Universidade Estadual de Campinas

Faculdade de Engenharia Mecânica

Relatório Final

Trabalho de Graduação II

Análise de desempenho computacional em cálculo de cinemática reversa de mesa do tipo Stewart

Autor: Nilson Bernardo Pereira

Orientador: Prof. Dr. Andre Ricardo Fioravanti

Campinas, Novembro de 2015

Resumo

PEREIRA, Nilson Bernardo, *Análise de Desempenho Computacional em Cálculo de Cinemática Reversa de Mesa do tipo Stewart*, Faculdade de Engenharia Mecânica, Universidade Estadual de Campinas, Trabalho de Conclusão de Curso, (2015).

Em um ambiente onde é necessário uma plataforma que possua seis graus de liberdade, sendo eles: as translações nas três dimensões (*x,y,z*) e as rotações (*rolo, passo e guinada*), é comum o uso da plataforma do tipo Stewart. Existem diferentes formas de construção dessa plataforma e nesse trabalho foca-se em um sistema baseado em cilindros excêntricos. Apesar das baixas amplitudes de movimentos, esse modelo pode ser encontrado em alguns laboratórios de pesquisa como é o caso do Laboratório Nacional de Luz Síncrotron localizado em Campinas. Propõe-se analisar o modelo matemático do sistema assim como sua cinemática direta e reversa, além de investigar soluções em algoritmos paralelos que visam minimizar o tempo computacional exigido nos cálculos de cinemática reversa a partir ou não de uma trajetória. O desempenho desses algoritmos são comparados entre diferentes sistemas computacionais embarcados como a Raspberry PI e Parallella da Adapteva.

Palavras Chave: Plataforma Stewart, hexapode, parallela, cinemática reversa

Lista de Figuras

Lista de Tabelas

Conteúdo

Capítulo 1

Introdução

1.1 Motivação

Uma plataforma do tipo Stewart consiste basicamente de duas bases paralelas onde uma é fixa e outra é idealmente livre para transladar e rotacionar em todas as direções. As limitações de movimentos diferem de acordo com o tipo de construção. Esse sistema é comumente usado em soluções robóticas onde rigidez estrutural, e baixas amplitudes de movimento são necessárias.

Um grande número dessas plataformas podem ser encontradas em laboratórios de luz síncrotron ao redor do mundo. No Stanford Linear Accelerator Center (SLAC) a plataforma é usada para alinhamento de feixe de elétrons, assim como no Swiss Light Source(SLS) e no Australian Synchrotron. No Laboratório Nacional de Luz Síncrotron (LNLS) localizado na cidade de Campinas / SP, há uma dessas plataformas sendo usado no alinhamento de espelhos[5] em uma de suas linhas de luz.

Para controlar tal dispositivo, é necessário um sistema computacional de controle dedicado. Como nesse ambiente, a precisão de movimento é mais importante que as velocidade máximas de transição, o sistema de controle deve ser capaz de trabalhar com uma alta taxa de amostragem a fim de reduzir os erros de posicionamento durante a trajetória. É comum encontrar soluções de controle para a plataforma baseadas em FPGA ou sistemas completos do tipo "desktop"como parte do pacote que acompanha tal dispositivo. Tais soluções exigem espaço adicional e um custo proporcional à sua capacidade computacional. É de interesse da indústria buscar soluções compatíveis em termos de processamento mas com custo e tamanho reduzidos.

Dada as limitações de velocidade dos processadores atuais, há uma tendência de criar computadores contendo vários processadores que trabalham simultaneamente em paralelo. Infelizmente, esse aumento no número de núcleos, não traz o mesmo valor de ganho na velocidade final

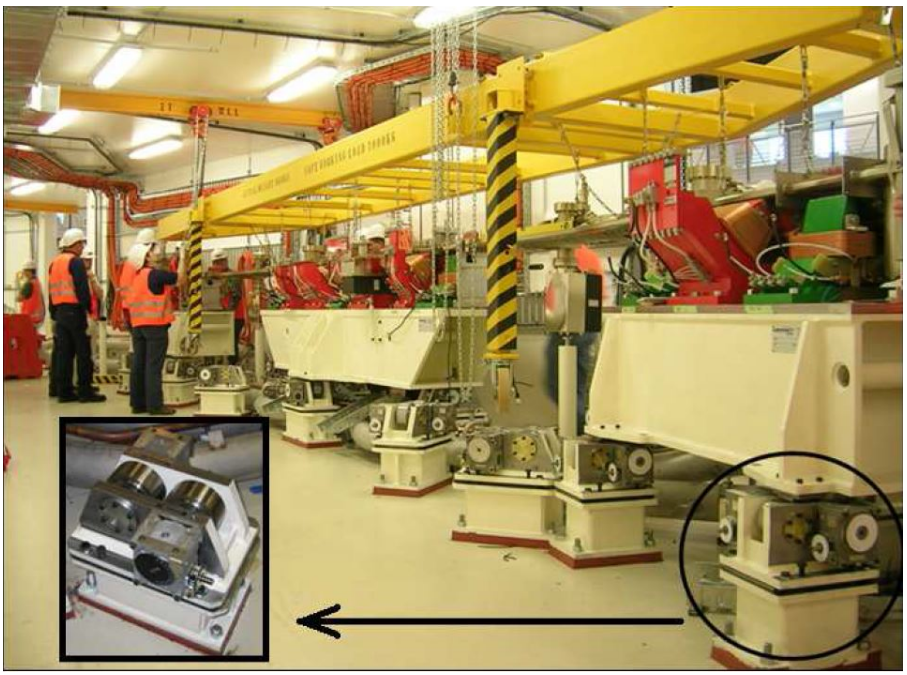


Figura 1.1: Berços de imãs do acelerador de elétrons no Australian Synchrotron. Em detalhe o sistema de alinhamento do berço [5].

de execução de um único processo a não ser que esse processo ou algoritmo seja capaz de tirar proveito dos múltiplos processadores disponíveis no sistema no qual ele está sendo executado. A paralelização de algoritmos seriais tem sido tema de muitas pesquisas atualmente e que se tem mostrado promissora na busca de melhores desempenhos.

Há uma iniciativa também por parte dos desenvolvedores de hardware de criar sistemas embarcados de alto desempenho contendo dezenas e até centenas de núcleos como é o caso da placa Parallella, desenvolvida pela Adapteva, e a Jetson, desenvolvida pela NVIDIA. O desempenho é proveniente portanto da combinação de um hardware contendo vários núcleos e um bom algoritmo que tenha sido paralelizado, ou seja, que tire proveito de tal arquitetura.

1.2 Nosso Trabalho

Usaremos nesse trabalho um modelo de plataforma Stewart encontrada no Laboratório Síncrotron brasileiro como referência para realizar a modelagem matemática desse sistema. O movimento da plataforma se baseia no deslocamento de três pontos de apoio. Na interface desses pontos encontra-se uma semi esfera que se apoia sobre dois cilindros. Os cilindros rotacionam ao redor de um ponto distante do seu centro geométrico e por isso serão chamados de excêntricos. Tal configuração permite que a esfera de apoio seja posicionado em 2 dois eixos (x, z) e fique livre no terceiro (y). A amplitude de movimento final desse sistema é pequeno mas suficiente para

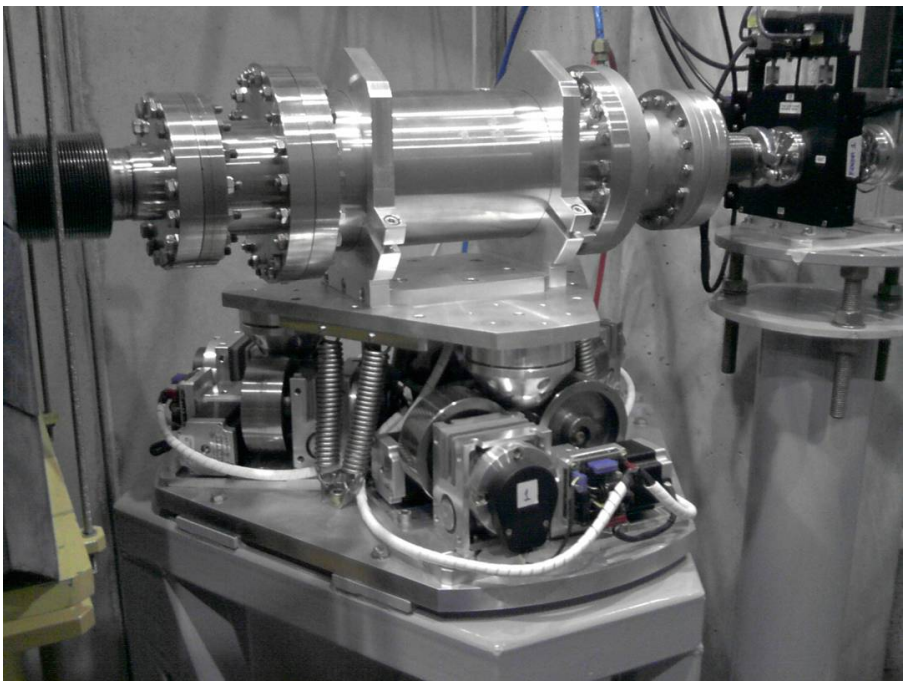


Figura 1.2: Sistema de alinhamento da câmara de espelhos no LNLS baseado em plataforma do tipo Stewart [7].

a necessidade dos laboratórios que empregam essa tecnologia.

Por simplicidade, abordaremos o modelamento da plataforma a partir de dois modelos distintos. No primeiro, faremos a cinemática direta que relaciona os ângulos de rotação dos excêntricos com a posição do centro da esfera de apoio. A partir disso, obteremos a cinemática reversa desse problema que relaciona o deslocamento desejado do centro da esfera com os deslocamentos necessários dos ângulos de rotação dos excêntricos. No segundo, faremos apenas a cinemática direta que relaciona a orientação e translação da plataforma com a localização dos centros das esferas de apoio.

Para realizar os testes de desempenho, faremos uso de uma técnica chamada de HIL ou "Hardware in the loop". Nesse ambiente, um PC com um software de simulação (no nosso caso o Matlab da Mathworks) é responsável pela interface de visualização e comando da simulação da plataforma, assim como as respostas dinâmicas dos atuadores simulados. Os sinais de comando são enviados via rede UDP/IP para o hardware em teste que fica responsável pelos cálculos de erro de posicionamento e cálculos dos sinais de controle que devem ser enviados aos atuadores virtuais contidos na simulação também via rede UDP/IP.

O objetivo de usar o HIL é simular um ambiente real onde os comandos de posicionamento da plataforma é proveniente de um usuário diante de um sistema de interface e serão enviados ao sistema de controle. O hardware de controle atuaria diretamente nos motores de posicionamento

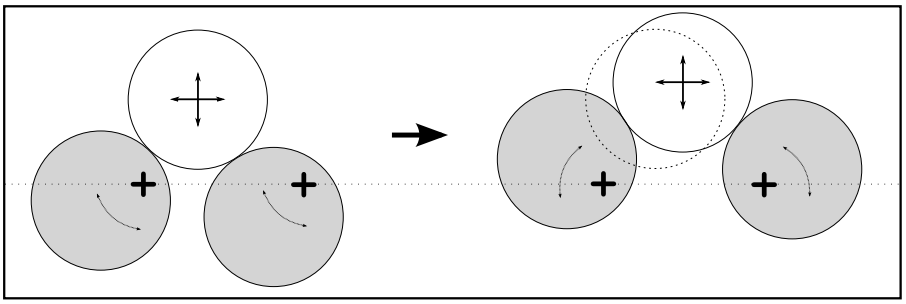


Figura 1.3: Representação do sistema de posicionamento da esfera de apoio baseado em excêntricos [5].

dos excêntricos. No entanto, o Matlab receberá esses dados de atuação e responderá de acordo com um modelo de dinâmico dos supostos atuadores. A avaliação de desempenho do hardware de controle provém da medição do tempo entre o envio dos comandos de posicionamento e o recebimento do comandos de atuação sobre os motores.

Usaremos como hardware de teste diferentes plataformas de sistemas embarcados. Cada uma dessas plataformas possuem características de funcionamento e paradigmas de programção distintas entre si e cujos impactos no desempenho é o foco do nosso trabalho. Pretende-se testar 4 diferentes plataformas: Arduino Mega 2560 (com ethernet Shield), Raspberry PI (Modelo B), Parallela da Adapteva e Jetson da Nvidia.

Para comparar o desempenho das diferentes placas, usaremos como métrica a taxa de amostragem média obtida para cada sistema. O algoritmo não terá restrições de tempo no ciclo de cálculo e portanto a taxa de amostragem média obtida será a média máxima possível. Os tempos envolvidos na comunicação UDP entre o computador que está executando o Matlab e a placa de teste serão avaliados e, se significantes, serão incorporados na comparação final. Além das placas mencionadas, usaremos também um computador do tipo desktop com boa capacidade computacional como referência de desempenho.

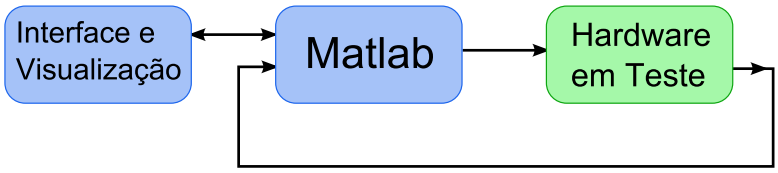


Figura 1.4: Diagrama de comunicação no HIL (Hardware in the Loop). Interface, visualização e planta do sistema acontecem em um PC (azul) enquanto cálculos de controle são executados no hardware em teste (verde).

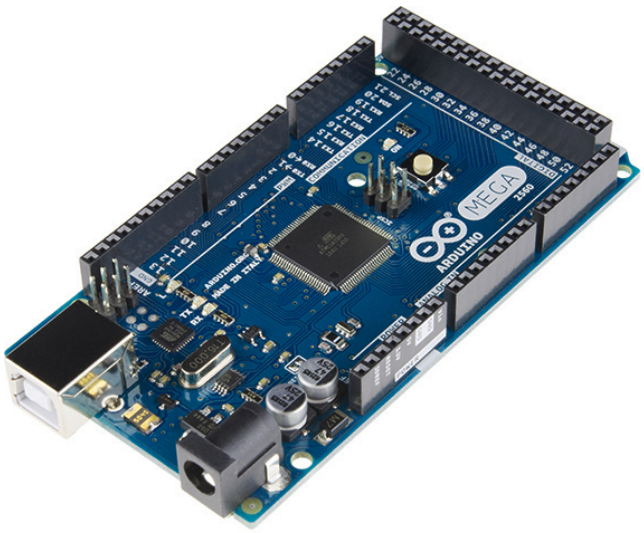


Figura 1.5: Arduino Mega 2560 [3].

Arduino Essa placa se tornou muito popular entre os hobistas de automação no últimos anos. Os desenvolvedores tiraram proveito dos microcontroladores da AVR, que já possuíam um mercado consolidado na área, porém exigia um certo conhecimento por partes dos usuários tanto na área de eletrônica, quanto na área de programação.

O Arduino nada mais é do que uma plataforma constituída de uma interface de fácil programação aliada uma placa de código aberto que faz a interface eletrônica com um dos microcontroladores da AVR. Ao criar uma solução comercial relativamente barata e mais amigável ao consumidor final, a popularidade aumentou e devido ao seu código aberto, criou-se um universo de acessórios para diferentes aplicações podendo ser explorado por outras companhias.

O arduíno, por ser um hardware dedicado, não possui um sistema operacional e executa apenas um programa por vez. Isso pode trazer certas vantagens no determinismo nos cálculos de controle, porém devido ao seu baixo clock de funcionamento, tende a ser muito lento quando exigido computacionalmente.

