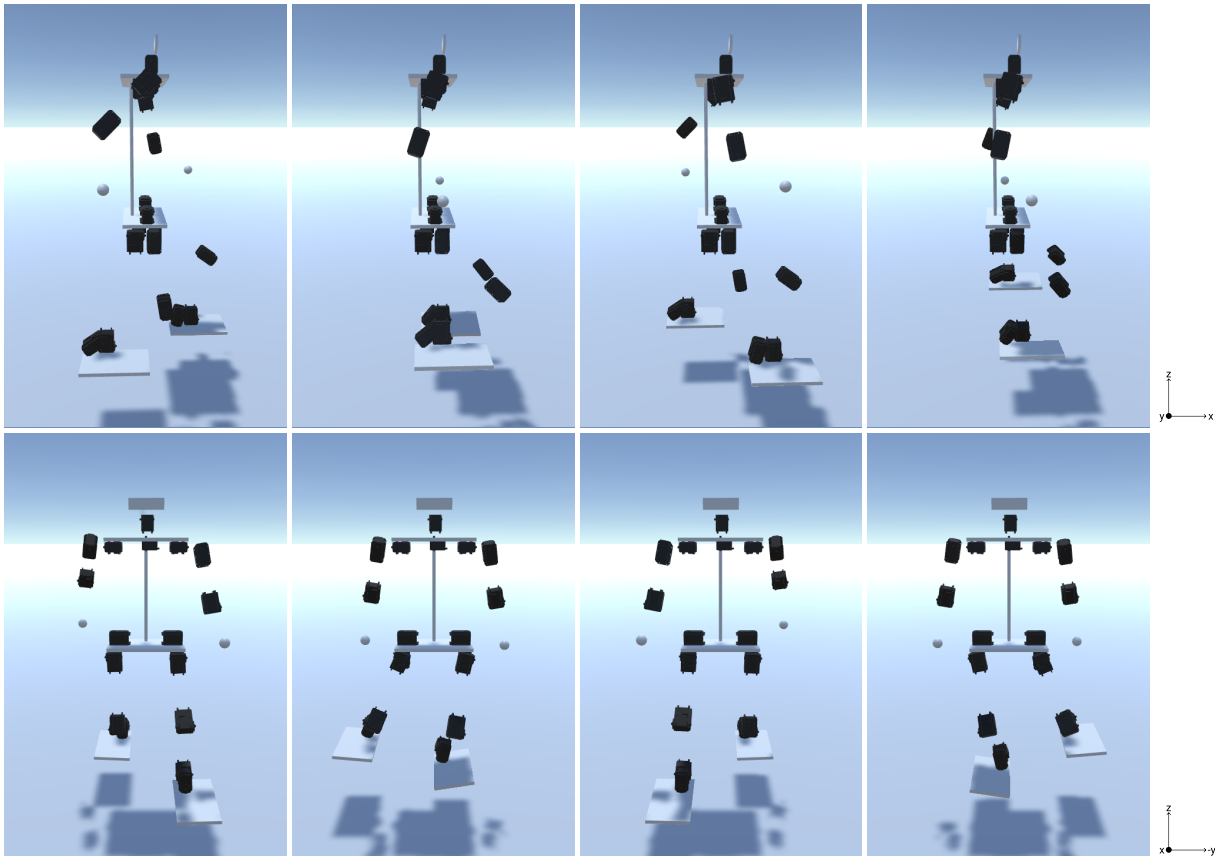

Figura 23: Visualização frontal e lateral dos movimentos da caminhada.

Um esquema de configurações persistidas em disco em arquivos JSON atacou diretamente o problema da falta de controle sobre as configurações geradas. Assim, os arquivos

de configuração podem ser versionados utilizando algum sistema de versionamento de código.

Apesar da simulação não ter sido implementada neste trabalho, seu futuro acomplamento será menos complexo, já que a nova implementação usa o conceito de um sistema distribuído baseado em serviços.

Considerando que o *walking gait* funciona dentro do controlador principal. Futuras melhorias que possam exigir mais recursos – processamento ou memórias RAM e ROM – podem ser implementadas.

Finalmente, de acordo com os resultados obtidos, podemos concluir que a implementação do *walking gait* foi realizada alcançando solucionando os problemas listados neste trabalho. Adicionalmente, o resultado pode ser utilizado como base para diversas melhorias descritas na seção 5.1.

5.1 Trabalhos futuros

As atualizações realizadas no *walking gait* original do time AUTUofM agora permitem diversas expensões no código para futuras atualizações e investigações. Assim, os próximos parágrafos descrevem alguns possíveis trabalhos futuros.

