UNIVERSIDADE FEDERAL DE SANTA CATARINA

CURSO DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

CONTRIBUIÇÃO AO ESTUDO, ANÁLISE E PROJETO DE ROBÔS MANIPULADORES: UM SOFTWARE SIMULADOR

Dissertação submetida a
Universidade Federal de Santa Catarina
como parte dos requisitos para a
obtenção do grau de Mestre em Engenharia Elétrica.

CARLOS RODRIGUES ROCHA

Florianópolis, Abril de 2001.

CONTRIBUIÇÃO AO ESTUDO, ANÁLISE E PROJETO DE ROBÔS MANIPULADORES: UM SOFTWARE SIMULADOR

CARLOS RODRIGUES ROCHA

Esta Dissertação foi julgada adequada para obtenção do Título de Mestre em Engenharia Elétrica, Área de Concentração Controle, Automação e Informática Industrial, e aprovada em sua forma final pelo Curso de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Santa Catarina.

Prof. Raul Guenther, D.Sc.
Orientador

Prof. Edson Roberto de Pieri, Dr.
Co-Orientador

Prof. Edson Roberto de Pieri, Dr.
Coordenador do Curso de Pós-Graduação em Engenharia Elétrica

Banca Examinadora:

Prof. Marcelo Ricardo Stemmer, Dr.

Prof. Altamir Dias, D.Sc.

À minha esposa, minha filha e meus pais

AGRADECIMENTOS

À minha esposa, Patrícia Anselmo Zanotta, pelo seu apoio, compreensão e paciência.

Aos meus pais, pelo zelo e apoio.

Aos professores do Curso de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Santa Catarina, pelo conhecimento e experiência transmitidos, e em particular aos meus orientadores, Raul Guenther e Edson De Pieri.

Ao professor Sebastião Cícero Pinheiro Gomes, que me incentivou a seguir na área de robótica, desde a graduação, e cujo projeto por ele orientado serviu como base para este trabalho.

Ao M. Eng. Henrique Simas, pelas sugestões, críticas e apoio em vários estágios do meu trabalho.

Aos colegas e amigos professores do Colégio Técnico Industrial Prof. Mário Alquati, da Fundação Universidade Federal do Rio Grande, pelo incentivo e apoio constantes.

A todos aqueles, que direta ou indiretamente acompanharam ou contribuíram para a elaboração deste trabalho.

Resumo da Dissertação apresentada à UFSC como parte dos requisitos necessários para a obtenção do grau de Mestre em Engenharia Elétrica.

CONTRIBUIÇÃO AO ESTUDO, ANÁLISE E PROJETO DE ROBÔS MANIPULADORES: UM SOFTWARE SIMULADOR

Carlos Rodrigues Rocha

Abril/2001

Orientadores: Raul Guenther, Dr., Edson Roberto De Pieri, Dr.
Área de Concentração: Controle, Automação e Informática Industrial.
Palavras-chave: Robótica, robôs manipuladores, simulação, orientação a objetos.
Número de Páginas: 179.

O presente trabalho aborda o desenvolvimento de uma ferramenta computacional para a simulação de robôs manipuladores, com a finalidade de auxiliar no estudo, análise e projeto destes equipamentos. Utilizando o paradigma da orientação a objetos, o *software* é desenvolvido de forma totalmente modular. Procurando estabelecer uma metodologia de trabalho ao usuário, faz-se uma revisão das características cinemáticas e dinâmicas dos manipuladores, também necessária para a modelagem e implementação destes. Os elementos da simulação são configuráveis, permitindo ao usuário definir o tipo de manipulador e seus parâmetros, o gerador de trajetórias empregado no movimento deste ao longo da tarefa e o controlador do manipulador e seus parâmetros. A fim de permitir o estudo de algoritmos de controle, é possível ao usuário implementá-los em módulos externos ao simulador, que são vinculados a ele durante a execução de tarefas. A análise dos resultados das simulações é feita em gráficos de variáveis de estado e em animações de modelos dos manipuladores em um ambiente virtual 3D, utilizando componentes de *software* criados para este fim. São apresentados estudos de caso de modelagem de manipuladores e de tarefas, para ilustrar o funcionamento e utilização do simulador. Além da contribuição do *software* de simulação, salienta-se uma contribuição secundária caracterizada pelas bibliotecas de código criadas para este trabalho, que podem ser reutilizadas no desenvolvimento de outras aplicações.

Abstract of Dissertation presented to UFSC as a partial fulfillment of the requirements for the degree of Master in Electrical Engineering.

CONTRIBUTION TO THE STUDY, THE ANALYSIS AND THE PROJECT OF ROBOT MANIPULATORS: A SIMULATION SOFTWARE

Carlos Rodrigues Rocha

April/2001

Advisors: Raul Guenther, Edson Roberto De Pieri.
Area of Concentration: Controle, Automação e Informática Industrial.
Keywords: Robotics, robot manipulators, simulation, object-oriented paradigm.
Number of Pages: 179.

This work tackles the development of a computational tool to simulate robot manipulators, with the purpose of aiding in the study, analysis and project of these equipments. The software is developed in a modular way, by use of the object-oriented paradigm. A revision of the kinematics and dynamics of manipulators is made, in order to establish a work methodology to the user and as a basis to their modeling and implementation. The elements of the simulation are configurable, allowing to the user to define the type of the manipulator and its parameters, the trajectory generator employed in the movement of the manipulator along the task and the manipulator's controller and its parameters. It is possible to the user to implement control algorithms in modules which are external to the simulator. The modules are linked to the simulator during the execution of tasks, allowing the study of these algorithms. The simulation results are analyzed in graphs of the state variables and in animations of the manipulators' model in a virtual environment, using specific software components created for the simulator. Case studies of manipulators' modeling and of task simulation are presented, to illustrate the operation and use of the simulator. Besides the contribution of the simulation software, it is pointed out a secondary contribution characterized by the code libraries created for this work, that can be reused in the development of another applications.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

1. INTRODUÇÃO

É inegável a importância da automação nas mais diversas atividades. A robótica, em especial, é utilizada em um número cada vez maior de aplicações, trazendo como vantagens redução de custos, aumento de precisão, produtividade, flexibilidade e a possibilidade de executar tarefas difíceis ou mesmo impossíveis para seres humanos[1].

Como aplicações típicas na indústria, estão a manipulação de objetos (movimentação de materiais e peças, carga e descarga em equipamentos, classificação e empacotamento), a manufatura (soldagem a arco e a ponto, pintura spray, usinagem, corte com laser e água, montagem e acabamento) e a medição (inspeção de objetos, seguimentos de contornos e detecção de falhas)[2,3].

Na medicina, a robótica está sendo utilizada na análise e diagnóstico de doenças, em operações delicadas e remotas. Na área aeroespacial, robôs são empregados na manutenção de satélites, na construção de estações espaciais e na exploração planetária. Outros exemplos incluem atividades de mineração, navegação e controle de veículos automotivos[4].

O grande número de aplicações da robótica implica em variedade de modelos de robôs. O projeto destes e o planejamento das tarefas a serem executadas é essencial para o emprego correto desta tecnologia. A simulação computacional é uma ferramenta de apoio importante para estas atividades, permitindo testar diferentes configurações de robôs e de seus componentes antes da construção de protótipos ou maquetes, além de possibilitar a viabilidade da execução de uma tarefa sem arriscar um equipamento valioso.