Raspberry PI Essa placa também, se tornou muito popular nos últimos anos. Ela foi criada para suprir a necessidade de um sistema computacional completo similar a um computador pessoal, porém com dimensões e preço reduzidos. Ela necessita de um sistema operacional e possui suporte para diferentes distribuições de Linux. O mais indicado para essa placa é uma versão reduzida do Linux Debian chamado de Raspbian. Com 700MHz de clock, ela possui um poder computacional significante dado o baixo consumo energético, seu preço atraente e a facilidade de incorporá-la em projetos onde o espaço físico é limitado.

Ela é capaz de executar qualquer tipo de programa que seja compatível com o Linux OS, o

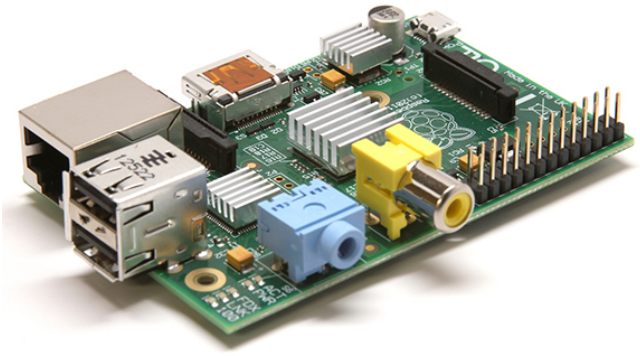


Figura 1.6: Raspberry PI modelo B [12].

que traz grande flexibilidade aos desenvolvedores dada ao imenso acervo de software gratuito disponível.

Existe ainda a opção de substituir o sistema operacional padrão da Raspberry PI por um sistema operacional de tempo real como FreeRTOS ou RTEMS. Tais sistemas proporcionam tempos de respostas menores a eventos externos e com menores variações dos tempos de processamento. Se existir a necessidade de melhores taxas de amostragem por parte desse hardware, uma versão contendo um sistema operacional de tempo real poderá ser implementada para efeito comparativo.

Parallella A Parallela foi desenvolvida com o objetivo de ser uma plataforma de desenvolvimento de baixo custo a desenvolvedores e pesquisadores de sistemas de alto desempenho baseado em paralelismo, ou seja, programas que fazem uso de uma arquitetura de hardware contendo mais de 1 núcleo de processamento. Ela dispõe de um processador ARM de 2 núcleos dedicados ao sistema operacional e programas além de um processador adicional periférico contendo outros 16 núcleos de processamento chamado Epiphany. Similar à Raspberry PI, ela também necessita de um sistema operacional baseado em Linux para funcionar. A comunidade de desenvolvedores está crescendo e hoje ela possui suportes para diferentes paradigmas de computação paralela como PThreads, MPI e OpenMP. A empresa desenvolvedora, Adapteva, busca popularizar a computação paralela com o objetivo de acelerar o desenvolvimento dessa área.

Jetson TK1 Com a necessidade de cada vez mais desempenho em sistemas computacionais, algumas empresas são praticamente dedicadas a desenvolver arquiteturas de hardware capazes de incorporar o maior número de núcleos em um único processador que é o caso da NVIDIA. Essa empresa tem como base placas de aceleradores gráficos de vídeo voltado à indústria de

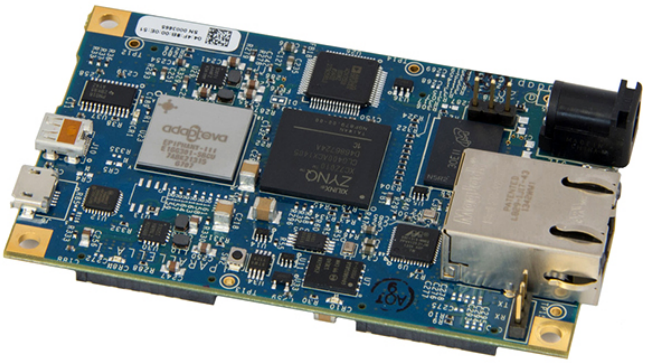


Figura 1.7: Parallella-16 Desktop Computer [11].

vídeo games, porém, pesquisadores encontraram nessa placas, um grande potencial para executar cálculos de diferente tipos ao invés de apenas cálculos voltados ao mundo gráfico. A NVIDIA, percebendo a oportunidade, investiu em hardwares que possui centenas e até milhares de núcleos e também em uma biblioteca chamada CUDA que facilita o uso dessas placas em computação de alto desempenho. Devido à sua arquitetura voltada à computação gráfica, a programação em CUDA não é trivial e exige um certo conhecimento desse paradigma para tirar o melhor proveito do hardware. Porém, se o problema abordado for compatível com esse paradigma, o grande número de núcleos propiciam ganhos computacionais únicos na indústria.

A placa Jetson TK1 é também uma placa de desenvolvimento voltado à pesquisadores de sistemas embarcados que buscam plataformas portáteis de alto desempenho e relativo baixo consumos de energia. A indústria automobilística é um setor que vem investindo bastante em desenvolvimentos de veículos autônomos e encontraram na Jetson TK1 uma platarforma robusta capaz de lidar com o grande números de dados que é necessário ser tratado durante a fusão sensorial para navegação do veículo. Por consequência, a indústria de automação robótica também pode se beneficiar desse hardware e é o que será explorado nesse trabalho.

Figura 1.8: NVIDIA Jetson TK1 [6].

Tabela 1.1: Especificações das diferentes placas de testes.

Board	**Arduino**	**Rasp. PI**	**Parallela**	**Jetson**
Reg Size	8bit	32bit	32bit	32bit
Frequency (Mhz)	16	700	1000	2320
Cores	1	1	18	196
RAM	256kB	512MB	1GB	2GB
LAN	100Mb	100Mb	1Gb	1Gb
Dimensões(mm)	108x53x33	93x64x20	90x55x15	127x127x26
Preço (USD)	$40	$35	$100	$200

Capítulo 2

Modelagem

Usaremos como referência para a modelagem matemática uma versão da plataforma Stewart baseada em excêntricos similar à encontrada no LNLS. Uma versão simplificada foi criada em CAD para melhorar a visualização dos componentes básicos de posicionamento (fig. 2.1). Nota-se que a plataforma possui semi-esferas como apoio sobre os cilindros.

Foi adotado dois sistemas de coordenadas (O_0 e O_1), ambos coincidentes e posicionados no centro da plataforma. O sistema O_0 é fixo enquanto o sistema O_1 acompanha a movimentação da plataforma. No centro entre os excêntricos há um sistema de coordenadas denominados A_0, B_0 e C_0 (fig. 2.2a), usados para criar uma relação entre os ângulos de rotação dos pares de excêntricos e a posição do ponto E_i situado no centro da semi esfera de apoio (fig. 2.2b).

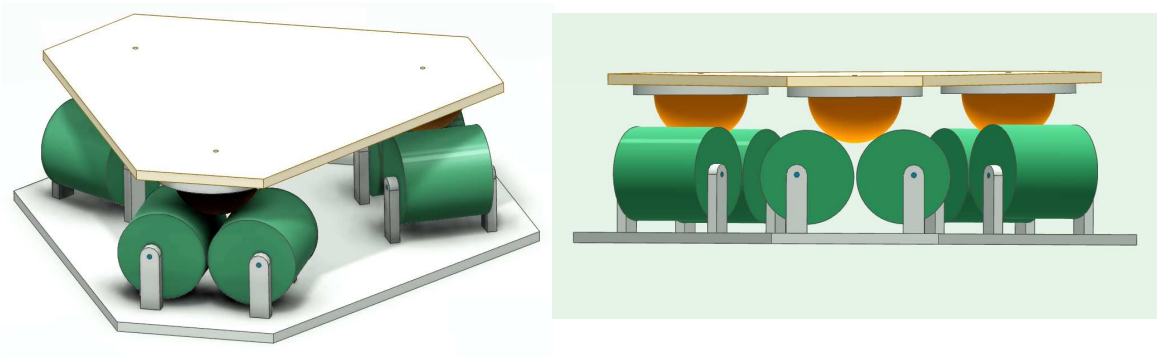
2.1 Cinemática direta

O primeiro passo é encontrar uma relação entre os ângulos α e β e a posição do ponto E para cada par de excêntricos. Para qualquer ângulo α e β de rotação dos cilindros, nota-se que a magnitude do vetor L que liga o centro geométrico de um dos cilindros ao ponto E é constante. Basta portanto encontrar o ângulo de inclinação λ do vetor L com o eixo x para encontrar a posição do ponto **E** a partir do ponto **A**.

Os vetores **A** e **B** são definidos por

$$\mathbf{A} = r[\cos(\alpha)\ 0\ \sin(\alpha)]^T + a_0$$

$$\mathbf{B} = r[\cos(\beta)\ 0\ \sin(\beta)]^T + b_0$$



(a) (b)

Figura 2.1: Visão superior (a) e lateral (b) do modelo em CAD da plataforma usado como referência para modelamento.

Para $F = |\mathbf{A} - \mathbf{B}|$ temos

$$\theta = \arcsin\left(\frac{B_z - A_z}{F}\right)$$
$$\epsilon = \arcsin\left(\frac{F}{2L}\right)$$
$$\lambda = \frac{\pi}{2} - \epsilon + \theta$$

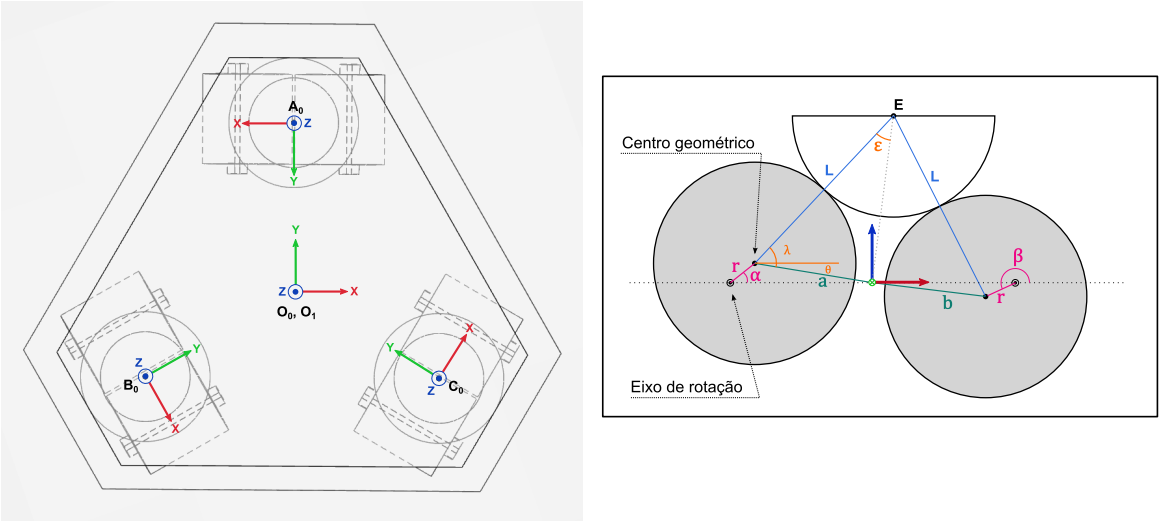
O ponto $\mathbf{E}$ é portanto

$$\mathbf{E} = L[\cos(\lambda)\ 0\ \sin(\lambda)]^T + \mathbf{A} \tag{2.1}$$

2.2 Cinemática Reversa

A cinemática reversa consiste em encontrar uma relação inversa à relação direta, ou seja, uma função que tenha como resultado ângulos α e β a partir de uma desejada posição do ponto $\mathbf{E}$. Porém nem sempre é possível de estabelecer uma relação direta ou uma única configuração dos atuadores pra uma determinada entrada, ao invés obtém-se um conjunto de respostas. Para evitar essa situação, faremos uso de uma ferramenta matemática chamada matriz Jacobiana. Através dela é possível estabelecer uma relação entre pequenos deslocamentos do ponto $\mathbf{E}$ com pequenos deslocamentos dos ângulos dos atuadores α e β para uma configuração conhecida [13]. Esse método soluciona o problema da cinemática reversa porém possui um certo custo computacional pois a matriz deve ser recalculada constantemente pra cada nova configuração dos ângulos dos atuadores.

Para calcular a matrix Jacobiana é necessário primeiramente ter uma equação que define o ponto $\mathbf{E}$ em função dos ângulos dos atuadores conforme equação 2.1 dadas suas devidas substituições. Separando a equação em seus componentes cartesianos temos:



(a) (b)

Figura 2.2: (a) Visão superior da plataforma com os sistemas de coordenadas adotado. (b) Visão frontal de um dos sistema de excêntricos com sistema de coordenadas e vetores usados na cinemática direta.

$$\mathbf{E_x} = f_1(\alpha, \beta)$$

$$\mathbf{E_y} = 0$$

$$\mathbf{E_z} = f_2(\alpha, \beta)$$

Observando a configuração dos excêntricos, percebe-se que não há atuação no eixo y. A semi esfera de apoio está livre nesse eixo e portando não é possível definir sua posição exata sem conhecer a posição dos outros excêntricos. Isso não será um problema no posicionamento da plataforma e facilitará alguns dos cálculos envolvidos na solução proposta.

A Jacobiana é composta das derivadas parciais de primeira ordem de uma função vetorial e no nosso caso ela irá relacionar as variações de deslocamento do ponto de apoio com os deslocamentos dos ângulos dos excêntricos[10]. Portanto temos a Jacobiando definida como:

$$J = \begin{bmatrix} \frac{\partial f_1}{\partial \alpha} & \frac{\partial f_1}{\partial \beta} \\ \frac{\partial f_2}{\partial \alpha} & \frac{\partial f_2}{\partial \beta} \end{bmatrix}$$

As componentes calculadas da matriz Jacobiana podem ser encontradas no apêndice A desse trabalho. A relação de deslocamento para um ponto de apoio é portanto:

$$\begin{bmatrix} \dot{E}_x \\ \dot{E}_z \end{bmatrix} = J \begin{bmatrix} \dot{\alpha} \\ \dot{\beta} \end{bmatrix}$$

Para conseguir a relação inversa desejada, basta fazermos:

$$\begin{bmatrix} \dot{\alpha} \\ \dot{\beta} \end{bmatrix} = J^{-1} \begin{bmatrix} \dot{E}_x \\ \dot{E}_z \end{bmatrix} \tag{2.2}$$

Com isso obtemos, para uma posição instantânea, as variações angulares necessárias dos excêntricos para uma variação desejada do ponto de apoio **E**.

2.3 Singularidades

Observando a equação 2.2, notamos que é necessário obter a matriz inversa da matriz Jacobiana. Como se trata de uma matriz quadrada, a única condição que pode tornar essa matriz não inversível é quando o seu determinante é igual a zero. Essa situação é chamada de singularidade. Fisicamente, a singularidade ocorre quando há restrição de movimento do ponto de apoio devido à configuração momentânea dos atuadores. No nosso caso, temos duas singularidas e ambas ocorrem quando o vetor r que interliga o centro de rotação do excêntrico com seu centro geométrico é paralelo ao vetor L que interliga o centro de apoio **E** ao centro geométrico do excêntrico.

Para evitar esse problema, faremos uso de uma técnica chamada de DLS (Damped Least Square)[10]. Essa técnica, também chamada de Método Levenberg-Marquardt [4], tem sido aplicada mais frequentemente em robótica nos últimos anos como uma maneira de evitar problemas com pseudo inversa no caso de matrizes Jacobianas não quadradas e se caracteriza como um método que produz resultados numericamente estáveis.