Este trabalho foi todo desenvolvido apenas usando *software* e visualização 3D dos ângulos gerados. A implementação e teste usando o robô real é um passo fundamental para a consolidação deste trabalho.

A implementação sem a utilização de um sistema de simulação apropriado não garante que o robô realmente possa caminhar. O uso do visualizador 3D mostra um esboço do movimento, sem a real caminhada, que pode gerar novos desafios. Em paralelo, embora os dados do leitor de orientação sejam levados em conta tem todos os cálculos, seus efeitos não foram testados. Consequentemente, a integração do módulo do *walking gait* a um *framework* de simulação é um passo em direção a um sistema mais completo.

Segundo, durante uma tarefa genérica realizada por um robô não apenas se caminha. Movimentos como chutes, levantar-se do chão, pegar um copo – entre outros – são normalmente pré-gravados na etapa de desenvolvimento e durante a tarefa são apenas reproduzidos. Desta forma, um módulo para gravar e reproduzir estes movimentos faz-se necessário.

Terceiro, o fato do *walking gait* rodar fora da *OpenCM9.04* adiciona algum atraso

extra entre o cálculo dos ângulos e a aplicação aos motores. Adicionalmente, os dados processados do leitor de orientação demoram mais a ter efeito nos cálculos das correções. Porém, o seu impacto real, durante a caminhada, não pode ser explorado neste trabalho deixando esta investigação para trabalhos futuros.

Quarto, alguns métodos de caminhada utilizam sensores nos pés de forma a detectar quando há o contato com o chão. Este momento é crítico pois, em caso de algum distúrbio, se houver uma diferença do momento esperado para o momento real do toque, ações diferenciadas poderão ser tomadas. Com isto em mente, sabe-se que os motores da série *MX* possuem *feedback* de carga baseado na voltagem de saída interna, onde deverá ser possível medir a diferença nas cargas esperadas e reais afim de prever se o pé está em contato com o chão.

Finalmente, outros métodos mais complexos de caminhada, utilizando mais processamento e mais memória, poderão ser implementados já que a barreira de recursos do *OpenCM9.04* foi quebrada.

Referências

CRAIG, J. J. *Introduction to Robotics: Mechanics and Control*. 2nd. ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1989. ISBN 0201095289.

GAZEBO. *Gazebo – Robot simulation made easy*. 2017. Acessado em: 25 de abr. de 2017. Disponível em: <http://gazebosim.org/>.

GREWAL, M. S.; ANDREWS, A. P. *Kalman Filtering: Theory and Practice with MATLAB*. 4th. ed. [S.l.]: Wiley-IEEE Press, 2014. ISBN 1118851218, 9781118851210.

KAJITA, S.; ESPIAU, B. Legged robots. In: *Springer Handbook of Robotics*. [S.l.: s.n.], 2008. p. 361–389.

KARIMI, M.; SADEGHNEJAD, S.; BALTES, J. *Online Omnidirectional Gait Modifications for a Full Body Push Recovery Control of a Biped Robot*. 2016.

MILLER, W. T. Real-time neural network control of a biped walking robot. *IEEE Control Systems*, v. 14, n. 1, p. 41–48, Feb 1994. ISSN 1066-033X.

ROBOTIS. *Robotis Support Page – OpenCM9.04*. 2016. Acessado em: 22 de abr. de 2017. Disponível em: <https://web.archive.org/web/20161210121538/http://support.robotis.com/en/product/controller/opencm9.04.htm>.

ROBOTIS. *Dynamixel Selection Guide*. 2017. Acessado em: 22 de abr. de 2017. Disponível em: <https://web.archive.org/web/20170310234926/http://en.robotis.com/index/product.php?cate_code=101310>.

ROS. *About ROS*. 2017. Disponível em: <http://www.ros.org/about-ros/>.

SADEGHNEJAD, S. et al. Aut-uofm humanoid teensize team. In: *Proceedings of RoboCup-2016 (Team Description Papers)*. Leipzig, Germany: [s.n.], 2016.

SPONG, M.; HUTCHINSON, S.; VIDYASAGAR, M. *Robot Modeling and Control*. [S.l.]: Wiley, 2005. ISBN 9780471649908.

TIANHUA, L. et al. The design and implementation of zero-copy for linux. In: *Proceedings of the 2008 Eighth International Conference on Intelligent Systems Design and Applications - Volume 01*. Washington, DC, USA: IEEE Computer Society, 2008. (ISDA '08), p. 121–126. ISBN 978-0-7695-3382-7. Disponível em: <http://dx.doi.org/10.1109/ISDA.2008.102>.