Neste capítulo é feita uma introdução à robótica e suas aplicações, iniciando por uma revisão histórica, seguida de uma análise da relação entre robótica e automação. As

características dos robôs industriais e seus componentes são mostradas após, além de se avaliar a importância da simulação como ferramenta de apoio à análise e projeto. Por fim, estabelecem-se os objetivos deste trabalho e apresenta-se a estrutura desta dissertação.

1.1 UMA VISÃO HISTÓRICA

Embora o termo robô seja relativamente novo, a busca da criação de entidades artificiais para imitar o comportamento dos seres humanos e sua interação com o meio é muito antiga. A idéia de dar vida a artefatos aparece em muitos mitos da antigüidade, como o de Prometeu, que moldou a humanidade a partir do barro; as donzelas de ouro criadas por Hefestos, o deus grego das forjas, que agiam, pensavam e falavam como seres humanos, presentes na Ilíada de Homero; o gigante Talos, protetor da ilha de Creta; e o Golem, ser criado do barro pelo rabino Loew com a finalidade de proteger o povo judeu de seus perseguidores[2,5].

Na era industrial, os mitos deram lugar às invenções, sendo desenvolvidos vários autômatos que podiam tocar instrumentos musicais, pintar, escrever cartas ou imitar animais. Na busca do aumento da produtividade, surgiram inventos como a fiandeira de Hargreaves, a máquina de fiar de Crompton e o tear de Jacquard[3]. A síntese das idéias da época sobre a criação de seres artificiais está em Frankenstein, escrito em 1817 por Mary Shelley[2,5].

O termo robô foi extraído da peça Os Robôs Universais de Rossum, escrita pelo escritor tcheco Karel Capek em 1921. Nela, o cientista Rossum cria autômatos para realizar serviços físicos para a humanidade. Os robôs acabam por se insurgir contra a escravidão (o termo *robota* significa trabalhador forçado ou escravo no idioma eslavo), exterminando os humanos[1,2,3,5].

Na década de 40, Isaac Asimov concebeu os robôs como autômatos humanóides, cujo comportamento seria programado por humanos em seus cérebros positrônicos visando satisfazer algumas regras de conduta ética. Como máquinas, seriam projetadas por engenheiros ou técnicos especializados, envolvidos com uma ciência denominada por ele de *Robótica*, que se basearia em três leis fundamentais:

1. Um robô não pode fazer mal a um ser humano ou, por omissão, permitir que um ser humano sofra algum tipo de mal;
2. Um robô deve obedecer às ordens dos seres humanos, a não ser que conflitem com a Primeira Lei;
3. Um robô deve proteger sua própria existência, a não ser que tal proteção entre em conflito com a Primeira ou a Segunda Leis.

A ficção científica moderna ainda mostra os robôs como criaturas independentes e completamente capazes de tomar decisões, na maioria das vezes com forma humanóide e, em alguns casos, tomados por emoções. Citam-se como exemplos os robôs dos filmes Guerra nas Estrelas, Exterminador do Futuro e o andróide Data, da série Jornada nas Estrelas - A Nova Geração. Apesar de popularizarem a robótica, tais descrições ainda estão longe da realidade atual.

O desenvolvimento da robótica nas industrias deve-se à junção das tecnologias da *teleoperação* e do *controle numérico*[1]. A primeira foi desenvolvida durante a Segunda Guerra Mundial para a manipulação à distância de materiais radioativos, enquanto a segunda foi criada para atender os requisitos de alta precisão nos processos de usinagem de componentes usados em áreas como a aviação[1,3,6].

1.2 Robótica e Automação

O termo automação surgiu nos anos 40, na Ford Motor Company, para descrever a operação coletiva de várias máquinas interconectadas[1]. Mais recentemente, considera-se a automação como a tecnologia cujo objetivo é substituir seres humanos por máquinas em processos produtivos, não somente na execução física dos mesmos, mas também no processamento inteligente da informação inerente ao processo[2]. Por esse motivo, a automação envolve diversas áreas do conhecimento, como mecânica, eletrônica, informática e planejamento da produção.

[1] É bastante aceito como primeiro trabalho dessa fusão de tecnologias o dispositivo programável para manipulação de materiais desenvolvido e patenteado por George Devol em 1954. Este dispositivo é considerado a origem dos manipuladores industriais[3, 6].

Na indústria, a automação costuma ser classificada em fixa, programável e flexível[3,4]. A automação *fixa* está associada a grandes volumes de produção, onde os equipamentos são projetados especificamente para um tipo de processo. Na automação *programável*, por outro lado, os equipamentos são mais versáteis, possibilitando a variação no processo produtivo mediante reprogramação, sendo empregados para a produção de pequeno a médio volume.

A automação *flexível* é considerada uma evolução da automação programável, permitindo a fabricação de um volume variável de produtos diferentes, procurando minimizar o tempo gasto na reprogramação[2]. Em geral, os sistemas flexíveis de manufatura, ou de manufatura integrada por computador[2], consistem em estações de trabalho conectadas por um sistema de manuseio e armazenamento de materiais, o que torna essa classe de automação uma intermediária entre a automação fixa e a automação programável[3].

A robótica é considerada como um tipo de automação programável, devido a versatilidade e flexibilidade dos robôs empregados na indústria. A definição de robô industrial do Robot Institute of America(RIA) reforça essa afirmação: "*Um robô é um manipulador multifuncional reprogramável, projetado para mover materiais, peças, ferramentas ou dispositivos especializados através de movimentos variáveis programados para a realização de uma variedade de tarefas*"[1,2,3]. Apesar dessa classificação, os robôs industriais também podem ser encontrados em sistemas flexíveis (como sistemas de manuseio de materiais, por exemplo) e fixos.

1.3 CARACTERIZAÇÃO DE ROBÔS INDUSTRIAIS

Sob muitos aspectos, qualquer equipamento que opera com algum grau de autonomia, usualmente controlado por computador, pode ser chamado de robô[1]. Para tanto, considera-se que um robô é geralmente composto por um sistema de locomoção, para se movimentar pelo ambiente, e um manipulador, responsável pela execução de

[2] Em inglês, Computer Integrated Manufacturing (CIM).

tarefas[2]. Os dois sistemas costumam ser estudados separadamente, sendo denominados de *robôs móveis* e *robôs manipuladores*[7].

Os aspectos inerentes à autonomia são fundamentais para a execução de tarefas em ambientes pouco estruturados, insalubres ou perigosos. O termo *robótica avançada* começou a ser utilizado na década de 80, para designar a reunião de áreas como ciência da cognição, inteligência artificial, computação de tempo real, teoria de controle e teoria da informação, além da mecânica e da eletrônica, com a finalidade de possibilitar tal autonomia através do uso de vários tipos de sensores e explorando o poder computacional dos equipamentos de custo relativamente baixo disponíveis no mercado[4]. Esta área ainda apresenta muitos desafios, estando os trabalhos na fase de protótipos ou em poucas aplicações práticas.

Por outro lado, a *robótica industrial*, que corresponde ao projeto, controle e aplicações na indústria está alcançando um nível de maturidade tecnológica, em parte pelo fato do ambiente industrial costumar ser fortemente estruturado, sendo o grau de autonomia requerido para as aplicações bastante reduzido[2].

O objeto de estudo deste trabalho é o robô industrial, conforme a definição da RIA, que consiste em um manipulador. Este é constituído por uma estrutura mecânica (o manipulador propriamente dito), atuadores (que movimentam o manipulador), sensores[3] (que fornecem informações sobre o robô e o ambiente em que este se encontra) e um sistema de controle (que possibilita o controle e supervisão do movimento do manipulador).