A relação entre a inversa da Jacobiana e a DSL é definida por:

$$J^{-1} \approx J^T (JJ^T + \lambda^2 I)^{-1} \tag{2.3}$$

Uma comparação numérica entre a DLS e a Jacobiana em regiões distantes e próximas à singularidade para diferentes valores de lambda pode ser encontrada no apêndice B desse trabalho.

2.4 Pontos de apoio

Uma particularidade observada na plataforma Stewart implementada no LNLS é a ausência de sensores capazes de identificar a orientação e a translação da plataforma. O sensoriamento é

feito apenas nos excêntricos através de encoders rotativos, o que garante apenas o posicionamento angular dos mesmos. Como a estrutura é rígida e as cargas envolvidas são pequenas, ficou-se comprovado[5] que os erros de posicionamento da plataforma são insignificantes para a aplicação em que é usada quando os pontos de apoio (Ponto **E** para cada par de excêntricos) se encontram na posição correta.

Dessa forma usaremos como valores de referência as coordenadas dos três pontos de apoio ($\mathbf{E_A}$, $\mathbf{E_B}$ e $\mathbf{E_C}$) que serão enviados para o hardware de teste para cálculo dos ângulos dos excêntricos através da cinemática reversa e cálculo dos valores de atuação sobre os motores.

Comforme observado na sessão 2.2 não há atuação sobre o eixo y nos pontos de apoio. Porém, as semi-esferas ficam livres para transladar nesse eixo sem restrições. Devido a configuração angulada entre os pares de atuadores (fig.2.2) a posição em y é consequência das coordenadas nos outros eixos x e z e portanto é redundante e pode ser descartada.

Inicialmente precisamos encontrar as coordenadas dos pontos de apoio a partir de uma posição e orientação da plataforma. Qualquer ponto na base superior da plataforma se encontra no sistema de coordenadas $\mathbf{O}_1$. Esses pontos se relacionam com o sistema referencial fixo da base inferior $\mathbf{O}_0$ através de uma matriz de rotação $\mathcal{R}_1$ e uma translação entres as origens $\mathbf{O}_1^0$.

Iremos empregar uma matriz de rotação do tipo roll-pitch-yaw, definida como o produto de rotações sucessivas em uma determinada ordem:

$$\begin{aligned}\mathcal{R}_a^b &= \mathcal{R}_{z,\phi}\mathcal{R}_{y,\theta}\mathcal{R}_{x,\psi} \\ &= \begin{bmatrix} c_\phi & -s_\phi & 0 \\ s_\phi & c_\phi & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} c_\theta & 0 & s_\theta \\ 0 & 1 & 0 \\ -s_\theta & 0 & c_\theta \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & c_\psi & -s_\psi \\ 0 & s_\psi & c_\psi \end{bmatrix}\end{aligned}$$

No exemplo acima a matriz de rotação traduz os pontos do referencial a no referencial b. Quando não há notação, assume-se referencial 0.

Os pontos de apoio em $\mathbf{O}_0$ são portanto:

$$\mathbf{E}_i = \mathcal{R}_1\mathbf{E}_i^1 + \mathbf{O}_1$$
$$i = A, B, C$$

Para encontrar a posição dos pontos de apoio em seus respectivos sistemas de coordenadas usaremos:

$$\mathbf{E_i^i} = \mathcal{R}^i\mathbf{E}_i + \mathbf{O}^i$$

As coordenadas dos pontos de apoio serão os valores de referência que desejamos controlar para garantir o posicionamento final da plataforma.

Capítulo 3

Sistema Computacional

3.1 Geração Automática de Código

Para testar o modelo matemático da plataforma Stewart, foi usado o software Matlab da Mathworks. Esse programa oferece também a opção de traduzir os scripts criados no Matlab em código fonte em outras linguagens de programação. Com o objetivo de testar esse recurso, foi gerado código fonte na linguagem C para ser compilado e executado nos respectivos equipamentos de teste desse trabalho.

Essa ferramenta oferece ainda a possibilidade de gerar código levando em conta a arquitetura de hardware do equipamento de destino. Essa opção influencia nos tipos de variáveis criadas entre ponto flutuante ou fixo e ainda o tamanho em bits (32 ou 64 bits) da variável. Outras opções ainda incluem suporte matemático a outros tipos de representação numérica como $\pm\infty$ (infinito) e NaN (Not a Number).

O uso dessa ferramenta diminui substancialmente o tempo necessário para criação de códigos que fazem uso de ferramentas matemáticas complexas dada a imensa biblioteca de funções disponíveis na plataforma Matlab. Além do acesso a essas funções, os algoritmos empregados pelo Matlab sempre refletem um bom compromisso entre acurácia do resultado e desempenho computacional. Outra vantagem dessa abordagem está no uso dos recursos de simulação do ambiente Matlab que facilitam e agilizam a criação dos scripts primários que darão origem ao código final.

O código gerado pelo matlab compilou sem problemas em todas as plataformas com exceção da compilação para o processador Epiphany da Parallella. O tamanho total da memória disponível em cada core é de 32kB e representa todo o espaço disponível para variáveis e código de execução. O excesso de funções de suporte geradas pelo Matlab fazia com que o código com-

pilado excedesse o espaço disponível em cada core. Foi necessário desabilitar algumas funções como o suporte aos tipos numéricos ∞ e NaN, assim como funções que definiam alguns tipos de variáveis para que o código final compilado não excedesse o espaço disponível. A remoção dessas funções não alterou o funcionamento do código final. Os scripts criados no Matlab que funcionaram como fonte para a geração de código podem ser encontrados no apêndice C. Os quatro scripts são:

- sample.m - Usado pelo sistema de geração automática de código para identificar os tipos de variáveis usados como dados de entrada e saída para cada função.
- direct.m - Recebe como entrada os ângulos atuais dos excêntricos e a posição desejada dos pontos de apoio. Retorna o erro de posição dos pontos de apoio.
- jacobian.m - Recebe os ângulos atuais dos excêntricos e retorna as matrizes Jacobianas para cada ponto de apoio.
- dls.m - Recebe as matrizes Jacobianas e um valor de λ como entrada. Retorna as matrizes DLS correspondentes.
- mx multiply.m - Recebe as matrizes DLS e os erros de posicionamento dos pontos de apoio. Retorna as variações angulares dos excêntricos.

3.2 Algoritmo para Análise de Desempenho

Com o objetivo de padronizar a avaliação do desempenho das plataformas testadas, foi criado uma rotina em C responsável pela medição dos tempos gastos durante os testes. Essa rotina foi executada sempre no mesmo sistema operacional (Ubuntu Linux 12.04 LTS) e no mesmo hardware. Vale ressaltar que a placa de rede desse PC, assim como o switch usado para conexão de rede possuem suporte a comunicações de até 1 Gbps por porta.

Os testes consistem em enviar uma trajetória pré-calculada dividida em 10000 passos para medir o tempo médio necessário para cálculo de cada ponto. Esse tempo médio funciona como indicador das frequências máximas possíveis para aquele hardware nas condições propostas.

Os dados são enviados para o hardware de teste através das placas de rede usando o protocolo UDP. Esse protocolo não exige confirmação de recebimento dos dados como faz o protocolo TCP e por isso proporciona taxas de transferências mais rápidas devido ao menor overhead de comunicação.

Padronizou-se enviar (em binário) um vetor de dados (*txvec*) contendo 13 valores de *doubles* (8 bytes para cada valor), ou seja, 104 bytes de dados organizados da seguinte maneira:

$$txvec = \begin{bmatrix} tag & \alpha_A & \beta_A & \alpha_B & \beta_B & \alpha_C & \beta_C & E_{xA} & E_{zA} & E_{xB} & E_{zB} & E_{xC} & E_{zC} \end{bmatrix}$$

Onde *tag* é um número que identifica o pacote, α_i e β_i representam as posições angulares atuais dos excêntricos e E_{xi} e E_{zi} representam as posições desejadas para cada ponto de apoio determinados pela orientação e posição da plataforma.

O vetor de resposta (*rxvec*) é composto pela *tag* de identificação além de mais 6 valores de *doubles* que representam as variações angulares dos excêntricos. O pacote de dados possui portanto 56 bytes de dados.

$$rxvec = \begin{bmatrix} tag & \Delta\alpha_A & \Delta\beta_A & \Delta\alpha_B & \Delta\beta_B & \Delta\alpha_C & \Delta\beta_C \end{bmatrix}$$

Ao finalizar o teste, o algoritmo imprime na tela a quantidade de pacotes que foram recebidos com sucesso, o valor médio de tempo encontrado para cada iteração e o desvio padrão dessa média. Para poder separar o tempo envolvido em comunicação do tempo de cálculo, para cada hardware foram realizados 2 testes: o primeiro era realizado por completo envolvendo todos os cálculos necessários enquanto o segundo consistia apenas em medir o tempo necessário para enviar o vetor de dados e receber um vetor de resposta contendo apenas zeros. A diferença entre os dois programas estava apenas nas chamadas das funções de cálculo, que eram executadas ou não dependendo do teste. O mesmo procedimento foi realizado nos algoritmos paralelos para que o tempo de comunicação incluísse o overhead de paralelização inerente à arquitetura.

3.3 Algoritmo serial

Como a maioria das plataformas testadas possuem linux como sistema operacional, optou-se por criar um algoritmo de base que pudesse ser compilado e executado sem alterações. O código completo pode ser encontrado no apêndice F.

O algoritmo basicamente abre uma porta de comunicação usando o protocolo UDP/IP e padronizou-se usar a porta 50101 no PC cliente e a porta 50102 no hardware de teste. Uma vez executado, o programa fica suspenso a espera por um dado de entrada. Uma vez recebido o dado, executa-se a rotina de cálculos ou não, dependendo do tipo de teste, e envia o vetor de resultados de volta. O único dispositivo que precisou de um algoritmo ligeiramente diferente foi

o Arduino pois ele possui suas próprias bibliotecas de desenvolvimento[2]. O código completo usado no Arduino pode ser encontrado no apêndice G.

3.4 Algorítmo paralelo

Nem todo algoritmo pode ser paralelizado em função da dependência temporal entre as variáveis durante processamento[9]. No nosso caso em particular a paralelização do algoritmo é simples pois os cálculos de um ponto de apoio não depende dos cálculos dos pontos vizinhos. Basta portanto dedicar um core para cada ponto de apoio para dividir a tarefa em 3 cores para processamento. A dificuldade aparece em sincronizar a entrada e saída de dados entre os cores. Essa sincronia e divisão do trabalho podem ser feitas de diferentes formas e dependem do paradigma de programação e arquitetura do hardware. Como o processador Epiphany da Parallella difere bastante da arquitetura CUDA da Nvidia, foram feitas duas abordagens distintas para paralelizar nosso algoritmo de cálculo.

3.4.1 Parallela

Para executar programas em paralelo na placa Parallella é necessário criar duas rotinas distintas. Uma é executada no processador principal da placa (host) que é responsável por carregar um segundo código, que pode ser chamado de kernel, já compilado para a memória da Epiphany para execução. Uma vez carregado e iniciado, o mesmo código é executado em todos os cores que fazem parte do grupo de cores designados pelo programa principal. Cabe ao programador dividir e sincronizar as tarefas para evitar conflitos de acesso a memória.

Algo importante sobre essa arquitetura é o espaço de memória designado para cada core que é de apenas 32kB. Esse espaço é compartilhado entre o código a ser executado, constantes e variáveis. É possível verificar o tamanho do código após compilação usando a ferramenta `"e-objdump"` que faz parte do seu pacote de desenvolvimento (SDK)[1]. Para paralelizar foram usados 4 cores apenas. Cada core possui seu próprio espaço de memória e possuem acesso às memória de outros cores através das funções `"e_read"` e `"e_write"`. Para facilitar essa troca de informações, foram criadas posições de memória estáticas que eram comuns entre todos os cores usados. O algoritmo se dá através dos seguinte passos:

1. Ao receber novo dado, o programa host escreve os dados recebidos em um vetor no core 0.

2. Core 0, ao notar o novo dado através da tag, sinaliza os outros 3 cores sobre a existência de novo dado.

3. Ao receber o sinal, cores 1,2 e 3 copiam os dados de interesse do core 0 e efetuam o cálculo.

4. Cada core atualiza um sinalizador no core 0 contendo o tag do dado calculado.

5. O core 0 espera pela atualização dos tags no seu vetor de sinalizadores e atualiza um sinalizador em sua memória que o host está monitorando.

6. O programa host ao perceber a atualização do sinalizador do core 0, recolhe os dados e envia de volta ao cliente.

O uso de um outro core para controle dos sinalizadores foi necessário para evitar conflitos de leitura de uma única posição de memória por várias cores. O código completo se encontra no apêndice H.

3.4.2 Cuda

A arquitetura Cuda se diferencia das outras plataformas de múltiplos núcleos principalmente pela maneira em que o código é executado pelos núcleos. Na Epiphany, cada core possui seu próprio stack de instruções e podem executá-los independente dos outros cores. Em Cuda, todos os cores executam a mesma instrução ao mesmo tempo, o que reduz o espaço de memória designado para instruções e pode facilitar a sincronia entre acesso a memória e transferência de dados. Outra característica dessa arquitetura é existência de 3 níveis de memória. Uma global para acesso direto de qualquer core, uma compartilhada entre os cores pertencentes ao mesmo bloco e um espaço para constantes de rápido acesso compartilhado entre todos os cores, porém de somente leitura ("texture memory"). A existência desses diferentes níveis introduz mais flexibilidade de acesso a diferentes tipos de dados. Para paralelizar nossos cálculos, uma abordagem muito parecida com a anterior foi tomada porém sem a necessidade de incluir um core para controle do trafego de dados pois isso pode ser feito pelo host nessa arquitetura devido às ferramentas de sincronia que fazem parte de seu SDK[8]. O algoritmo se dá através dos seguinte passos:

1. Ao receber novo dado, o programa host escreve os dados recebidos em um vetor na memória global.

2. O programa host inicia o kernel na GPU.

3. Cada core copia seu dado da memória global, efetua o cálculo e transfere para um vetor de resposta novamente na memória global.

4. Através da função `"cudaDeviceSynchronize();"` o host espera pelo fim da execução do kernel

5. Host recolhe o resultado da memória global e envia de volta ao cliente

O código completo se encontra no apêndice I.

3.5 Hardware in the loop

Para poder simular um ambiente real onde um usuário interage com o sistema, foi desenvolvido o que é chamado de "Hardware In the Loop"(HIL) conforme discutido anteriormente. Esse ambiente possui uma representação virtual em 3D da plataforma (Fig. 3.1) que é controlada por um usuário através de um um joystick conectado via USB ao PC. Foi usado novamente as ferramentas de simulação e visualização do Matlab para esse propósito. O mundo e objetos virtuais são criados usando o "VRML editor"do Matlab. A atual atitude da plataforma define a posição dos pontos do apoio desejados que é enviado ao hardware de teste juntamente com a posição virtual dos excêntricos. Criou-se também uma representação de cada ponto de apoio (Fig. 3.2) para ilustrar a posição atual dos excêntricos juntamente com a posição de referência desejada.

Para criar todas as representações visuais, é necessário um tempo grande de processamento em relação aos tempos de cálculo, o que faz com que a frequência de amostragem seja baixa em relação às frequências máximas possíveis pelos hardwares testados. O objetivo desse sistema serve somente para ilustrar o uso e não como instrumento de métrica de desempenho.

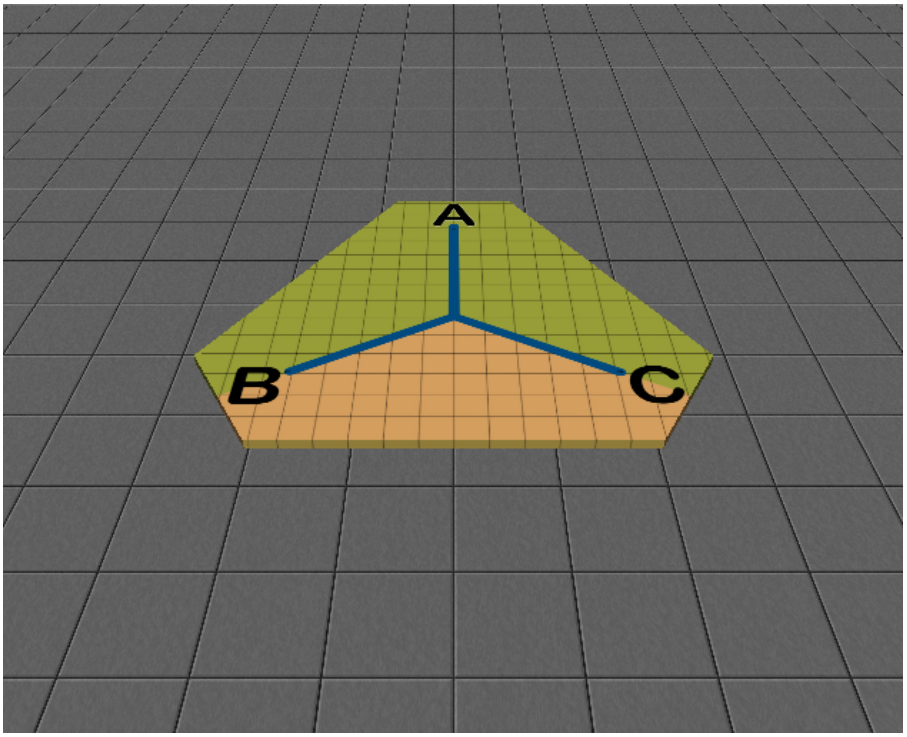


Figura 3.1: Representação em 3D da plataforma usado no sistema HIL

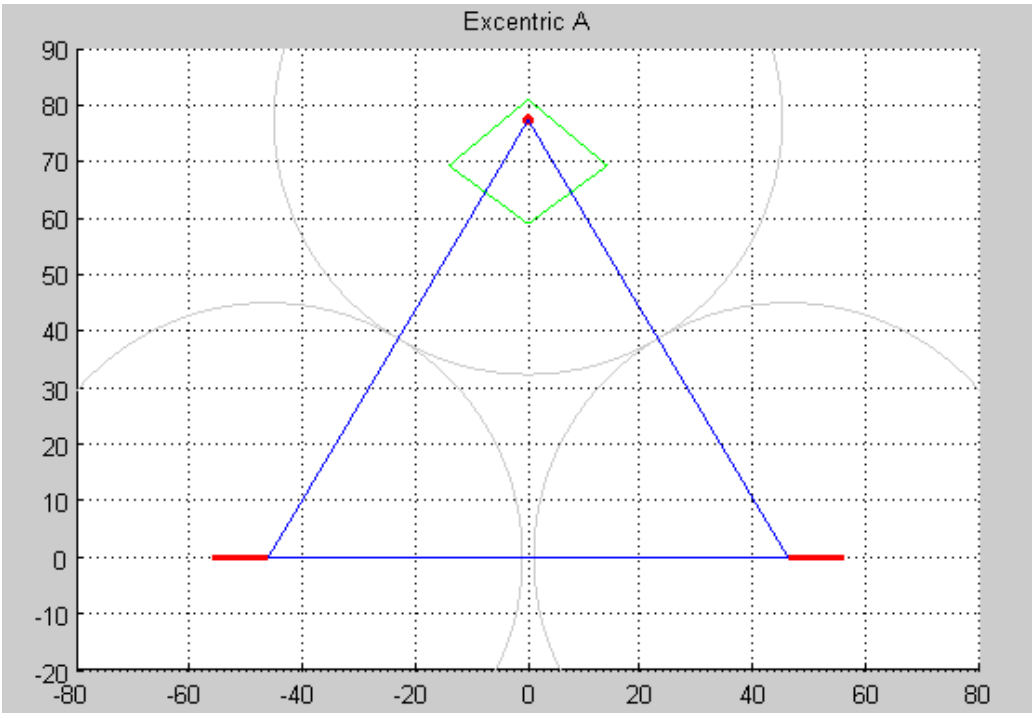


Figura 3.2: Representação em 2D de um ponto de apoio usado no sistema HIL

Capítulo 4

Resultados

Os dados obtidos através da metodologia discutido nesse trabalho se encontram nas tabelas 4.1 e 4.2.

Tabela 4.1: Média de tempo obtido pelos hardwares em teste.

Sistema	**Comunicação**	**Completo**
PC	0.205 ± 0.046 ms	0.280 ± 0.065 ms
Arduino	8.003 ± 2.706 ms	21.349 ± 0.469 ms
Rasp. Pi	0.528 ± 0.041 ms	0.775 ± 0.058 ms
Parallella (Seq)	0.224 ± 0.015 ms	0.306 ± 0.020 ms
Parallella (Par)	0.250 ± 0.017 ms	2.836 ± 0.107 ms
Jetson (Seq)	0.353 ± 0.180 ms	0.391 ± 0.211 ms
Jetson (Par)	1.116 ± 0.285 ms	1.469 ± 0.321 ms
OSX (Seq)	0.396 ± 0.071 ms	0.433 ± 0.084 ms
OSX (CUDA)	0.382 ± 0.085 ms	0.418 ± 0.085 ms

Tabela 4.2: Média de tempo de processamento do Matlab.

Matlab	Tempo Médio por Iteração
Comunicação UDP	5.690 ± 3.032 ms
Interface HIL	83.516 ± 0.866 ms
Cálculos Cinemática	0.119 ± 0.008 ms

Tabela 4.3: Norma do erro de cálculo.

Representação	Norma do erro
Double	1.022 x 10^{-13}
Conversão Float	1.639 x 10^{-4}

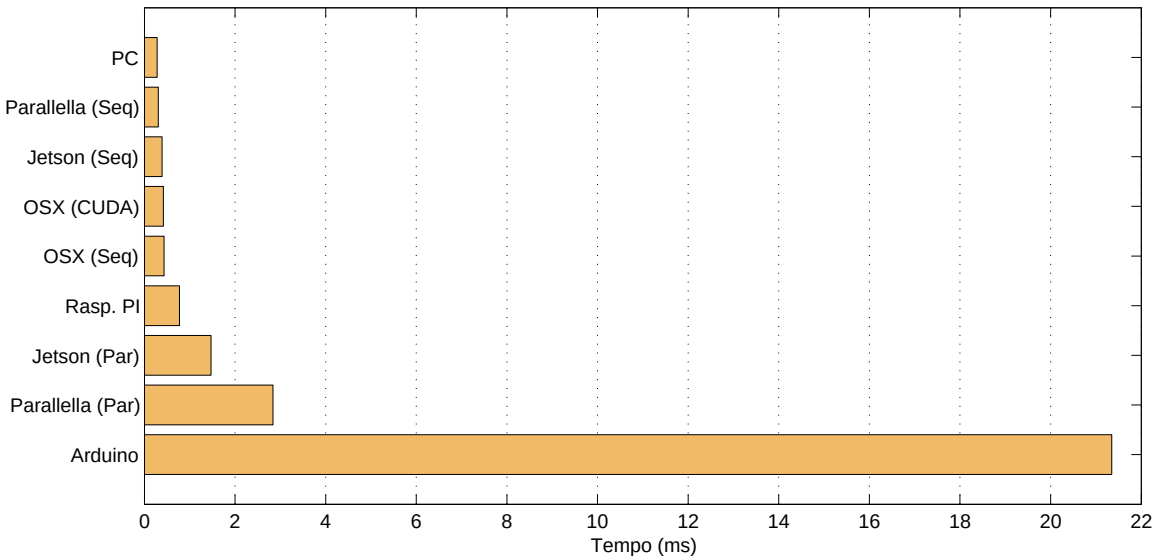


Figura 4.1: Rank dos tempos obtidos por hardware. (Menor é melhor). "S"representa o algoritmo sequencial e "P"em paralelo.

Capítulo 5

Conclusões e Discussões

Esse trabalho teve como foco principal trazer melhorias a um sistema já existente pertencente ao Laboratório Nacional de Luz Síncrotron. Durante o período de estágio do autor nessa instituição, surgiu-se a oportunidade de adquirir conhecimentos mais aprofundados sobre a construção e funcionamento de uma plataforma do tipo Stewart. Observando a abordagem feita pela equipe responsável ao desenvolvimento desse equipamento notou-se que poderia haver espaço para melhorias na parte de controle de movimento.

O primeiro item que chamou a atenção foram os tempo envolvidos no cálculo da cinemática reversa. De acordo com o relatório técnico de construção do equipamento[5], o sistema de controle fazia uso de funções do Matlab e exigia em média de 30 a 150 ms para efetuar um cálculo de reposicionamento da plataforma sendo que esse cálculo envolvia apenas a atitude inicial e final da plataforma. Notou-se também que o sistema exigia um PC conectado à mesma rede do equipamento para efetuar tais cálculos de posicionamento, elevando ainda mais o custo final do projeto. Buscou-se portanto criar uma solução ao cálculo de posicionamento que reduzisse os tempos envolvidos o suficiente para introduzir um número maior de pontos entre a atitude de origem e destino da plataforma, suavizando o movimento e abrindo oportunidades a controle de movimentos mais elaborados. Além disso, procurou-se usar sistemas de hardwares de baixo custo e de menores dimensões físicas que fossem capazes de lidar com o custo computacional para que o controle se tornasse embarcado no equipamento. Os primeiros resultados feitos com o Matlab (Tabela 4.2) demonstraram que a comunicação via rede de pequenos pacotes de dados exige tempos muito superiores quando comparados com comunicações de mesmo tipo gerenciadas por executável compilado a partir de código C (Tabela 4.1). Como exemplo, um PC com o sistema operacional Linux demonstrou ganhos de até 27 vezes, reduzindo de 5.6 ms para 0.205 ms o tempo de transmissão e recepção de um pacote de dados via UDP.

O Matlab pode executar funções via script compilado em arquivos MEX ou via script puro o qual é mais comumente empregado. A tradução desse script durante execução adiciona ainda mais tempo computacional às funções chamadas, o que talvez possa ter contribuído aos elevados tempo obtidos pelos desenvolvedores do equipamento. Para evitar esses atrasos optou-se pela geração automática de código em C que foi compilado diretamente no hardware de teste, otimizando ainda mais os cálculos de posicionamento.

Os desenvolvedores do equipamento abordaram a solução de posicionamento da plataforma através da cinemática direta do modelo matemática aplicando uma função do Matlab chamada `"fsolve"` que faz uso de métodos iterativos na solução de sistemas com múltiplas variáveis. Nesse trabalho, optou-se por aplicar a cinemática reversa através da matriz Jacobiana. Apesar de aumentar o número de equações a serem resolvidas, a solução final é alcançada em um tempo computacional menor. A única exigência desse método é a criação de um planejador de trajetória pois a matriz Jacobiana só é valida em uma região próxima da posição atual e portanto só é válida para pequenos deslocamentos.

Aplicando as modificações descritas acima já foram suficientes para conseguir tempos de cálculos inferiores a 0.1 ms em um PC (Tabela 4.1). Tal desempenho é mais que suficiente para a aplicação em questão e o foco torna-se portanto em conseguir desempenhos parecidos porém em um hardware de menor custo e de menores dimensões físicas.

O Arduíno, apesar de ser um hardware muito simples, possui uma enorme versatilidade e se mostrou capaz de realizar os cálculos necessários com dupla precisão (double float) porém com tempos muito maiores com relação aos outros equipamentos, mas ainda inferior aos tempos de referência. Infelizmente, o "shield"que proporciona a comunicação via rede se mostrou instável causando interrupções prematuras durante o teste. O número máximo de cálculos consecutivos obtidos nessa placa sem falha foram por volta de 30 cálculos apenas.

A Raspberry Pi se mostrou ser uma ótima opção para esse projeto. Além do preço reduzido, ele é de fácil operação e programação devido ao Linux OS e a grande comunidade de usuários. Os tempos obtidos foram satisfatórios e poderiam ser até menores se possuísse uma placa de rede gigabit.

A Parallella com seu ARM dual core e sua placa gigabit obteve os melhores resultados e os menores desvios-padrão para o código sequencial. A Parallella possui um preço acessível e é a placa de menor dimensão física. Porém ela exige refrigeração ativa para se manter abaixo da temperatura máxima sugerida pelo fabricante ($<70°C$). Explorar o paralelismo através do processador Epiphany desta placa não é trivial. É necessário se familiarizar com sua arquitetura incomum e saber lidar com inúmeros problemas durante compilação. O compilador, juntamente

com seu SDK, aparentemente se encontra no seu estágio inicial de desenvolvimento e a busca de informações depende de uma comunidade relativamente pequena de desenvolvedores. Os tempos obtidos com o overhead de paralelização foram bons o suficiente para apresentar ganhos teóricos de desempenho. Como cada core é responsável por apenas um ponto de apoio, o tempo necessário para cálculo seria em tese parecido com os tempos obtidos com o processador ARM dividido por 3. Somando esse tempo com o tempo necessário para comunicação via rede e overhead de paralelização, teríamos tempos inferiores quando comparados ao sequencial. Porém, tempos muito maiores foram obtidos para cálculo cujo motivo não foi encontrado mas apontam para a compilação ou para a falta de suporte às funções seno e cosseno implementadas em hardware. Caso seja possível diminuir os tempo de cálculo, existe a possibilidade de explorar essa arquitetura adicionando o controle de outras plataformas à mesma placa pois como apenas 4 cores foram usados para cálculo, seria possível de fazer uso dos outros 12 cores remanescentes ao adicionar mais 9 pontos de apoio, ou seja, controlar mais 3 plataformas sem grande impacto ao tempo total. Esse escalabilidade é um dos atrativos na computação paralela.

A arquitetura CUDA já vem sendo desenvolvida a um certo tempo e por isso está bem documentada além de existir uma grande comunidade de desenvolvedores que oferecem significativo suporte. Buscou-se explorar essa arquitetura primeiramente através do uso da GPU de uma placa gráfica da NVidia para depois implementar o mesmo algoritmo na placa Jetson. Os tempos de comunicação entre o computador contendo a GPU e o PC de teste foram maiores do que o esperado. Uma possível explicação pode ser o diferente sistema operacional (OSX 10.10.5) nesse equipamento. Nota-se também que os tempos de comunicação incluindo os overheads de paralelização são menores que o sequencial. Como a GPU que se encontra nesse equipamento não possui suporte à variáveis do tipo double, o pacote de dados transferido é metade do original e isso consequentemente afeta também os tempos de cálculo. Fazendo medições de tempo usando as próprias bibliotecas do CUDA, foram observados tempos de overhead em média de 33μs e adicionando o tempo de cálculo passou a ser em média 72μs. Com isso, essa plataforma se torna a mais rápida para os cálculos mas não a mais rápida no tempo total devido ao tempo gasto na comunicação de rede. Comparando a norma dos erros obtidos (tabela 4.3) observou-se que a conversão de double para float introduz um acréscimo considerável na norma do erro que pode ser motivo de preocupação dependendo da precisão esperada.