Um manipulador pode ser visto como uma cadeia cinemática aberta, formada por elos conectados por juntas móveis[1,6,7]. As juntas costumam prover apenas um grau de mobilidade, que pode ser translacional ou rotacional.

O arranjo das juntas na cadeia cinemática define o grau de mobilidade do manipulador, devendo ser cuidadosamente planejado para fornecer ao efetuador final os graus de liberdade necessários à execução das tarefas. Este corresponde à terminação da cadeia cinemática. No caso mais geral, o número máximo de graus de liberdade é igual a

[3] Neste trabalho, consideram-se como sensores apenas os internos, que disponibilizam informações de posição, velocidade, torque e aceleração das juntas.

seis, correspondendo a três componentes de posição e três componentes que definem a orientação. Eventualmente, o manipulador pode ter mais graus de liberdade que os necessários para a execução das tarefas, sendo nesse caso considerado redundante[2].

São cinco os tipos de manipuladores industriais mais comuns, definidos em função de sua estrutura: cartesiano, cilíndrico, esférico, articulado e SCARA (Selective Compliant Articulated Robot for Assembly). A Figura 1.1 ilustra esses arranjos e seus volumes de trabalho característicos.

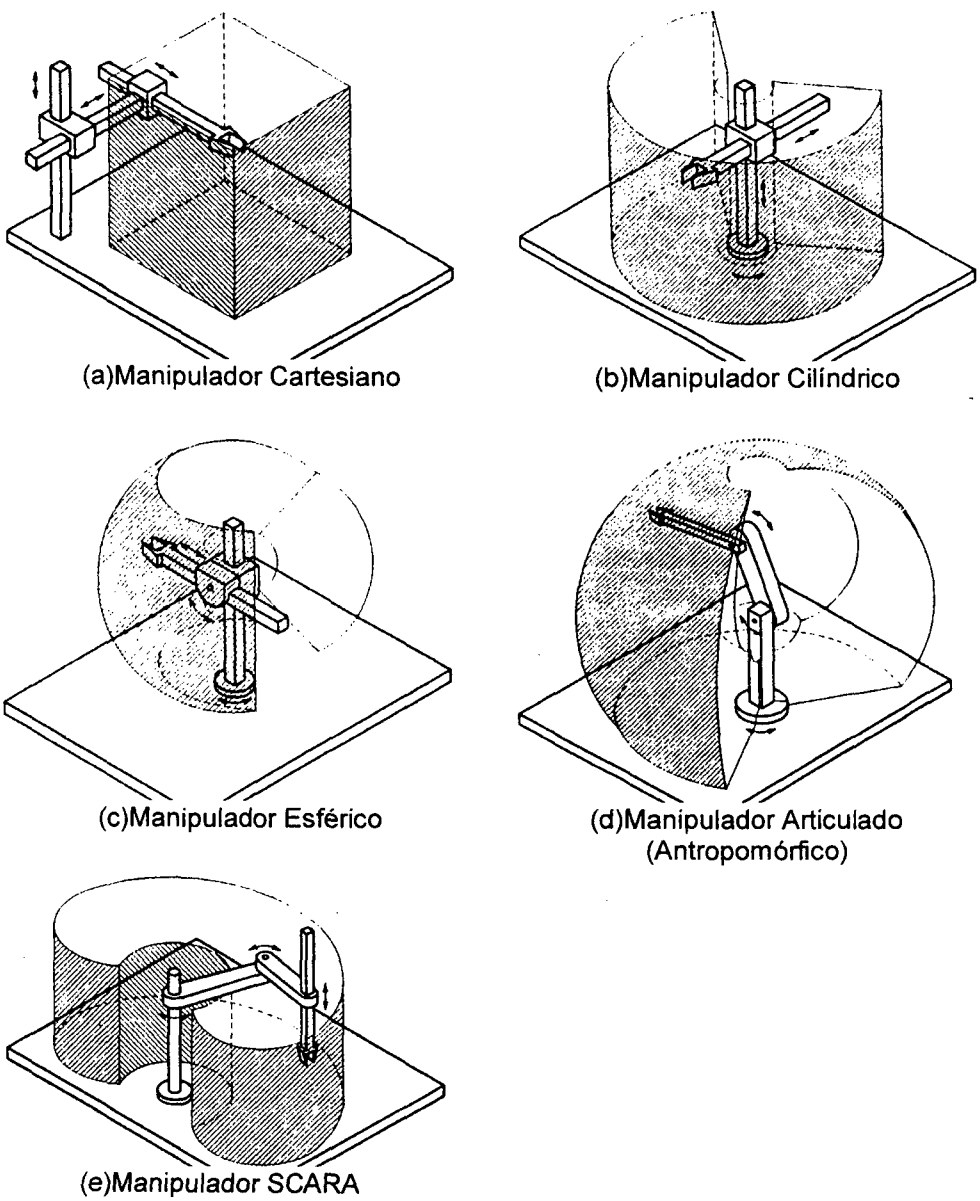


Figura 1.1 Estruturas cinemáticas comuns nos robôs manipuladores e seus espaços de trabalho

A modelagem dos manipuladores é importante para o projeto mecânico das suas estruturas, escolha dos atuadores, determinação de estratégias de controle e simulação. Para tanto, é fundamental o conhecimento da cinemática (direta, inversa e diferencial) e da dinâmica destes. O problema de controle de posição e força também deve ser estudado, para garantir a performance do sistema na execução das tarefas[1,2].

1.4 A SIMULAÇÃO COMO FERRAMENTA

Simulação é definida como a representação e avaliação de um sistema através de modelos que contenham as características relevantes para o entendimento do seu comportamento[8]. A simulação é uma interessante aplicação dos computadores no estudo e análise de projetos em engenharia. A partir de modelos matemáticos, é possível representar objetos do mundo real e seu comportamento na interação destes com o ambiente onde estão inseridos[9].

A vantagem da simulação computacional, em relação aos métodos mais convencionais, está na rapidez com que os resultados são obtidos e analisados, pois um grande número de possíveis soluções pode ser avaliado em um menor espaço de tempo, reduzindo o uso de maquetes e protótipos apenas às soluções mais viáveis, fator que implica em economia.

Diversas tecnologias são empregadas em simulação, como a computação gráfica, realidade virtual e matemática computacional, particularmente os métodos numéricos para solução de sistemas. Considera-se a simulação uma área propicia à abordagem de desenvolvimento orientado a objetos, pois nestes a modelagem pode ser feita com uma maior correspondência com o mundo real[10].

Para uma simulação realista, os modelos matemáticos devem representar as entidades do mundo real de forma precisa e exata, procurando porém manter a menor complexidade possível, por questões de performance. Devem também ser de fácil verificação e manipulação, além de identificação fácil da correspondência entre os termos do modelo e os fenômenos representados. São requisitos desejados a facilidade de interação com o usuário e de análise de resultados, em particular utilizando gráficos e

animações[8,9]. Na simulação de robôs manipuladores, os modelos correspondem a sistemas de equações diferenciais fortemente acopladas, que requerem métodos eficientes para a sua solução.

1.5 UM SIMULADOR DE ROBÔS MANIPULADORES

Um simulador de robôs manipuladores pode consistir em um *software* cuja finalidade é modelar as características cinemáticas e dinâmicas dos manipuladores, possibilitando analisar seu funcionamento na simulação de execução de tarefas.