A Jetson, executando o algoritmo sequencial, também apresentou tempos de comunicação um pouco maiores que a Parallela, porém os tempos de cálculo foram bem menores devido a sua maior frequência de clock. Não eram esperados tempos de cálculo em paralelo tão altos para o overhead de paralelização e para os cálculos. O algoritmo usado foi o mesmo para

ambas as plataformas CUDA e talvez a Jetson exija um algoritmo dedicado que explore melhor sua arquitetura tirendo proveitos dos recursos implementados em hardware. Por apresentar 196 cores, também é provável que seu benefício apareça somente com o crescimento da carga computacional.

Podemos concluir que trocar o Matlab por executáveis compilados a partir da linguagem C e aplicando a cinemática reversa a partir do cálculo da Jacobiana é suficiente para reduzir o custo computacional a ponto que um simples processador ARM de apenas um núcleo é suficiente para obter os desempenhos esperados. A paralelização desse processo não é necessário mas pode ser considerada em casos onde há mais equipamentos a serem controlados ou no caso de sistemas mais custosos computacionalmente.

Apêndice A

Matriz Jacobiana

Para calcular as derivadas parciais da matriz Jacobiana, foi usado as funções matemáticas simbólicas do Matlab. Cada elemento calculado da matriz Jacobiana se encontra abaixo:

$$J = \begin{bmatrix} \frac{\partial f_1}{\partial \alpha} & \frac{\partial f_1}{\partial \beta} \\ \frac{\partial f_2}{\partial \alpha} & \frac{\partial f_2}{\partial \beta} \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} \\ r_{21} & r_{22} \end{bmatrix}$$

$$r_{11} = -r\sin(\alpha) - L\sin\left(\arcsin\left(\frac{\sqrt{k_1}}{2L}\right) - \frac{\pi}{2} + \arcsin\left(\frac{k_3}{\sqrt{k_1}}\right)\right) * \\ * \left(\frac{\frac{r\cos(\alpha)}{\sqrt{k_1}} + \frac{k_2 k_3}{2k_1^{\frac{3}{2}}}}{\sqrt{1 - \frac{k_3^2}{k_1}}} - \frac{k_2}{4L\sqrt{k_1\left(1 - \frac{k_1}{4L^2}\right)}}\right)$$

$$k_1 = k_3^2 + k_4^2$$

$$k_2 = 2r\left(k_4\sin(\alpha) - k_3\cos(\alpha)\right)$$

$$k_3 = a_{0y} - b_{0y} + e\left(\sin(\alpha) - \sin(\beta)\right)$$

$$k_4 = a_{0x} - b_{0x} + e\left(\cos(\alpha) - \cos(\beta)\right)$$

$$r_{12} = L\sin\left(\arcsin\left(\frac{\sqrt{h_1}}{2L}\right) - \frac{\pi}{2} + \arcsin\left(\frac{h_3}{\sqrt{h_1}}\right)\right) * \\ * \left(\frac{\frac{r\cos(\beta)}{\sqrt{h_1}} + \frac{h_2 h_3}{2h_1^{\frac{3}{2}}}}{\sqrt{1 - \frac{h_3^2}{h_1}}} - \frac{h_2}{4L\sqrt{h1\left(1 - \frac{h_1}{4L^2}\right)}}\right)$$

$$h_1 = h_3^2 + \left(a_{0x} - b_{0x} + r(\cos(\alpha) - \cos(\beta))\right)^2$$

$$h_2 = 2r\left(\sin(\beta)(a_{0x} - b_{0x} + r\cos(\alpha) - r\cos(\beta)) - \cos(\beta)\right)$$

$$h_3 = a_{0y} - b_{0y} + r\left(\sin(\alpha) - \sin(\beta)\right)$$

$$r_{21} = -r\cos(\alpha) - L\cos\left(\arcsin\left(\frac{\sqrt{k_1}}{2L}\right) - \frac{\pi}{2} + \arcsin\left(\frac{k_3}{\sqrt{k_1}}\right)\right) *$$
$$* \left(\frac{\frac{e\cos(\alpha)}{\sqrt{k_1}} + \frac{k_2 k_3}{2k_1^{\frac{3}{2}}}}{\sqrt{1 - \frac{k_3^2}{k_1}}} - \frac{k_2}{4L\sqrt{k_1\left(1 - \frac{k_1}{4L^2}\right)}}\right)$$

$$r_{22} = L\cos\left(\arcsin\left(\frac{\sqrt{h_1}}{2L}\right) - \frac{\pi}{2} + \arcsin\left(\frac{h_3}{\sqrt{h_1}}\right)\right) *$$
$$* \left(\frac{\frac{r\cos(\beta)}{\sqrt{h_1}} + \frac{h_2 h_3}{2h_1^{\frac{3}{2}}}}{\sqrt{1 - \frac{h_3^2}{h_1}}} - \frac{h_2}{4L\sqrt{h1\left(1 - \frac{h_1}{4L^2}\right)}}\right)$$

Apêndice B

Comparação Numérica entre DLS e Jacobiana Inversa

Para poder ter uma ideia de como a DLS se comporta em relação a inversa da Jacobiana, iremos comparar numericamente o valor da norma 2 dessas matrizes em regiões próximas da singularidade. De acordo com a equação 2.3, a DLS depende de um parâmetro λ. Para $\lambda = 0$ a DLS é equivalente à inversa da Jacobiana. Esse parâmetro reduz a instabilidade numérica próximo da singularidade porém insere um amortecimento na dinâmica do sistema.

Faremos uma comparação entre a norma da inversa da Jacobiana com as normas da DSL para diferentes valores de λ. Como o determinante da Jacobiana é 0 na singularidade, usaremos esse valor como métrica de "distância"desse ponto à singularidade.

Nota-se na figura B.1 que a DSL é praticamente nula na singularidade e surge uma diferença entre as normas proporcional à λ.

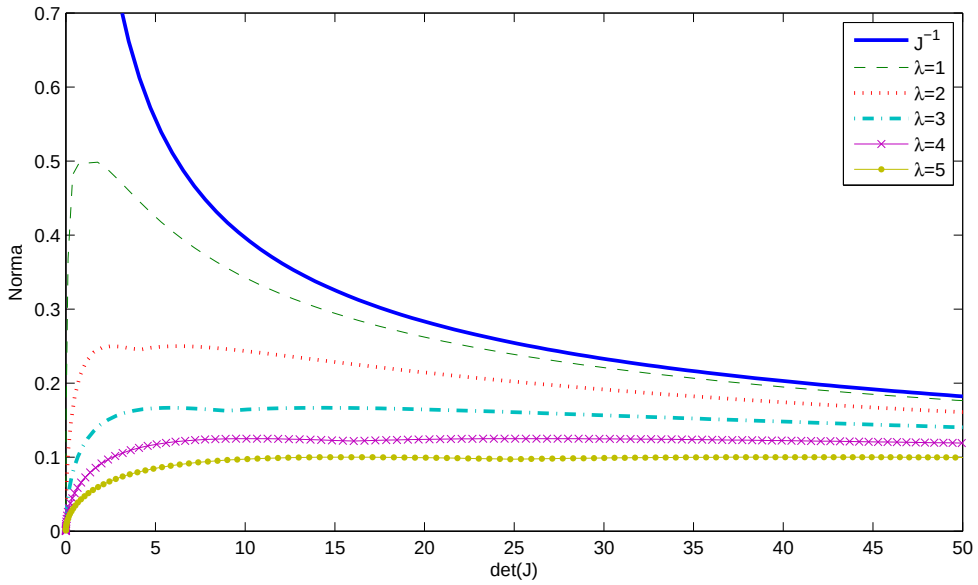


Figura B.1: Comparação numérica entre normas das matrizes J^{-1} e DSL para diferentes valores de λ próximo à singularidade

Apêndice C

Scripts em Matlab para Cálculo da Cinemática Reversa

Os scripts abaixo constituem o cálculo da cinemática reversa no Matlab que funcionaram como fonte para a geração de código automático na linguagem C. O primeiro código abaixo é usado como exemplo para que o aplicativo reconheça os tipos de dados que serão usados como entrada e saída de cada função. Os demais scripts se encontram em seguida.

`sample.m`

```
% Script gerado como exemplo de calculo
clc; close all; clear all;

% Entrada ficticia para exemplo
rxvec = [1 -0.1309 3.2725 -0.1309 3.2725 -0.1309 3.2725 0 76 0 76 0 76];

% Calculo da cinematica direta para encontrar os pontos de apoio baseado
% nas posicoes atuais dos excentricos. Retorna o erro entre a posicao atual
% dos pontos de apoio e as posicoes desejadas dos mesmos.
dE = direct(rxvec);

% Calculo das matrizes jacobianas para a posicao atual dos pontos de apoio
Jm = jacobian(rxvec);

% Valor de lambda para ser usado no calculo da matriz DLS.
lambda = 5;

% Calculo das matrizes DLS.
dlsm = dls(Jm, lambda);

% Calculo das variacoes angulares dos excentricos para seguir referencia
dangles = mx_multiply(dlsm, dE);
```

`direct.m`

```
function dE = direct(rxvec)
r = 10;
L = 90;
a0 = [-56 0];
b0 = [56 0];

Aa = r*[cos(rxvec(2)) sin(rxvec(2))]+a0;
Ba = r*[cos(rxvec(3)) sin(rxvec(3))]+b0;
```

```
9
10 Ab = r*[cos(rxvec(4)) sin(rxvec(4))]+a0;
11 Bb = r*[cos(rxvec(5)) sin(rxvec(5))]+b0;
12
13 Ac = r*[cos(rxvec(6)) sin(rxvec(6))]+a0;
14 Bc = r*[cos(rxvec(7)) sin(rxvec(7))]+b0;
15
16 Fa = norm(Ba-Aa);
17 Fb = norm(Bb-Ab);
18 Fc = norm(Bc-Ac);
19
20 theta = asin((Ba(2)-Aa(2))/Fa);
21 epson = asin(Fa/(2*L));
22 lambda = pi/2 - epson + theta;
23 Ea = L*[cos(lambda) sin(lambda)]+Aa;
24
25 theta = asin((Bb(2)-Ab(2))/Fb);
26 epson = asin(Fb/(2*L));
27 lambda = pi/2 - epson + theta;
28 Eb = L*[cos(lambda) sin(lambda)]+Ab;
29
30 theta = asin((Bc(2)-Ac(2))/Fc);
31 epson = asin(Fc/(2*L));
32 lambda = pi/2 - epson + theta;
33 Ec = L*[cos(lambda) sin(lambda)]+Ac;
34
35 dEa = [rxvec(8) rxvec(9)] - Ea;
36 dEb = [rxvec(10) rxvec(11)] - Eb;
37 dEc = [rxvec(12) rxvec(13)] - Ec;
38
39 dE = [dEa dEb dEc];
40 end
```

jacobian.m

```
1  function [Jm] = jacobian(rxvec)
2  % Constants
3  r = 10;
4  L = 90;
5  a0x = -56;
6  a0y = 0;
7  b0x = 56;
8  b0y = 0;
9
10 %Jacobian for A
11 alpha = rxvec(2);
12 beta = rxvec(3);
13
14 k3 = a0y - b0y + r*(sin(alpha) - sin(beta));
15 k4 = a0x - b0x + r*(cos(alpha) - cos(beta));
16 k1 = k3^2+k4^2;
17 k2 = 2*r*(k4*sin(alpha) - k3*cos(alpha));
18 k5 = (((r*cos(alpha))/(sqrt(k1)))+((k2*k3)/(2*sqrt(k1^3))))/(sqrt(1-((k3^2)/k1)));
19 k6 = (k2)/(4*L*sqrt(k1*(1-(k1/(4*L^2)))));
20 k7 = asin(sqrt(k1)/(2*L))-(pi/2)+asin((k3)/(sqrt(k1)));
21
22 h3 = a0y - b0y + r*(sin(alpha) - sin(beta));
23 h2 = 2*r*sin(beta)*(a0x - b0x + r*cos(alpha) - r*cos(beta)) - 2*r*cos(beta)*h3;
24 h1 = h3^2 + (a0x - b0x + r*(cos(alpha) - cos(beta)))^2;
25 h4 = (((r*cos(beta))/(sqrt(h1)))+((h2*h3)/(2*sqrt(h1^3))))/(sqrt(1-((h3^2)/h1)));
26 h5 = (h2)/(4*L*sqrt(h1*(1-(h1/(4*L^2)))));
27 h6 = asin(sqrt(h1)/(2*L))-(pi/2)+asin((h3)/(sqrt(h1)));
28
29 r11 = -r*sin(alpha) - L*sin(k7)*(k5-k6);
30 r12 = L*sin(h6)*(h4-h5);
31 r21 = r*cos(alpha) - L*cos(k7)*(k5-k6);
32 r22 = L*cos(h6)*(h4-h5);
33
34 Jm = [r11 r12 r21 r22];
```

```

%Jacobian for B
alpha = rxvec(4);
beta = rxvec(5);

k3 = a0y - b0y + r*(sin(alpha) - sin(beta));
k4 = a0x - b0x + r*(cos(alpha) - cos(beta));
k1 = k3^2+k4^2;
k2 = 2*r*(k4*sin(alpha) - k3*cos(alpha));
k5 = (((r*cos(alpha))/(sqrt(k1)))+((k2*k3)/(2*sqrt(k1^3))))/(sqrt(1-((k3^2)/k1)));
k6 = (k2)/(4*L*sqrt(k1*(1-(k1/(4*L^2)))));
k7 = asin(sqrt(k1)/(2*L))-(pi/2)+asin((k3)/(sqrt(k1)));

h3 = a0y - b0y + r*(sin(alpha) - sin(beta));
h2 = 2*r*sin(beta)*(a0x - b0x + r*cos(alpha) - r*cos(beta)) - 2*r*cos(beta)*h3;
h1 = h3^2 + (a0x - b0x + r*(cos(alpha) - cos(beta)))^2;
h4 = (((r*cos(beta))/(sqrt(h1)))+((h2*h3)/(2*sqrt(h1^3))))/(sqrt(1-((h3^2)/h1)));
h5 = (h2)/(4*L*sqrt(h1*(1-(h1/(4*L^2)))));
h6 = asin(sqrt(h1)/(2*L))-(pi/2)+asin((h3)/(sqrt(h1)));

r11 = -r*sin(alpha) - L*sin(k7)*(k5-k6);
r12 = L*sin(h6)*(h4-h5);
r21 = r*cos(alpha) - L*cos(k7)*(k5-k6);
r22 = L*cos(h6)*(h4-h5);

Jm = [Jm r11 r12 r21 r22];

%Jacobian for C
alpha = rxvec(6);
beta = rxvec(7);

k3 = a0y - b0y + r*(sin(alpha) - sin(beta));
k4 = a0x - b0x + r*(cos(alpha) - cos(beta));
k1 = k3^2+k4^2;
k2 = 2*r*(k4*sin(alpha) - k3*cos(alpha));
k5 = (((r*cos(alpha))/(sqrt(k1)))+((k2*k3)/(2*sqrt(k1^3))))/(sqrt(1-((k3^2)/k1)));
k6 = (k2)/(4*L*sqrt(k1*(1-(k1/(4*L^2)))));
k7 = asin(sqrt(k1)/(2*L))-(pi/2)+asin((k3)/(sqrt(k1)));

h3 = a0y - b0y + r*(sin(alpha) - sin(beta));
h2 = 2*r*sin(beta)*(a0x - b0x + r*cos(alpha) - r*cos(beta)) - 2*r*cos(beta)*h3;
h1 = h3^2 + (a0x - b0x + r*(cos(alpha) - cos(beta)))^2;
h4 = (((r*cos(beta))/(sqrt(h1)))+((h2*h3)/(2*sqrt(h1^3))))/(sqrt(1-((h3^2)/h1)));
h5 = (h2)/(4*L*sqrt(h1*(1-(h1/(4*L^2)))));
h6 = asin(sqrt(h1)/(2*L))-(pi/2)+asin((h3)/(sqrt(h1)));

r11 = -r*sin(alpha) - L*sin(k7)*(k5-k6);
r12 = L*sin(h6)*(h4-h5);
r21 = r*cos(alpha) - L*cos(k7)*(k5-k6);
r22 = L*cos(h6)*(h4-h5);

Jm = [Jm r11 r12 r21 r22];
end
```

dls.m

```
function dlsm = dls(Jm, lambda)
I = eye(2);

% A
J = [Jm(1) Jm(2); Jm(3) Jm(4)];
A = J'/((J*J'+lambda*lambda*I));
dlsm = [A(1) A(3) A(2) A(4)];

% B
J = [Jm(5) Jm(6); Jm(7) Jm(8)];
A = J'/((J*J'+lambda*lambda*I));
dlsm = [dlsm A(1) A(3) A(2) A(4)];

```

```
% C
J = [Jm(9) Jm(10); Jm(11) Jm(12)];
A = J'/((J*J'+lambda*lambda*I));
dlsm = [dlsm A(1) A(3) A(2) A(4)];
end
```

mx_multiply.m

```
function dang = mx_multiply(dlsm, dE)

%A
J = [dlsm(1) dlsm(2); dlsm(3) dlsm(4)];
da = J*[dE(1); dE(2)];

%B
J = [dlsm(5) dlsm(6); dlsm(7) dlsm(8)];
db = J*[dE(3); dE(4)];

%C
J = [dlsm(9) dlsm(10); dlsm(11) dlsm(12)];
dc = J*[dE(5); dE(6)];

dang = [da' db' dc'];
end
```

Apêndice D

Código fonte em C gerado pelo Matlab

A seguir se encontra as funções em C geradas pelo Matlab a partir dos scripts do apêndice C.