Além das características inerentes a qualquer simulação computacional, destaca-se a importância de uma ferramenta desta natureza como auxiliar na análise e estudo de manipuladores, pois a disponibilidade destes equipamentos, especialmente os de porte industrial, costuma ser limitada por diversos fatores, particularmente no meio acadêmico. Além disso, a simulação computacional também é uma excelente maneira de obter conhecimento e experiência em projeto e análise de controladores para manipuladores. Sendo digitais a maioria dos sistemas atuais, conceitualmente há pouca variação entre a simulação e sua efetiva implementação[11].

Um requisito desejável para este tipo de *software* é a flexibilidade na definição das características dos manipuladores, possibilitando a análise de diferentes tipos de robôs com a mesma ferramenta. Tal flexibilidade não se refere apenas à mudanças paramétricas, mas também na estrutura dos manipuladores. Para satisfazer tal requisito, deve-se desenvolver uma metodologia geral de modelagem, que possa ser implementada no *software* e adotada pelo usuário. Para tanto, é necessária a compreensão das características *de modelagem cinemática* e *modelagem dinâmica* dos manipuladores.

O trabalho ora apresentado é uma derivação de um projeto iniciado no Departamento de Matemática da Fundação Universidade Federal do Rio Grande, orientado pelo Prof. Dr. Sebastião Cícero Pereira Gomes, com a finalidade de desenvolver um simulador de robôs manipuladores voltado inicialmente para o ambiente acadêmico[9]. Salienta-se, porém, que apenas os conceitos e experiência do projeto original foram

aproveitados neste trabalho, sendo feito todo um novo desenvolvimento de *software*, visando melhorar características de performance, flexibilidade e reusabilidade de código.

Em relação ao projeto original, o *software* aqui apresentado tem como aprimoramentos a *generalização da modelagem* de manipuladores, possibilitando simular diferentes tipos de robôs apenas pela modificação das características cinemáticas e dinâmicas dos mesmos na interface de uso ou em um arquivo de definição de robôs, a possibilidade de *testar controladores implementados pelo usuário* em módulos que podem ser acoplados ao software em tempo de execução e *recursos gráficos* 3D mais realistas e eficientes, graças ao uso de uma biblioteca gráfica específica para esse fim.

1.6 OBJETIVO DA DISSERTAÇÃO

O objetivo principal deste trabalho é o desenvolvimento de um simulador de robôs manipuladores, realizado a partir de uma análise dos aspectos inerentes à modelagem de manipuladores e dos demais elementos necessários para a execução de uma tarefa, como controladores e geradores de trajetórias.

Para a visualização dos resultados da simulação, é criado um ambiente virtual que possibilita ver um modelo do manipulador estudado executando uma tarefa, mostrada através de câmeras sintéticas, que se movimentam por esse ambiente. Também é criado um componente para traçado de gráficos, utilizados para visualização do comportamento das variáveis envolvidas na simulação.

A principal contribuição deste trabalho é o *software* simulador de robôs manipuladores, que serve como ferramenta de apoio à análise e projeto tanto dos manipuladores quanto de controladores, particularmente para estudantes de robótica. Outra contribuição importante é a criação de bibliotecas e componentes que podem ser reutilizados em outras aplicações, como matrizes numéricas e suas operações, métodos de solução de equações diferenciais, modelagem e visualização 3D e modelagem de manipuladores.

1.7 ESTRUTURA DA DISSERTAÇÃO

No capítulo 2 são analisados os aspectos da modelagem cinemática de um robô manipulador, que corresponde à descrição do seu movimento. Apresenta-se uma metodologia para a obtenção da cinemática direta e o problema da inversão cinemática, além de conceitos como espaço operacional, redundância e a necessidade de uma representação mínima para a orientação do efetuador final.

No capítulo 3 estuda-se a modelagem dinâmica do manipulador, que relaciona o movimento do manipulador com as forças e torques aplicados nos seus atuadores e no efetuador final. São mostradas metodologias para a obtenção da dinâmica de um manipulador, e faz-se uma análise dos aspectos computacionais de sua implementação, particularmente para a simulação do comportamento de robôs manipuladores.

O capítulo 4 apresenta o desenvolvimento do simulador proposto neste trabalho, sendo definidos os requisitos do projeto e as ferramentas para a implementação do simulador. Faz-se uma análise do desenvolvimento baseado em objetos, paradigma utilizado para o desenvolvimento do *software*, onde este é visto como uma junção de módulos desenvolvidos separadamente, que podem ser reutilizados em outras aplicações. Estes módulos também são analisados neste capítulo.

No capítulo 5 são mostrados modelos de manipuladores implementados. Alguns modelos são apenas cinemáticos, para demonstrar como estes são definidos e para a análise do movimento de estruturas comuns utilizadas nos manipuladores. Também são apresentados modelos dinâmicos, cujos parâmetros e geometria foram retirados da literatura e de manipuladores reais. Para estes, são simuladas tarefas para demonstração da efetividade da simulação.

O capítulo 6 é dedicado às conclusões gerais e perspectivas de trabalhos futuros.

Apresentam-se, nos anexos, guias de utilização das bibliotecas e componentes relevantes desenvolvidos para este simulador, as descrições dos manipuladores estudados neste trabalho implementados segundo o formato texto utilizado pelo simulador e exemplos de implementação de controladores para uso em simulação dinâmica.

2. MODELAGEM CINEMÁTICA

Com a finalidade de identificar as características necessárias à modelagem de robôs manipuladores, neste capítulo é feita uma revisão da cinemática dos manipuladores, resumindo o que se apresenta em textos clássicos sobre o assunto, como ASADA e SLOTINE[6], SPONG e VIDYASAGAR[1] e SCIAVICCO e SICILIANO[2], entre outros.

À modelagem cinemática de um manipulador concerne a descrição do movimento deste em relação a um sistema de coordenadas referencial fixo, ignorando-se as forças e momentos que fazem com que o manipulador se mova[7]. Este modelo compreende tanto a descrição do movimento do efetuador final em função dos movimentos das juntas, conhecida como *cinemática direta*, como a correspondência entre o movimento do efetuador final e os movimentos das juntas, denominada *cinemática inversa*.

É comum, no estudo da cinemática do manipulador, fazer distinção entre o estudo das *relações entre posições* e o estudo das *relações entre velocidades e acelerações*. O primeiro caso é conhecido como *cinemática* propriamente dita, enquanto o segundo é referido como *cinemática diferencial*. Neste capítulo é apresentada primeiramente a cinemática direta, seguida da cinemática inversa e da cinemática diferencial.

2.1 CINEMÁTICA DIRETA

A cinemática direta corresponde à determinação da posição e orientação do efetuador final em função da configuração do manipulador, ou seja, da geometria de seus elos, dos tipos de juntas que os vinculam e das variáveis das juntas. Para análise cinemática, considera-se o manipulador como uma *cadeia cinemática* formada por uma

série de elos conectados por juntas de translação ou de rotação[4]. Na maioria dos robôs industriais essas cadeias são *abertas*, onde uma extremidade corresponde à base e a outra ao efetuador final, como ilustra a Figura 2.1[6].

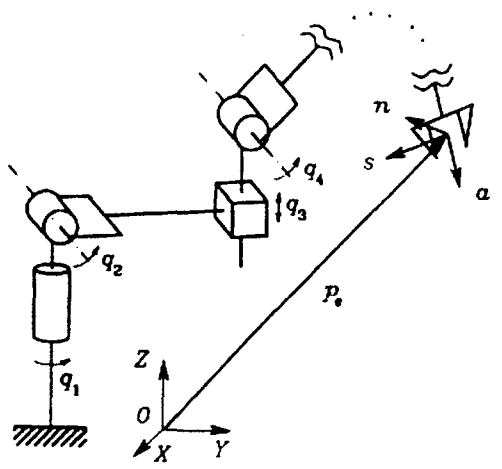


Figura 2.1 Manipulador visto como uma cadeia cinemática aberta

A partir das considerações acima, pode-se definir um método para a determinação da cinemática direta. Inicialmente, *sistemas de coordenadas* são fixados aos elos, representando suas posições[5] relativas a uma referência comum. A seguir, descreve-se a estrutura do manipulador por um conjunto mínimo de parâmetros geométricos, através da *notação Denavit-Hartenberg*. Por fim, relacionam-se os sistemas de coordenadas dos elos através de *transformações de coordenadas*, representadas por *matrizes de transformações homogêneas*. A *equação cinemática direta* resultante consiste em uma matriz que representa a posição do efetuador final em função das variáveis das juntas. Eventualmente, este resultado ainda precisará ser transformado, para que a orientação seja representada de forma *mínima*.

[4] Segundo DENAVIT e HARTENBERG[12], esta análise é válida para qualquer *vínculo inferior*, onde o contato é feito entre duas superfícies. Nesse caso, os vínculos com mais de um grau de liberdade (*par esférico*, *par cilíndrico*, *par plano*, *parafuso*) podem ser analisados através de sua decomposição em vínculos translacionais e/ou rotacionais.

[5] Deste ponto em diante, posição corresponderá à posição e orientação, exceto nos casos em que a diferenciação seja clara.

2.1.1 POSIÇÃO E ORIENTAÇÃO DE UM ELO

Nesta análise cinemática, os elos são considerados corpos rígidos, cuja posição e orientação no espaço é descrita em relação a um sistema de coordenadas de referência (v. Figura 2.2). A *posição* pode ser especificada com relação a um ponto qualquer do corpo rígido, enquanto a *orientação* pode ser definida por um sistema de coordenadas ortonormal vinculado ao ponto fixo[2].

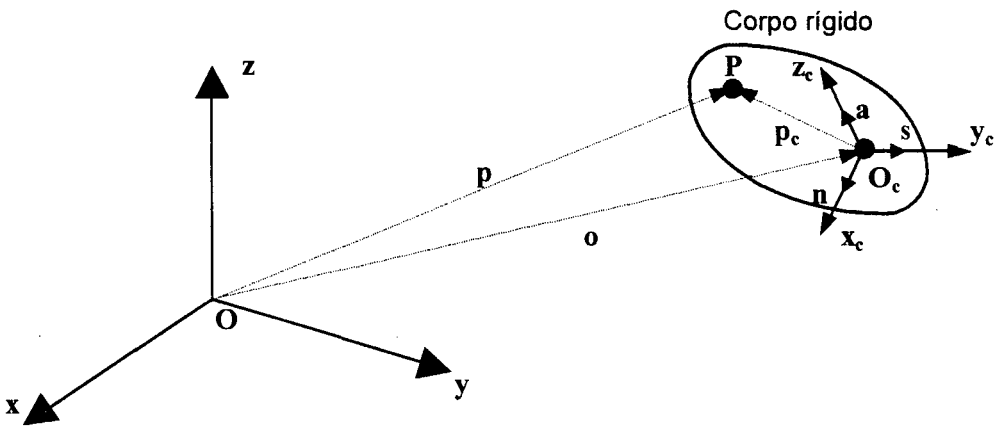


Figura 2.2 Posição e orientação de um corpo rígido

Considerando-se um ponto $\mathbf{O_c}$ do corpo rígido, ao qual se vincula um sistema de coordenadas fixo ao elo, pode-se descrever o corpo rígido no espaço através de uma matriz da forma

$$\mathbf{A} = \left[\begin{array}{c|c} \mathbf{R} & \mathbf{o} \\ \hline 0 \quad 0 \quad 0 & 1 \end{array}\right] \tag{2.1}$$

onde o vetor **o** corresponde à translação de $\mathbf{O_c}$ da origem do sistema de coordenadas de referência (que define a posição do elo), e **R** representa a orientação do corpo rígido em relação a este sistema. A matriz **R** é composta pelos vetores unitários nas direções $\mathbf{x_c}$, $\mathbf{y_c}$ e $\mathbf{z_c}$, expressos segundo o sistema de referência, tendo a forma

$$\mathbf{R} = \begin{bmatrix} \mathbf{n} & \mathbf{s} & \mathbf{a} \end{bmatrix} = \begin{bmatrix} n_x & s_x & a_x \\ n_y & s_y & a_y \\ n_z & s_z & a_z \end{bmatrix} \tag{2.2}$$

Como **n**, **s**, **a** são ortogonais entre si, a chamada *matriz de rotação* **R** é *ortonormal*, ou seja, $\mathbf{R}^T\mathbf{R} = \mathbf{I}$ ou, ainda, $\mathbf{R}^T = \mathbf{R}^{-1}$. Esta matriz pode ser interpretada como

- a orientação de um sistema de coordenadas em relação a outro;
- a relação entre as coordenadas de um mesmo ponto em sistemas distintos;
- um operador que determina a rotação de um ponto segundo um mesmo sistema de coordenadas.

Uma orientação pode ser definida por um conjunto de rotações, combinadas através da multiplicação de suas matrizes de rotação. Em particular, é comum definir-se uma orientação através de combinações de rotações elementares em torno dos eixos de coordenadas[6], ou por uma rotação em torno de um eixo arbitrário[1].

A matriz **A** é uma *matriz de transformação homogênea*, que compreende não apenas as rotações representadas por **R**, como também uma translação definida pela posição da origem do sistema de coordenadas[7]. Como ilustração, sejam **p** e $\mathbf{p}_c$ representações do ponto **P** segundo o sistema de referência e o sistema do corpo rígido, respectivamente (v. Figura 2.2). A relação entre estas é dada por

$$\begin{bmatrix} \mathbf{p} \\ \cdots \\ 1 \end{bmatrix} = \mathbf{A} \begin{bmatrix} \mathbf{p}_c \\ \cdots \\ 1 \end{bmatrix} \tag{2.3}$$

$$\mathbf{p} = \mathbf{o} + \mathbf{R}\mathbf{p}_c \tag{2.4}$$

A relação inversa pode ser obtida multiplicando-se (2.4) por $\mathbf{R}^T$, definindo a matriz

$$\mathbf{A}^{-1} = \left[\begin{array}{ccc|c} & \mathbf{R}^T & & -\mathbf{R}^T\mathbf{o} \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \tag{2.5}$$

---

[6] Como qualquer operação de multiplicação de matrizes, a composição de matrizes de rotação não é comutativa, ou seja, $\mathbf{R}_1\mathbf{R}_2 \neq \mathbf{R}_2\mathbf{R}_1$ em geral. Isso significa que a ordem de composição das rotações é importante para o resultado final.

[7] Na verdade, essa matriz pode ser usada para representar outros tipos de transformações, como escalas, distorções ou perspectivas, sendo estas mais utilizadas em computação gráfica [13].