`funlib.c`

```c
#include "funlib.h"

static void mrdivide(const double A[4], const double B[4], double y[4]);
static double norm(const double x[2]);
static void mrdivide(const double A[4], const double B[4], double y[4])
{
  int r1;
  int r2;
  double a21;
  double a22;
  int k;
  if (fabs(B[1]) > fabs(B[0])) {
    r1 = 1;
    r2 = 0;
  } else {
    r1 = 0;
    r2 = 1;
  }

  a21 = B[r2] / B[r1];
  a22 = B[2 + r2] - a21 * B[2 + r1];
  for (k = 0; k < 2; k++) {
    y[k + (r1 << 1)] = A[k] / B[r1];
    y[k + (r2 << 1)] = (A[2 + k] - y[k + (r1 << 1)] * B[2 + r1]) / a22;
    y[k + (r1 << 1)] -= y[k + (r2 << 1)] * a21;
  }
}

static double norm(const double x[2])
{
  double y;
  double scale;
  int k;
  double absxk;
  double t;
  y = 0.0;
  scale = 2.2250738585072014E-308;
  for (k = 0; k < 2; k++) {
    absxk = fabs(x[k]);
    if (absxk > scale) {
      t = scale / absxk;
      y = 1.0 + y * t * t;
      scale = absxk;
    } else {
```

```
      t = absxk / scale;
      y += t * t;
    }
  }

  return scale * sqrt(y);
}

void direct(const double rxvec[13], double dE[6])
{
  double dv0[2];
  double dv1[2];
  double dv2[2];
  double dv3[2];
  double dv4[2];
  double dv5[2];
  double Ba[2];
  double Aa[2];
  double b_Ba[2];
  double Ab[2];
  double Bb[2];
  double Ac[2];
  double Bc[2];
  double b_Bb[2];
  double b_Bc[2];
  int i0;
  double Fa;
  double Fb;
  double Fc;
  double c_Bb;
  double b_Ac;
  double c_Bc;
  double b[2];
  double b_rxvec[2];
  double c_rxvec[2];
  double dv6[2];
  double d_rxvec[2];
  dv0[0] = cos(rxvec[1]);
  dv0[1] = sin(rxvec[1]);
  dv1[0] = cos(rxvec[2]);
  dv1[1] = sin(rxvec[2]);
  dv2[0] = cos(rxvec[3]);
  dv2[1] = sin(rxvec[3]);
  dv3[0] = cos(rxvec[4]);
  dv3[1] = sin(rxvec[4]);
  dv4[0] = cos(rxvec[5]);
  dv4[1] = sin(rxvec[5]);
  dv5[0] = cos(rxvec[6]);
  dv5[1] = sin(rxvec[6]);
  for (i0 = 0; i0 < 2; i0++) {
    Fa = 10.0 * dv0[i0] + (-56.0 + 56.0 * (double)i0);
    Fb = 10.0 * dv1[i0] + (56.0 + -56.0 * (double)i0);
    Fc = 10.0 * dv2[i0] + (-56.0 + 56.0 * (double)i0);
    c_Bb = 10.0 * dv3[i0] + (56.0 + -56.0 * (double)i0);
    b_Ac = 10.0 * dv4[i0] + (-56.0 + 56.0 * (double)i0);
    c_Bc = 10.0 * dv5[i0] + (56.0 + -56.0 * (double)i0);
    Ba[i0] = Fb - Fa;
    b_Bb[i0] = c_Bb - Fc;
    b_Bc[i0] = c_Bc - b_Ac;
    Aa[i0] = Fa;
    b_Ba[i0] = Fb;
    Ab[i0] = Fc;
    Bb[i0] = c_Bb;
    Ac[i0] = b_Ac;
    Bc[i0] = c_Bc;
  }

  Fa = norm(Ba);
  Fb = norm(b_Bb);
```

```
  Fc = norm(b_Bc);
  Fa = (1.5707963267948966 - asin(Fa / 180.0)) + asin((b_Ba[1] - Aa[1]) / Fa);
  b[0] = cos(Fa);
  b[1] = sin(Fa);
  Fa = (1.5707963267948966 - asin(Fb / 180.0)) + asin((Bb[1] - Ab[1]) / Fb);
  b_Ba[0] = cos(Fa);
  b_Ba[1] = sin(Fa);
  Fa = (1.5707963267948966 - asin(Fc / 180.0)) + asin((Bc[1] - Ac[1]) / Fc);
  b_rxvec[0] = rxvec[7];
  b_rxvec[1] = rxvec[8];
  c_rxvec[0] = rxvec[9];
  c_rxvec[1] = rxvec[10];
  dv6[0] = cos(Fa);
  dv6[1] = sin(Fa);
  d_rxvec[0] = rxvec[11];
  d_rxvec[1] = rxvec[12];
  for (i0 = 0; i0 < 2; i0++) {
    dE[i0] = b_rxvec[i0] - (90.0 * b[i0] + Aa[i0]);
  }

  for (i0 = 0; i0 < 2; i0++) {
    dE[i0 + 2] = c_rxvec[i0] - (90.0 * b_Ba[i0] + Ab[i0]);
  }

  for (i0 = 0; i0 < 2; i0++) {
    dE[i0 + 4] = d_rxvec[i0] - (90.0 * dv6[i0] + Ac[i0]);
  }
}

void dls(const double Jm[12], double lambda, double dlsm[12])
{
  double J[4];
  double a;
  double b_J[4];
  int i1;
  int i2;
  double c_J[4];
  double d0;
  int i3;
  static const signed char b[4] = { 1, 0, 0, 1 };

  double b_dlsm[4];
  double c_dlsm[8];
  J[0] = Jm[0];
  J[2] = Jm[1];
  J[1] = Jm[2];
  J[3] = Jm[3];
  a = lambda * lambda;
  for (i1 = 0; i1 < 2; i1++) {
    for (i2 = 0; i2 < 2; i2++) {
      b_J[i2 + (i1 << 1)] = J[i1 + (i2 << 1)];
    }
  }

  for (i1 = 0; i1 < 2; i1++) {
    for (i2 = 0; i2 < 2; i2++) {
      d0 = 0.0;
      for (i3 = 0; i3 < 2; i3++) {
        d0 += J[i1 + (i3 << 1)] * J[i2 + (i3 << 1)];
      }

      c_J[i1 + (i2 << 1)] = d0 + a * (double)b[i1 + (i2 << 1)];
    }
  }

  mrdivide(b_J, c_J, J);
  b_dlsm[0] = J[0];
  b_dlsm[1] = J[2];
  b_dlsm[2] = J[1];
```

```
  b_dlsm[3] = J[3];
  J[0] = Jm[4];
  J[2] = Jm[5];
  J[1] = Jm[6];
  J[3] = Jm[7];
  a = lambda * lambda;
  for (i1 = 0; i1 < 2; i1++) {
    for (i2 = 0; i2 < 2; i2++) {
      b_J[i2 + (i1 << 1)] = J[i1 + (i2 << 1)];
    }
  }

  for (i1 = 0; i1 < 2; i1++) {
    for (i2 = 0; i2 < 2; i2++) {
      d0 = 0.0;
      for (i3 = 0; i3 < 2; i3++) {
        d0 += J[i1 + (i3 << 1)] * J[i2 + (i3 << 1)];
      }

      c_J[i1 + (i2 << 1)] = d0 + a * (double)b[i1 + (i2 << 1)];
    }
  }

  mrdivide(b_J, c_J, J);
  for (i1 = 0; i1 < 4; i1++) {
    c_dlsm[i1] = b_dlsm[i1];
  }

  c_dlsm[4] = J[0];
  c_dlsm[5] = J[2];
  c_dlsm[6] = J[1];
  c_dlsm[7] = J[3];
  J[0] = Jm[8];
  J[2] = Jm[9];
  J[1] = Jm[10];
  J[3] = Jm[11];
  a = lambda * lambda;
  for (i1 = 0; i1 < 2; i1++) {
    for (i2 = 0; i2 < 2; i2++) {
      b_J[i2 + (i1 << 1)] = J[i1 + (i2 << 1)];
    }
  }

  for (i1 = 0; i1 < 2; i1++) {
    for (i2 = 0; i2 < 2; i2++) {
      d0 = 0.0;
      for (i3 = 0; i3 < 2; i3++) {
        d0 += J[i1 + (i3 << 1)] * J[i2 + (i3 << 1)];
      }

      c_J[i1 + (i2 << 1)] = d0 + a * (double)b[i1 + (i2 << 1)];
    }
  }

  mrdivide(b_J, c_J, J);
  memcpy(&dlsm[0], &c_dlsm[0], sizeof(double) << 3);
  dlsm[8] = J[0];
  dlsm[9] = J[2];
  dlsm[10] = J[1];
  dlsm[11] = J[3];
}

void funlib_initialize(void)
{
}

void funlib_terminate(void)
{
}
```

```
252
253 void jacobian(const double rxvec[13], double Jm[12])
254 {
255   double k3;
256   double k4;
257   double k1;
258   double k5;
259   double k6;
260   double k7;
261   double h4;
262   double b_Jm[4];
263   int i4;
264   double c_Jm[8];
265   k3 = 10.0 * (sin(rxvec[1]) - sin(rxvec[2]));
266   k4 = -112.0 + 10.0 * (cos(rxvec[1]) - cos(rxvec[2]));
267   k1 = k3 * k3 + k4 * k4;
268   k4 = 20.0 * (k4 * sin(rxvec[1]) - k3 * cos(rxvec[1]));
269   k5 = (10.0 * cos(rxvec[1]) / sqrt(k1) + k4 * k3 / (2.0 * sqrt(pow(k1, 3.0)))) /
270     sqrt(1.0 - k3 * k3 / k1);
271   k6 = k4 / (360.0 * sqrt(k1 * (1.0 - k1 / 32400.0)));
272   k7 = (asin(sqrt(k1) / 180.0) - 1.5707963267948966) + asin(k3 / sqrt(k1));
273   k1 = 10.0 * (sin(rxvec[1]) - sin(rxvec[2]));
274   k3 = 20.0 * sin(rxvec[2]) * ((-112.0 + 10.0 * cos(rxvec[1])) - 10.0 * cos
275     (rxvec[2])) - 20.0 * cos(rxvec[2]) * k1;
276   k4 = -112.0 + 10.0 * (cos(rxvec[1]) - cos(rxvec[2]));
277   k4 = k1 * k1 + k4 * k4;
278   h4 = (10.0 * cos(rxvec[2]) / sqrt(k4) + k3 * k1 / (2.0 * sqrt(pow(k4, 3.0)))) /
279     sqrt(1.0 - k1 * k1 / k4);
280   k3 /= 360.0 * sqrt(k4 * (1.0 - k4 / 32400.0));
281   k4 = (asin(sqrt(k4) / 180.0) - 1.5707963267948966) + asin(k1 / sqrt(k4));
282   b_Jm[0] = -10.0 * sin(rxvec[1]) - 90.0 * sin(k7) * (k5 - k6);
283   b_Jm[1] = 90.0 * sin(k4) * (h4 - k3);
284   b_Jm[2] = 10.0 * cos(rxvec[1]) - 90.0 * cos(k7) * (k5 - k6);
285   b_Jm[3] = 90.0 * cos(k4) * (h4 - k3);
286   k3 = 10.0 * (sin(rxvec[3]) - sin(rxvec[4]));
287   k4 = -112.0 + 10.0 * (cos(rxvec[3]) - cos(rxvec[4]));
288   k1 = k3 * k3 + k4 * k4;
289   k4 = 20.0 * (k4 * sin(rxvec[3]) - k3 * cos(rxvec[3]));
290   k5 = (10.0 * cos(rxvec[3]) / sqrt(k1) + k4 * k3 / (2.0 * sqrt(pow(k1, 3.0)))) /
291     sqrt(1.0 - k3 * k3 / k1);
292   k6 = k4 / (360.0 * sqrt(k1 * (1.0 - k1 / 32400.0)));
293   k7 = (asin(sqrt(k1) / 180.0) - 1.5707963267948966) + asin(k3 / sqrt(k1));
294   k1 = 10.0 * (sin(rxvec[3]) - sin(rxvec[4]));
295   k3 = 20.0 * sin(rxvec[4]) * ((-112.0 + 10.0 * cos(rxvec[3])) - 10.0 * cos
296     (rxvec[4])) - 20.0 * cos(rxvec[4]) * k1;
297   k4 = -112.0 + 10.0 * (cos(rxvec[3]) - cos(rxvec[4]));
298   k4 = k1 * k1 + k4 * k4;
299   h4 = (10.0 * cos(rxvec[4]) / sqrt(k4) + k3 * k1 / (2.0 * sqrt(pow(k4, 3.0)))) /
300     sqrt(1.0 - k1 * k1 / k4);
301   k3 /= 360.0 * sqrt(k4 * (1.0 - k4 / 32400.0));
302   k4 = (asin(sqrt(k4) / 180.0) - 1.5707963267948966) + asin(k1 / sqrt(k4));
303   for (i4 = 0; i4 < 4; i4++) {
304     c_Jm[i4] = b_Jm[i4];
305   }
306
307   c_Jm[4] = -10.0 * sin(rxvec[3]) - 90.0 * sin(k7) * (k5 - k6);
308   c_Jm[5] = 90.0 * sin(k4) * (h4 - k3);
309   c_Jm[6] = 10.0 * cos(rxvec[3]) - 90.0 * cos(k7) * (k5 - k6);
310   c_Jm[7] = 90.0 * cos(k4) * (h4 - k3);
311   k3 = 10.0 * (sin(rxvec[5]) - sin(rxvec[6]));
312   k4 = -112.0 + 10.0 * (cos(rxvec[5]) - cos(rxvec[6]));
313   k1 = k3 * k3 + k4 * k4;
314   k4 = 20.0 * (k4 * sin(rxvec[5]) - k3 * cos(rxvec[5]));
315   k5 = (10.0 * cos(rxvec[5]) / sqrt(k1) + k4 * k3 / (2.0 * sqrt(pow(k1, 3.0)))) /
316     sqrt(1.0 - k3 * k3 / k1);
317   k6 = k4 / (360.0 * sqrt(k1 * (1.0 - k1 / 32400.0)));
318   k7 = (asin(sqrt(k1) / 180.0) - 1.5707963267948966) + asin(k3 / sqrt(k1));
319   k1 = 10.0 * (sin(rxvec[5]) - sin(rxvec[6]));
320   k3 = 20.0 * sin(rxvec[6]) * ((-112.0 + 10.0 * cos(rxvec[5])) - 10.0 * cos
```

```
    (rxvec[6])) - 20.0 * cos(rxvec[6]) * k1;
  k4 = -112.0 + 10.0 * (cos(rxvec[5]) - cos(rxvec[6]));
  k4 = k1 * k1 + k4 * k4;
  h4 = (10.0 * cos(rxvec[6]) / sqrt(k4) + k3 * k1 / (2.0 * sqrt(pow(k4, 3.0)))) /
    sqrt(1.0 - k1 * k1 / k4);
  k3 /= 360.0 * sqrt(k4 * (1.0 - k4 / 32400.0));
  k4 = (asin(sqrt(k4) / 180.0) - 1.5707963267948966) + asin(k1 / sqrt(k4));
  memcpy(&Jm[0], &c_Jm[0], sizeof(double) << 3);
  Jm[8] = -10.0 * sin(rxvec[5]) - 90.0 * sin(k7) * (k5 - k6);
  Jm[9] = 90.0 * sin(k4) * (h4 - k3);
  Jm[10] = 10.0 * cos(rxvec[5]) - 90.0 * cos(k7) * (k5 - k6);
  Jm[11] = 90.0 * cos(k4) * (h4 - k3);
}

void mx_multiply(const double dlsm[12], const double dE[6], double dang[6])
{
  double b_dlsm[4];
  double b_dE[2];
  double c_dlsm[2];
  double d_dlsm[4];
  double c_dE[2];
  double e_dlsm[2];
  double f_dlsm[4];
  double d_dE[2];
  double g_dlsm[2];
  int i5;
  int i6;
  b_dlsm[0] = dlsm[0];
  b_dlsm[2] = dlsm[1];
  b_dlsm[1] = dlsm[2];
  b_dlsm[3] = dlsm[3];
  b_dE[0] = dE[0];
  b_dE[1] = dE[1];
  d_dlsm[0] = dlsm[4];
  d_dlsm[2] = dlsm[5];
  d_dlsm[1] = dlsm[6];
  d_dlsm[3] = dlsm[7];
  c_dE[0] = dE[2];
  c_dE[1] = dE[3];
  f_dlsm[0] = dlsm[8];
  f_dlsm[2] = dlsm[9];
  f_dlsm[1] = dlsm[10];
  f_dlsm[3] = dlsm[11];
  d_dE[0] = dE[4];
  d_dE[1] = dE[5];
  for (i5 = 0; i5 < 2; i5++) {
    c_dlsm[i5] = 0.0;
    for (i6 = 0; i6 < 2; i6++) {
      c_dlsm[i5] += b_dlsm[i5 + (i6 << 1)] * b_dE[i6];
    }

    e_dlsm[i5] = 0.0;
    for (i6 = 0; i6 < 2; i6++) {
      e_dlsm[i5] += d_dlsm[i5 + (i6 << 1)] * c_dE[i6];
    }

    g_dlsm[i5] = 0.0;
    for (i6 = 0; i6 < 2; i6++) {
      g_dlsm[i5] += f_dlsm[i5 + (i6 << 1)] * d_dE[i6];
    }

    dang[i5] = c_dlsm[i5];
  }

  for (i5 = 0; i5 < 2; i5++) {
    dang[i5 + 2] = e_dlsm[i5];
  }

  for (i5 = 0; i5 < 2; i5++) {
```

```
390       dang[i5 + 4] = g_dlsm[i5];
391     }
392   }
```