Assim como as matrizes de rotação, as transformações homogêneas podem ser compostas através de transformações elementares pela multiplicação das respectivas matrizes.

2.1.2 NOTAÇÃO DENAVIT-HARTENBERG E A EQUAÇÃO CINEMÁTICA DIRETA

Considerando o manipulador uma cadeia cinemática aberta, verifica-se que a posição do efetuador final é função da configuração dessa cadeia, ou seja, da geometria dos elos e das posições relativas entre eles, definidas pelas variáveis das juntas (v. Figura 2.1).

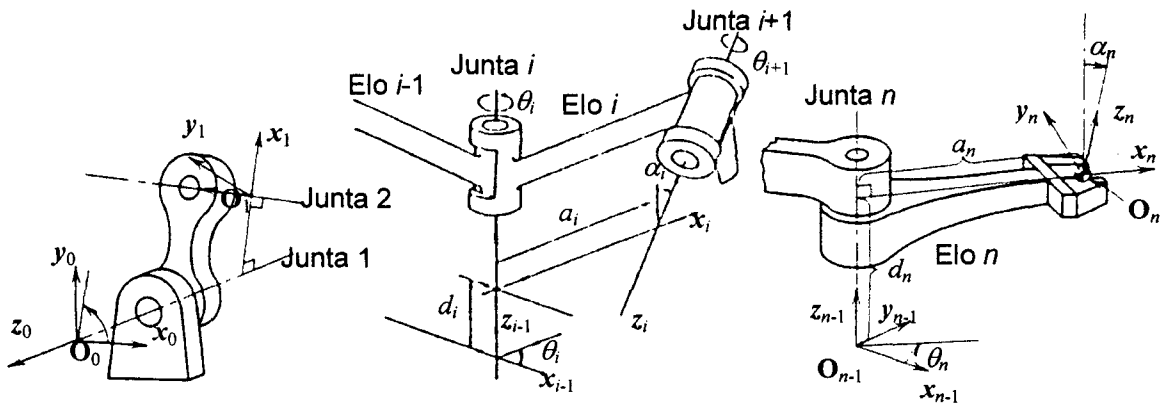


Figura 2.3 Parâmetros cinemáticos Denavit-Hartenberg

A *notação Denavit-Hartenberg* é empregada para descrever sistematicamente a relação cinemática entre os elos através de um número mínimo de parâmetros. Estes são utilizados para a determinação de matrizes de transformação homogênea que relacionam cada elo a seu anterior, sendo obtidos através dos passos abaixo[1,6]:

1. Localizar os eixos das juntas: Numeram-se os elos de 0 (base) a *n* (efetuador final). As juntas são numeradas de 1 a *n*. O eixo de uma junta *i* é aquele sobre o qual realiza-se o movimento relativo entre os elos *i*-1 e *i*. Sobre os eixos das juntas definem-se os eixos $\mathbf{z}_{i-1}$ dos sistemas de coordenadas dos elos.

2. Estabelecer o sistema de coordenadas da base: A origem do sistema de coordenadas da base é definida em qualquer ponto sobre o eixo $\mathbf{z}_0$. Os eixos $\mathbf{x}_0$ e $\mathbf{y}_0$ são orientados arbitrariamente, de forma a manter o sistema ortonormal (v. Figura 2.3a).

3. Estabelecer os sistemas de coordenadas dos elos 1...n-1: Localiza-se a origem do sistema do elo i na interseção do eixo $\mathbf{z}_i$ com a normal comum aos eixos $\mathbf{z}_{i-1}$ e $\mathbf{z}_i$. A origem $\mathbf{O}_i$ é escolhida arbitrariamente se $\mathbf{z}_{i-1}$ e $\mathbf{z}_i$ forem paralelos[8] ou no caso de juntas prismáticas[9]. O eixo $\mathbf{x}_i$ é definido sobre a normal comum a $\mathbf{z}_{i-1}$ e $\mathbf{z}_i$, no sentido $\mathbf{z}_{i-1}$-$\mathbf{z}_i$ (v. Figura 2.3b). Havendo interseção entre estes eixos, $\mathbf{x}_i$ é definido sobre a normal ao plano definido por $\mathbf{z}_{i-1}$ e $\mathbf{z}_i$. O eixo $\mathbf{y}_i$ é definido segundo a regra da mão direita.

4. Definir o sistema de coordenadas do elo n: A única restrição para a definição desse sistema é que o eixo $\mathbf{x}_n$ seja normal ao eixo $\mathbf{z}_{n-1}$. Arbitram-se tanto a origem quanto os eixos $\mathbf{z}_n$ e $\mathbf{y}_n$.

5. Estabelecer os parâmetros Denavit-Hartenberg: Para os elos de 1 a n, os seguintes parâmetros devem ser identificados:

   - a_i: comprimento da normal comum entre $\mathbf{z}_{i-1}$ e $\mathbf{z}_i$ (*comprimento* do elo i);
   - α_i: ângulo entre $\mathbf{z}_{i-1}$ e $\mathbf{z}_i$, medido segundo o eixo $\mathbf{x}_i$ (*inclinação* entre os eixos);
   - d_i: distância de $\mathbf{O}_{i-1}$ à interseção da normal comum a $\mathbf{z}_{i-1}$ e $\mathbf{z}_i$, medido segundo $\mathbf{z}_{i-1}$;
   - θ_i: ângulo entre os eixos $\mathbf{x}_{i-1}$ e $\mathbf{x}_i$, medido segundo o eixo $\mathbf{z}_{i-1}$.

   Os parâmetros a_i e α_i são constantes, determinados pela geometria do elo i, enquanto $d_i(\theta_i)$ corresponde à variável da junta prismática(revoluta).

6. Determinar as matrizes de transformação: Com os parâmetros Denavit-Hartenberg, determinam-se as matrizes de transformação homogênea entre os sistemas de cada elo e os sistemas de seus antecessores imediatos na cadeia cinemática. Para um elo i, a matriz é definida por[1,2,6]

$$\mathbf{A}_i^{i-1} = \begin{bmatrix} \cos\theta_i & -\operatorname{sen}\theta_i \cos\alpha_i & \operatorname{sen}\theta_i \operatorname{sen}\alpha_i & a_i \cos\theta_i \\ \operatorname{sen}\theta_i & \cos\theta_i \cos\alpha_i & -\cos\theta_i \operatorname{sen}\alpha_i & a_i \operatorname{sen}\theta_i \\ 0 & \operatorname{sen}\alpha_i & \cos\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.6}$$

[8] Pois a normal comum não é unicamente definida. Em geral, procura-se escolher a origem de forma a simplificar a cinemática direta e, para tanto, d_i=0.

[9] Nessas, somente a direção do eixo da junta é significativa. Havendo interseção entre os eixos $\mathbf{z}_{i-1}$ e $\mathbf{z}_i$ a origem será o ponto de interseção.

Alguns autores preferem usar uma notação Denavit-Hartenberg *modificada*, onde o eixo $\mathbf{z}_i$ é definido sobre o eixo da junta i, implicando em diferentes especificações dos parâmetros e das matrizes de transformação[10]. YOSHIKAWA[14] afirma que, a princípio, qualquer escolha de sistemas de coordenadas para os elos onde os eixos $\mathbf{z}_i$ estejam alinhados com os eixos das juntas tem a aplicabilidade da notação Denavit-Hartenberg. Neste trabalho, será adotada a notação Denavit-Hartenberg convencional.

Após a determinação das relações entre os elos, obtém-se a cinemática direta pela composição das matrizes de transformação que estabelecem essas relações. Assim, a posição do efetuador final em relação à base do manipulador é dada por

$$\mathbf{T}(\mathbf{q}) = \left[\begin{array}{ccc|c} \mathbf{n} & \mathbf{s} & \mathbf{a} & \mathbf{p} \\ \hline 0 & 0 & 0 & 1 \end{array}\right] = \mathbf{A}_1^0(q_1)\mathbf{A}_2^1(q_2)\cdots\mathbf{A}_i^{i-1}(q_i)\cdots\mathbf{A}_n^{n-1}(q_n) \qquad (2.7)$$

onde $\mathbf{A}_i^{i-1}(.)$ corresponde à transformação do sistema do elo i para o sistema do elo i-1 e q_i representa a variável da junta i[11].

2.1.3 ESPAÇO DAS JUNTAS E ESPAÇO OPERACIONAL

A configuração de um manipulador, que define a posição do efetuador final, pode ser descrita pelo vetor $\mathbf{q}[n\text{x}1]$ de variáveis das juntas. Define-se, então, o *espaço das juntas* como o conjunto de valores que $\mathbf{q}$ pode assumir correspondendo a todas configurações possíveis para o manipulador.

Em muitos casos, uma tarefa consiste na definição da posição e orientação do efetuador final ao longo do tempo, por ser este o elemento que realmente efetua o trabalho[1]. Tais tarefas podem ser especificadas em função das variáveis das juntas, embora seja preferido o emprego de um conjunto de variáveis que reflita diretamente as posições e orientações do efetuador final, por ser de emprego mais direto e natural no planejamento da atividade a ser executada pelo manipulador[2].

---

[10] Para a discussão dessa abordagem, v. DE WIT et al.[7].

[11] Esta é uma notação genérica, onde q_i corresponde à d_i para juntas de translação e à θ_i para juntas de revolução.

Para a posição do efetuador final, basta definir as três coordenadas segundo um sistema referencial. Por outro lado, especificar a orientação do vetor através de uma matriz de rotação é uma tarefa difícil, pois esta deve manter as condições de ortonormalidade impostas pela relação $\mathbf{R}^T\mathbf{R}=\mathbf{I}$. Este problema admite uma solução natural ao se adotar uma *representação mínima* para a orientação[7], permitindo agrupar posição e orientação em notação vetorial, como

$$\mathbf{x} = \begin{bmatrix} \mathbf{p} \\ \phi \end{bmatrix} \tag{2.8}$$

onde $\mathbf{p}$ corresponde à posição e ϕ corresponde à orientação do efetuador final[12]. Diz-se que o vetor $\mathbf{x}[m\text{x}1]$, onde $m \leq n$, é definido no *espaço operacional*, por ser o espaço no qual a tarefa é especificada[2]. Esse espaço tem dimensão $m \leq 6$, de acordo com a geometria da tarefa[7].

Levando em conta a relação entre a posição $\mathbf{x}$ do efetuador final e a configuração $\mathbf{q}$ das variáveis das juntas, pode-se reescrever a equação (2.7) como

$$\mathbf{x} = \mathbf{k}(\mathbf{q}) \tag{2.9}$$

onde $\mathbf{k}(.)[m\text{x}1]$ é uma função vetorial geralmente não linear, cuja dependência explícita de $\mathbf{q}$ não costuma existir a não ser em casos simples, devido à orientação. No caso mais geral, $\phi(\mathbf{q})$ é determinado em função da matriz de rotação em (2.7), podendo ter mais de uma solução[2,7].

Uma representação mínima para a orientação consiste na utilização de três parâmetros independentes para sua definição[13]. Duas abordagens comumente empregadas para a definição de uma representação mínima são os *ângulos de Euler* e os *ângulos RPY* [1,2,6,7].

Os ângulos de Euler definem a orientação através de três rotações consecutivas sobre o sistema de coordenadas. Em uma das representações usuais, realiza-se inicialmente

---

[12] Embora expresso em notação vetorial, $\mathbf{x}$ não é um vetor, pois a orientação não é comutativa[6].

[13] Pois verifica-se que o número de graus de liberdade rotacional de um corpo livre é no máximo 3. Os nove elementos de uma matriz de rotação, por exemplo, são relacionados por seis restrições devido às condições de ortogonalidade dos vetores $\mathbf{n}$, $\mathbf{s}$ e $\mathbf{a}$, sobrando os três graus de liberdade na definição da orientação[1].

uma rotação φ em torno do eixo **z**, seguindo-se uma rotação θ em torno do eixo **y'** e por fim uma rotação ψ em torno do eixo **z"**[14] (v. Figura 2.4). A composição das rotações elementares resulta na matriz de rotação

$$\mathbf{R} = \begin{bmatrix} c_\varphi c_\theta c_\psi - s_\varphi s_\psi & -c_\varphi c_\theta s_\psi - s_\varphi c_\psi & c_\varphi s_\theta \\ s_\varphi c_\theta c_\psi + c_\varphi s_\psi & -s_\varphi c_\theta s_\psi + c_\varphi c_\psi & s_\varphi s_\theta \\ -s_\theta c_\psi & s_\theta s_\psi & c_\theta \end{bmatrix} \tag{2.10}$$

onde c_α , s_α correspondem ao coseno e ao seno do ângulo α, respectivamente.

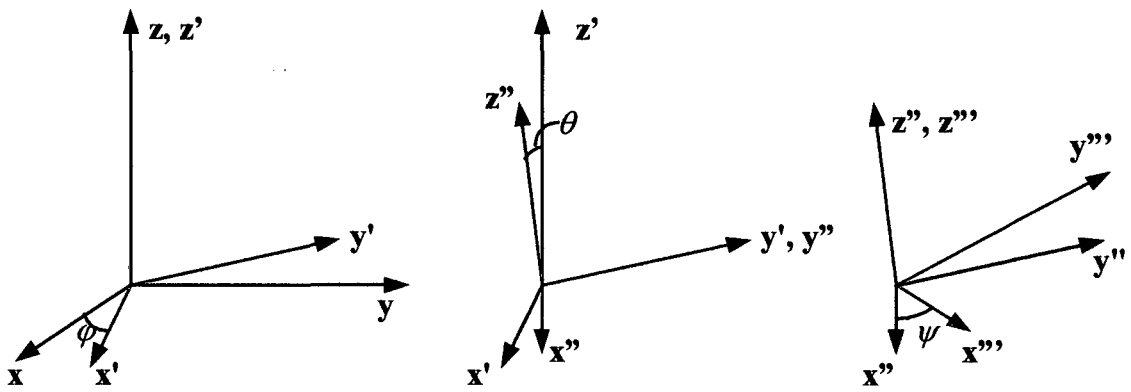


Figura 2.4 Representação dos ângulos de Euler ZYZ

A solução do problema inverso, ou seja, determinar os ângulos de Euler a partir de uma matriz de rotação, é bastante útil, sendo apresentada abaixo[15].

$$\varphi = \text{atan}(\pm r_{23}, \pm r_{13}) \tag{2.11a}$$

$$\theta = \text{atan}\left(\pm\sqrt{r_{13}^2 + r_{23}^2}, r_{33}\right) \tag{2.11b}$$

$$\psi = \text{atan}(\pm r_{32}, \mp r_{31}) \tag{2.11c}$$

onde atan(.) é a função arco tangente com dois argumentos e r_{ij} é o elemento da linha i coluna j da matriz de rotação **R**.