Apêndice E

Código fonte em C usado na medição de desempenho

O código a seguir foi usado para medição de desempenho dos hardwares em teste. O algoritmo carrega de um arquivo a trajetória pré calculada, envia os dados ponto a ponto para cálculo no hardware de teste via comunicação UDP/IP e mede o tempo gasto entre o envio do dado e o recebimento do mesmo após cálculo no hardware de teste.

Os tempos de cada iteração são armazenados em um vetor para que, após a conclusão do teste, seja calculado o tempo médio gasto em cada iteração e o desvio padrão dessa média.

`cli_max_calc.c`

```c
/*
  Client for time evaluation
  Complete calculation
  By Nilson Pereira
*/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <sys/time.h>
#include "mylib.h"

// Define Endereco IP do hardware de teste e as portas de comunicacao
#define SERVER "192.168.2.110"
#define BUFLEN 256
#define MYPORT 50101
#define REMOTEPORT 50102

int main(int argc, char *argv[]){
  int i, samples, tag;
  char *filename;
  FILE *fp, *fr;
  double *path_x, *path_y, *alfa_vec, *beta_vec;
  double alfa, beta;
  int s, recv_len;
  double rxvec[7];
  double txvec[13];
  struct sockaddr_in si_me, si_other;
```

```
31    socklen_t fromlen = sizeof(si_other);
32    int slen = sizeof(si_other);
33    struct timeval start, end, lstart, lend;
34    float testtime, avg, std;
35    float *timevec;
36    int goodmsg, badmsg;
37
38    // Abre arquivo e carrega os dados
39    if(argc == 1){
40      printf("error: No file defined\n");
41      return(0);
42    }
43    filename = argv[1];
44    fp = fopen(filename,"r");
45    if(fp == NULL){
46      perror("Could not open file");
47      return(0);
48    }
49    fscanf(fp, "%d", &samples);
50    path_x = malloc(samples*sizeof(double));
51    path_y = malloc(samples*sizeof(double));
52    alfa_vec = malloc(samples*sizeof(double));
53    beta_vec = malloc(samples*sizeof(double));
54    timevec = malloc(samples*sizeof(double));
55
56    for(i=0;i<samples;i++){
57      fscanf(fp, "%lf %lf", &path_x[i], &path_y[i]);
58    }
59
60    // Prepara o socket para comunicacao UDP
61    if ((s=socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP)) == -1)
62    {
63        die("socket");
64    }
65
66    memset((char *) &si_me, 0, sizeof(si_me));
67    memset((char *) &si_other, 0, sizeof(si_other));
68
69    si_me.sin_family = AF_INET;
70    si_me.sin_port = htons(MYPORT);
71    si_me.sin_addr.s_addr = htonl(INADDR_ANY);
72
73    si_other.sin_family = AF_INET;
74    si_other.sin_port = htons(REMOTEPORT);
75
76    if (inet_aton(SERVER , &si_other.sin_addr) == 0)
77    {
78        fprintf(stderr, "inet_aton() failed\n");
79        exit(1);
80    }
81
82    // Inicializa o socket
83    if( bind(s , (struct sockaddr*)&si_me, sizeof(si_me) ) == -1)
84    {
85        die("bind");
86    }
87
88    // Prepara as variaveis iniciais
89    printf("Running with %d calculations...\n", samples);
90    goodmsg = 0;
91    badmsg = 0;
92    tag = 1;
93    alfa = -0.1309;
94    beta = 3.2725;
95
96    // Loop principal de teste
97    gettimeofday(&start, NULL);
98    for(i=0;i<samples;i++){
99      txvec[0] = tag;
```

```
100     txvec[1] = alfa;
101     txvec[2] = beta;
102     txvec[3] = txvec[1];
103     txvec[4] = txvec[2];
104     txvec[5] = txvec[1];
105     txvec[6] = txvec[2];
106     txvec[7] = path_x[i];
107     txvec[8] = path_y[i];
108     txvec[9] = txvec[7];
109     txvec[10] = txvec[8];
110     txvec[11] = txvec[7];
111     txvec[12] = txvec[8];
112
113     // Inicio da contagem de tempo
114     gettimeofday(&lstart, NULL);
115
116     // Envia os dados via UDP para hardware de teste
117     if (sendto(s, &txvec, 13*sizeof(double) , 0 , (struct sockaddr *) &si_other, slen)==-1)
118     {
119         die("sendto()");
120     }
121
122     // Espera os dados de retorno  (Blocking call)
123     if ((recv_len = recvfrom(s, &rxvec, 7*sizeof(double), 0, (struct sockaddr *) &si_other, &fromlen)) == -1)
124     {
125         die("recvfrom()");
126     }
127
128     // Fim da contagem de tempo e armazenamento de dados
129     gettimeofday(&lend, NULL);
130     timevec[i] = elapsed(&lstart, &lend);
131     alfa += rxvec[1];
132     beta += rxvec[2];
133     alfa_vec[i] = alfa;
134     beta_vec[i] = beta;
135
136     // Conparacao entre tag enviada e tag recebida
137     if(rxvec[0] == tag){
138       goodmsg++;
139     }else{
140       badmsg++;
141     }
142     tag++;
143   }
144   gettimeofday(&end, NULL);
145
146   // Analise dos dados apos fim do teste e imprime na tela os resultados
147   testtime = elapsed(&start, &end);
148   avg = average(timevec, samples);
149   std = stdev(timevec, avg, samples);
150   printf("\nTest time: %.3f ms\n", testtime);
151   printf("Good: %d \tBad: %d\n", goodmsg, badmsg);
152   printf("Loop time average: %.3f ms +- %.3f ms\n\n", avg, std);
153   for(i=0;i<10;i++){
154     printf("alfa: %.15f \tbeta: %.15f \n", alfa_vec[i], beta_vec[i]);
155   }
156
157   // Armazena o resultado dos calculos para conferecencia
158   fr = fopen("result.txt","w");
159   if(fr == NULL){
160     perror("Could not open file");
161     return(0);
162   }
163   for(i=0;i<samples;i++){
164     fprintf(fr, "%.15f\t%.15f\n", alfa_vec[i], beta_vec[i]);
165   }
166   fclose(fr);
167
168   // Rotina de limpeza da memoria
```

```
169     close(s);
170     fclose(fp);
171     free(path_x);
172     free(path_y);
173     free(alfa_vec);
174     free(beta_vec);
175     free(timevec);
176     printf("OK\n");
177     return(0);
178 }
```

Apêndice F

Código fonte em C de base usado no hardware de teste

O código a seguir foi usado como base para todas as plataformas.

serv_max_calc.c

```c
/*
  Server for time evaluation
  Complete calculation
  by Nilson Pereira
*/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <sys/time.h>
#include "mylib.h"
#include "matlib.h"

// Define portas de comunicacao
#define BUFLEN 256
#define MYPORT 50102
#define REMOTEPORT 50101

// Funcao que executa a rotina de calculo
void stew(double *txvec, double *rxvec);

int main(void){
  int s, recv_len;
  double rxvec[13];
  double txvec[7];
  struct sockaddr_in si_me, si_other;
  socklen_t fromlen = sizeof(si_other);
  int slen = sizeof(si_other);

  // Prepara o socket para comunicacao UDP
  if ((s=socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP)) == -1)
  {
      die("socket");
  }

  memset((char *) &si_me, 0, sizeof(si_me));
  memset((char *) &si_other, 0, sizeof(si_other));

```

```
41    si_me.sin_family = AF_INET;
42    si_me.sin_port = htons(MYPORT);
43    si_me.sin_addr.s_addr = htonl(INADDR_ANY);
44
45    si_other.sin_family = AF_INET;
46    si_other.sin_port = htons(REMOTEPORT);
47
48    // Inicializa o socket
49    if( bind(s , (struct sockaddr*)&si_me, sizeof(si_me) ) == -1)
50    {
51        die("bind");
52    }
53
54    memset(txvec, 0, 7*sizeof(double));
55    printf("Waiting for connection...\n");
56
57    // loop infinito
58    while(1){
59        // Espera por dados do client (blocking call)
60        if ((recv_len = recvfrom(s, &rxvec, 13*sizeof(double), 0, (struct sockaddr *) &si_other, &fromlen)) == -1)
61        {
62            die("recvfrom()");
63        }
64
65        // Chamada principal da rotina de calculos
66        stew(txvec, rxvec);
67
68        // Envia dados de volta ao cliente
69        if (sendto(s, &txvec, 7*sizeof(double) , 0 , (struct sockaddr *) &si_other, slen)==-1)
70        {
71            die("sendto()");
72        }
73    }
74
75    close(s);
76    printf("OK\n");
77    return(0);
78 }
79
80 // Funcao que executa a rotina de calculo
81 void stew(double *txvec, double *rxvec){
82    double lambda;
83    double dE[6];
84    double Jm[12];
85    double dlsm[12];
86    double dang[6];
87
88    //Captura o tag recebido
89    txvec[0] = rxvec[0];
90
91    //Calcula a variacao de posicao de referencia baseado na cinematica direta
92    direct(rxvec, dE);
93
94    //Calcula as 3 matrizes Jacobianas para os 3 pontos de apoio
95    jacobian(rxvec, Jm);
96
97    //Encontra as matrizes DLS a partir das Jacobianas
98    lambda = 5;
99    dls(Jm, lambda, dlsm);
100
101   //Multiplacao de matrizes
102   mx_multiply(dlsm, dE, dang);
103
104   //Composicao do vetor de resposta
105   memcpy(&txvec[1], dang, 6*sizeof(double));
106 }
```

Apêndice G

Código fonte em C usado no Arduino

O código a seguir foi usado no Arduino para os testes de desempenho.

arduino.ino

```
/*
  Server for time evaluation
  Complete calculation for Arduino
  by Nilson Pereira
*/

#include <SPI.h>            // needed for Arduino versions later than 0018
#include <Ethernet.h>
#include <EthernetUdp.h>    // UDP library from: bjoern@cs.stanford.edu 12/30/2008
#include "mylib.h"
#include "matlib.h"

#define BUFFER_SIZE 256

// Configuracao dos parametros de rede
byte mac[] = {0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED};
IPAddress ip(192, 168, 2, 5);
IPAddress remoteip(192, 168, 2, 2);

unsigned int localPort = 50102;      // local port to listen on
unsigned int RemotePort = 50101;

// Buffers para trafego de informacao
char packetBuffer[BUFFER_SIZE]; //buffer to hold incoming packet,
byte txbuf[7*sizeof(double)];
float txvec[7], rxvec[13];

// Cria o objeto de comunicacao UDP
EthernetUDP Udp;

// Funcao que executa a rotina de calculo
void stew(float *txvec, float *rxvec);

// Inicializa socket de comunicao Serial e UDP
void setup() {
  Ethernet.begin(mac, ip);
  Udp.begin(localPort);

  Serial.begin(9600);
  Serial.println("Waiting for connection (max calc)...");
}

// Loop principal do pragrama
void loop() {
```

```
45
46    // Espera por dados recebidos via UDP
47    int packetSize = Udp.parsePacket();
48    if (packetSize){
49
50      Udp.read(packetBuffer, BUFFER_SIZE);
51      memcpy(&rxvec, packetBuffer, 13*sizeof(float));
52
53      // Chamada principal da rotina de calculos
54      stew(txvec, rxvec);
55
56      memcpy(&txbuf, txvec, 7*sizeof(float));
57
58      Udp.beginPacket(remoteip, RemotePort);
59      Udp.write(txbuf, 7*sizeof(float));
60      Udp.endPacket();
61
62      Udp.flush();
63
64    }
65    delay(10);
66  }
67
68  void stew(float *txvec, float *rxvec){
69    double lambda;
70    double dE[6];
71    double Jm[12];
72    double dlsm[12];
73    double dang[6];
74    unsigned int i;
75    double txvecd[7], rxvecd[13];
76
77    // Casting para double
78    for(i=0;i<13;i++){
79      rxvecd[i] = (double)rxvec[i];
80    }
81
82    //Captura o tag recebido
83    txvec[0] = rxvec[0];
84
85    //Calcula a variacao de posicao de referencia baseado na cinematica direta
86    direct(rxvec, dE);
87
88    //Calcula as 3 matrizes Jacobianas para os 3 pontos de apoio
89    jacobian(rxvec, Jm);
90
91    //Encontra as matrizes DLS a partir das Jacobianas
92    lambda = 5;
93    dls(Jm, lambda, dlsm);
94
95    //Multiplacao de matrizes
96    mx_multiply(dlsm, dE, dang);
97
98    // Casting para float
99    for(i=0;i<7;i++){
100     txvec[i+1] = (float)dang[i];
101   }
102 }
```

Apêndice H

Código fonte em C usado na Parallela e Epiphany

Os códigos a seguir foram usados na placa Parallella como algoritmo de cálculo em paralelo específico para o processador de múltiplos cores Epiphany.

A biblioteca "common.h"define as posições de memória estáticas em comum entre todos os cores da Epiphany para facilitar a transferência de dados entre eles. O código em "epiphany.c"é compilado para ser executado no processador principal da Parallella e é responsável pela comunicação e sincronização de dados entre o mundo externo e a Epiphany. O código em "e task.c"é executado nos diferentes cores da Epiphany.

`common.h`

```
/*
  Parallel Epiphany kernel
  Enderecos de memoria comuns entre os cores
  by Nilson Pereira
*/
#define D_RXVEC 0x7f00    // Vetor com dados de entrada
#define D_RESULT 0x7f10   // Vetor com resultados
#define D_FLAGS 0x7f20    // Vetor para sinalizadores
#define D_NEWTAG 0x7f30   // Tag sendo calculado
```

`epiphany.c`

```
/*
  Server for time evaluation
  Complete parallel calculation for epiphany
  by Nilson Pereira
*/
#include <stdlib.h>
#include <stdio.h>
#include <e-hal.h>
#include <unistd.h>
#include <string.h>
#include <arpa/inet.h>
#include "mylib.h"
#include "common.h"

// Define portas de comunicacao
```

```
16 #define BUFLEN 256
17 #define MYPORT 50102
18 #define REMOTEPORT 50101
19
20 int main(int argc, char *argv[]){
21   e_platform_t platform;
22   e_epiphany_t dev;
23
24   double rxvec[13], txvec[7];
25   double tag =  0.0;
26   int i,j;
27   double flag=0.0;
28   int s, recv_len;
29   struct sockaddr_in si_me, si_other;
30   socklen_t fromlen = sizeof(si_other);
31   int slen = sizeof(si_other);
32
33   // Prepara o socket para comunicacao UDP
34   if ((s=socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP)) == -1)
35   {
36       die("socket");
37   }
38
39   memset((char *) &si_me, 0, sizeof(si_me));
40   memset((char *) &si_other, 0, sizeof(si_other));
41
42   si_me.sin_family = AF_INET;
43   si_me.sin_port = htons(MYPORT);
44   si_me.sin_addr.s_addr = htonl(INADDR_ANY);
45
46   si_other.sin_family = AF_INET;
47   si_other.sin_port = htons(REMOTEPORT);
48
49   // Inicializa o socket
50   if( bind(s , (struct sockaddr*)&si_me, sizeof(si_me) ) == -1)
51   {
52       die("bind");
53   }
54
55   memset(txvec, 0, 7*sizeof(double));
56   printf("** Epiphany Parallel Server V1  **\n");
57
58   // Inicializa e prepara Epiphany
59   e_init(NULL);
60   e_reset_system();          // Reset Epiphany
61   e_get_platform_info(&platform);
62   e_open(&dev, 0, 0, 1, 4); // 1 row, 4 cols
63
64   // Transfere valores iniciais
65   rxvec[0] = 0;
66   e_write(&dev, 0, 0, D_RXVEC, rxvec, 1*sizeof(double));
67
68   // Inicializa rotina de calculo na Epiphany
69   e_load_group("e_task.srec", &dev, 0, 0, 1, 4, E_TRUE);
70
71   usleep(1000000);
72   printf("Waiting for connection ...\n");
73
74   // loop infinito
75   while(1){
76     // Espera por dados enviados pelo cliente
77     if ((recv_len = recvfrom(s, &rxvec, 13*sizeof(double), 0, (struct sockaddr *) &si_other, &fromlen)) == -1)
78     {
79         die("recvfrom()");
80     }
81
82     tag = rxvec[0];
83
84     // Transfere valores de calculo para memoria do core 0
```

```
85      e_write(&dev, 0, 0, D_RXVEC, &rxvec[1], 12*sizeof(double));
86      // Transfere a tag atual para memoria do core 0 que serve como trigger
87      e_write(&dev, 0, 0, D_RXVEC, &rxvec[0], 1*sizeof(double));
88
89      // Espera core 0 assinalar por fim do calculo
90      while(flag != tag){
91          e_read(&dev, 0, 0, D_RESULT, &flag, 1*sizeof(double));
92      }
93
94      // Trasnfere resultados para host
95      e_read(&dev, 0, 0, D_RESULT, &txvec, 7*sizeof(double));
96
97      // Envia dados de volta ao cliente
98      if (sendto(s, &txvec, 7*sizeof(double) , 0 , (struct sockaddr *) &si_other, slen)==-1)
99      {
100         die("sendto()");
101     }
102   }
103
104   // finaliza Epiphany
105   e_close(&dev);
106   e_finalize();
107   printf("Done!\n");
108 }
```

e_task.c

```
1  /*
2    Parallel Epiphany kernel
3    Complete parallel calculation for epiphany
4    by Nilson Pereira
5  */
6  #include <stdio.h>
7  #include <stdlib.h>
8  #include "e-lib.h"
9  #include "common.h"
10 #include "matgen/direct.c"
11
12 int main(void)
13 {
14   e_coreid_t coreid;
15   double *rxvec, *result, *flags, *newtag;
16   unsigned int i;
17   unsigned int row, col;
18   double mytag, mydang[2];
19   double dE[6];
20   double Jm[12];
21   double dlsm[12];
22
23   // Cria variaveis com enderecos de memoria estaticos
24   rxvec   = (double *) D_RXVEC;
25   result  = (double *) D_RESULT;
26   flags   = (double *) D_FLAGS;
27   newtag = (double *) D_NEWTAG;
28
29   coreid = e_get_coreid();
30   e_coords_from_coreid(coreid, &row, &col);
31
32   mytag = 0;
33   *newtag = 0;
34
35   // Inicializa sinalizadores
36   for(i=0;i<4;i++){
37     flags[i]=0;
38   }
39
40   if (col==0){
41     // *** Codigo somente para core 0 ***
42     while(1){
```

```
43        // Espera por novo tag para calculo
44        while(mytag == rxvec[0]){
45            // "Busy waiting"
46        }
47
48        // atualiza tag com nova tag
49        mytag = rxvec[0];
50
51        // Sinaliza os outros cores para calculo
52        for(i=1;i<4;i++){
53          e_write(&e_group_config, &mytag, 0, i, newtag, 1*sizeof(double));
54        }
55
56        // Loop esperando por sinalizadores dos cores
57        while( (mytag!=flags[1]) || (mytag!=flags[2]) || (mytag!=flags[3]) ){
58          // Busy waiting
59        }
60
61        // Sinaliza host sobre fim do calculo
62        result[0] = mytag;
63      }
64
65    }else{
66        // *** Codigo para os cores restantes ***
67        while(1){
68          // Espera por novo tag para calculo
69          while( mytag == *newtag ){
70            // Busy waiting
71          }
72          mytag = *newtag;
73
74          // Copia dados da memoria do core 0 para calculo
75          e_read(&e_group_config, &rxvec[0], 0, 0, &rxvec[(2*(col-1)+1)], 2*sizeof(double));
76          e_read(&e_group_config, &rxvec[2], 0, 0, &rxvec[(2*(col-1)+7)], 2*sizeof(double));
77
78          // Rotina de calculo
79          direct(rxvec, dE);
80          jacobian(rxvec, Jm);
81          dls(Jm, 5.0, dlsm);
82          mx_multiply(dlsm, dE, mydang);
83
84          // Cada core transfere o seu resultado no vetor de resposta no core 0
85          e_write(&e_group_config, mydang, 0, 0, &result[(2*(col-1)+1)], 2*sizeof(double));
86          e_write(&e_group_config, &mytag, 0, 0, &flags[col], 1*sizeof(double));
87        }
88    }
89  }
```

Apêndice I

Código fonte em C para CUDA usado na Jetson

O código a seguir foi feito usando as bibliotecas CUDA para ser executado na Jetson ou qualquer outra GPU da NVidia que possui suporte ao CUDA.

cuda.cu

```
/*
  Server for time evaluation
  Complete parallel calculation for Cuda
  by Nilson Pereira
*/
#include <stdio.h>
#include <math.h>
#include <unistd.h>
#include <Cuda.h>
#include <string.h>
#include <arpa/inet.h>
#include "mylib.h"
#include "matgen/funlib.c"

// Define portas de comunicacao
#define BUFLEN 256
#define MYPORT 50102
#define REMOTEPORT 50101

 // Kernel para ser executado na GPU
__global__ void  cuda_kernel(float *rxvec, float *txvec){
   int tid = threadIdx.x;
  float myrxvec[4];
  float mytxvec[2];
  int i;
  float dE[2];
  float Jm[4];
  float dlsm[4];

   // Transfere os dados da memoria global para variaveis locais
  for(i=0;i<2;i++){
    myrxvec[i] = rxvec[(2*tid)+1+i];
    myrxvec[i+2] = rxvec[(2*tid)+7+i];
  }

   // Rotina de calculo
   direct(myrxvec, dE);
   jacobian(myrxvec, Jm);
```

```
39     dls(Jm, 5.0, dlsm);
40     mx_multiply(dlsm, dE, mytxvec);
41
42     // Transfere os resulados para a memoria global
43   txvec[(2*tid)+1] = mytxvec[0];
44   txvec[(2*tid)+2] = mytxvec[1];
45 }
46
47 // Programa principal para ser executado no host
48 int main(void){
49   float h_rxvec[13];
50   float h_txvec[7];
51   float *d_rxvec, *d_txvec;
52   int s, recv_len;
53   struct sockaddr_in si_me, si_other;
54   socklen_t fromlen = sizeof(si_other);
55   int slen = sizeof(si_other);
56    cudaEvent_t start, stop;
57    cudaEventCreate(&start);
58    cudaEventCreate(&stop);
59    float time=0;
60
61    // Prepara o socket para comunicacao UDP
62   if ((s=socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP)) == -1){
63     die("socket");
64   }
65
66   memset((char *) &si_me, 0, sizeof(si_me));
67   memset((char *) &si_other, 0, sizeof(si_other));
68    memset(h_txvec, 0, 7*sizeof(float));
69
70   si_me.sin_family = AF_INET;
71   si_me.sin_port = htons(MYPORT);
72   si_me.sin_addr.s_addr = htonl(INADDR_ANY);
73
74   si_other.sin_family = AF_INET;
75   si_other.sin_port = htons(REMOTEPORT);
76
77   // Inicializa o socket
78   if( bind(s , (struct sockaddr*)&si_me, sizeof(si_me) ) == -1)
79   {
80       die("bind");
81   }
82
83    // Cria dois vetores na memoria global da gpu
84   cudaMalloc((void**)&d_rxvec,13*sizeof(float));
85   cudaMalloc((void**)&d_txvec,7*sizeof(float));
86
87   // Configura o numero e organizacao dos cores em cada bloco e a quantidade de blocos
88    // bloco (x,y,z) e grid (x,y,z)
89    dim3 dimBlock(3,1,1);
90    dim3 dimGrid(1,1,1);
91
92   printf("Waiting for connection...\n");
93
94    // Loop infinito
95   while(1){
96     // Espera por dados enviados pelo cliente
97     if ((recv_len = recvfrom(s, &h_rxvec, 13*sizeof(float), 0, (struct sockaddr *) &si_other, &fromlen)) == -1){
98         die("recvfrom()");
99     }
100
101       // Transfere os dados para a memoria global da GPU
102    cudaMemcpy(d_rxvec,h_rxvec,(13*sizeof(float)),cudaMemcpyHostToDevice);
103
104       // executa o kernel na GPU
105     cuda_kernel<<<dimGrid,dimBlock>>>(d_rxvec, d_txvec);
106
107       // Espera ate que todos os cores tenham terminado de executar o kernel
```

```
108     cudaDeviceSynchronize();
109
110       // Recolhe os resultados da memoria global da GPU
111     cudaMemcpy(h_txvec,d_txvec,(7*sizeof(float)),cudaMemcpyDeviceToHost);
112
113       // Atualiza tag do resultado
114       h_txvec[0] = h_rxvec[0];
115
116     // Envia dados de volta ao cliente
117     if (sendto(s, &h_txvec, 7*sizeof(float) , 0 , (struct sockaddr *) &si_other, slen)==-1){
118         die("sendto()");
119     }
120   }
121
122    // Rotina de finalizacao e limpeza da memoria
123   cudaFree(d_rxvec);
124    cudaFree(d_txvec);
125   close(s);
126   printf("Done.\n");
127   return 0;
128 }
```

Bibliografia

[1] Adapteva. Epiphany sdk (esdk). http://www.adapteva.com/epiphany-sdk/. [Online; acessado Out-2015].

[2] Arduino. Arduino - sdk reference. https://www.arduino.cc/en/Reference/HomePage. [Online; acessado Out-2015].

[3] Arduino. Sparkfun - loja online - arduino mega 2560 r3. https://www.sparkfun.com/products/11061. [Online; acessado Set-2015].

[4] Buss, S. R. Introduction to inverse kinematics with jacobian transpose, pseudoinverse and damped least squares methods. http://www.math.ucsd.edu/~sbuss/ResearchWeb/ikmethods/iksurvey.pdf, October 2009. University of California, San Diego.

[5] Geraldes, R. Desenvolvimento do protótipo de um sistema hexapode para posicionamento e alinhamento de câmara de espelhos. Tech. rep., Laboratório Nacional de Luz Síncrotron, 2010.

[6] Jetson. Site de reviews - nvidia jetson tk1 cuda performance. https://www.pugetsystems.com/labs/articles/NVIDIA-Jetson-TK1-CUDA-performance-569/. [Online; acessado Set-2015].

[7] LNLS. Projetos desenvolvidos pelo grupo de desenho mecânico. http://lnls.cnpem.br/engineering/pro/projects/. [Online; acessado Set-2015].

[8] NVidia. Cuda sdk reference. https://docs.nvidia.com/cuda/. [Online; acessado Out-2015].

[9] Pacheco, P. S. *An introduction to parallel programming*, 1st ed. Morgan Kaufmann, 2011.

[10] Pal, B., and Magadum, S. Trajectory tracking of a 3-dof articulated arm by jacobian solutions. *International Journal of Advanced Engineering Applications 5* (2012), pp.14–21.

[11] Parallella. Loja online do fabricante - adapteva parallella-16 desktop computer. http://adapteva.myshopify.com/collections/featured-products/products/parallella-16-desktop-computer. [Online; acessado Set-2015].

[12] Raspberry, P. Website do fabricante - raspberry pi modelo b. https://www.raspberrypi.org/blog/made-in-the-uk/. [Online; acessado Set-2015].

[13] Spong, M. W., Hutchinson, S., and Vidyasagar, M. *Robot modeling and control*, 1st ed. John Wiley & Sons, Hoboken (N.J.), 2006.