[14] Na verdade, existem doze possibilidades distintas para compor os ângulos de Euler, correspondentes às combinações das rotações elementares, sendo a notação ZYZ a mais empregada[2].
[15] A demonstração da obtenção da solução está em SPONG e VIDYASAGAR[1] e SCIAVICCO e SICILIANO[2].

Pode-se perceber, pelas equações acima, que é possível encontrar mais de um conjunto de ângulos de Euler para uma mesma matriz de rotação, dependendo das considerações de sinais e da faixa de valores admitidas para os ângulos. Além disso, quando s_θ=0, a solução degenera, sendo possível determinar apenas a soma ou a diferença entre φ e ψ. Nesses casos, as rotações sucessivas de φ e ψ são feitas em torno de eixos paralelos, dando contribuições equivalentes à rotação final e possibilitando infinitas soluções[2].

Assim como nos ângulos de Euler, os ângulos RPY definem a orientação por três rotações. A diferença está nestas, que são realizadas segundo os eixos **x**(*yaw*), **y**(*pitch*), **z**(*roll*), que se *mantém fixos* (v. Figura 2.5). A matriz de rotação resultante é

$$\mathbf{R} = \begin{bmatrix} c_\varphi c_\theta & -s_\varphi c_\psi + c_\varphi s_\theta s_\psi & s_\varphi s_\psi + c_\varphi s_\theta c_\psi \\ s_\varphi c_\theta & c_\varphi c_\psi + s_\varphi s_\theta s_\psi & -c_\varphi s_\psi + s_\varphi s_\theta c_\psi \\ -s_\theta & c_\theta s_\psi & c_\theta c_\psi \end{bmatrix} \tag{2.12}$$

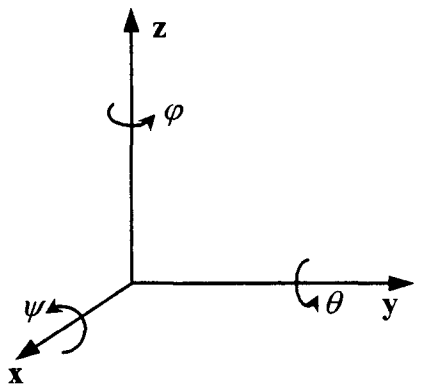


Figura 2.5 Representação dos ângulos *Roll-Pitch-Yaw*

A solução do problema inverso, é, como nos ângulos de Euler, baseada na análise da matriz de rotação resultante, sendo mostrada abaixo.

$$\varphi = \text{atan}(\pm r_{21}, \pm r_{11}) \tag{2.13a}$$

$$\theta = \text{atan}\left(-r_{31}, \pm\sqrt{r_{32}^2 + r_{33}^2}\right) \tag{2.13b}$$

$$\psi = \text{atan}(\pm r_{32}, \pm r_{33}) \tag{2.13c}$$

Da mesma forma, verifica-se que é possível encontrar mais de uma solução para uma mesma matriz de rotação, dependendo das considerações de sinais e da faixa de valores admitidas para os ângulos, havendo degeneração quando $c_\theta=0$, onde consegue-se determinar apenas a soma ou a diferença entre φ e ψ[2].

A definição dos espaços de junta e operacional introduz o conceito de *redundância cinemática*, que ocorre quando a dimensão do espaço das juntas é maior que a dimensão do espaço operacional. Esse conceito também é relativo à tarefa a ser executada, pois depende do conjunto de variáveis do espaço operacional considerado[7].

2.2 CINEMÁTICA INVERSA

A especificação de tarefas no espaço operacional cria a necessidade de se determinar a configuração correspondente no espaço das juntas. Esse é o problema da *cinemática inversa*, que pode ser equacionado como

$$\mathbf{q} = \mathbf{k}^{-1}(\mathbf{x}) \tag{2.14}$$

Esse problema é consideravelmente mais complexo que a cinemática direta, pois enquanto esta resulta em uma única solução para um conjunto de variáveis das juntas[16], a cinemática inversa pode apresentar *múltiplas soluções*[17] ou *nenhuma solução realizável*[18] para a estrutura do manipulador[6]. Além disso, como a cinemática direta é composta de equações não lineares simultâneas com muitas funções trigonométricas, nem sempre é possível obter uma solução fechada correspondente à sua inversa. Para estes casos, torna-se necessário o emprego de métodos numéricos para tal solução, pois estes são aplicáveis a qualquer estrutura cinemática, embora não permitam, em geral, o cálculo de todas as soluções admissíveis[2,7].

A obtenção de uma solução fechada é preferível ou uso de técnicas numéricas, por apresentar maior velocidade de resposta e pela possibilidade de estabelecimento de regras

[16] Verdadeiro para a equação (2.7), mas nota-se que, para (2.8), a determinação de uma representação mínima para a orientação pode levar a mais de uma solução[2].
[17] No caso de manipuladores redundantes podem existir infinitas soluções para uma mesma posição **x**.

para a escolha de uma solução entre as várias disponíveis[1]. A sua existência depende da estrutura cinemática do manipulador, sendo por isso que a maioria dos robôs industriais é projetada de forma a ter *estrutura solúvel*[6].

Mesmo tendo uma estrutura solúvel, a existência de uma ou mais soluções para a cinemática inversa depende de fatores como o número de parâmetros Denavit-Hartenberg não nulos, o número de graus de liberdade do manipulador e dos limites de valores para as variáveis das juntas. Além de estrutura solúvel, muitos manipuladores industriais são projetados de forma que estes limites forcem a obtenção de uma solução única para a cinemática inversa [15].

Vários autores propuseram soluções para manipuladores de diferentes estruturas. Um caso particular bastante citado é o do manipulador de 6 graus de liberdade com punho esférico, onde os eixos das três últimas juntas se encontram em um ponto. Nesse caso, o problema pode ser dividido em dois subproblemas, a *cinemática inversa da posição* e a *cinemática inversa da orientação*. Roteiros para a solução desses subproblemas são apresentados em SPONG e VIDYASAGAR[1] e SCIAVICCO e SICILIANO[2].

2.3 CINEMÁTICA DIFERENCIAL

A relação entre as velocidades lineares e angulares do efetuador final e as velocidades das juntas é uma das mais importantes, aparecendo em praticamente todos os aspectos da robótica, como no planejamento de trajetórias, na determinação de singularidades cinemáticas, na coordenação de movimentos, na determinação das equações dinâmicas do movimento e na relação entre as forças e os torques que atuam no manipulador[1]. O objetivo da cinemática diferencial é determinar essa relação, descrita através de uma matriz denominada *Jacobiano*, que é função da configuração do manipulador[2].

A equação cinemática diferencial tem a forma

[18] Quando se determina uma posição para o efetuador final que não se localize no espaço de trabalho do manipulador.

$$\mathbf{v} = \begin{bmatrix} \dot{\mathbf{p}} \\ \omega \end{bmatrix} = \mathbf{J}(\mathbf{q})\dot{\mathbf{q}} \tag{2.15}$$

onde a velocidade $\mathbf{v}$ do efetuador final é composta por velocidades lineares ($\dot{\mathbf{p}}$) e angulares (ω), $\dot{\mathbf{q}}$ é o vetor de velocidades das juntas e $\mathbf{J}(\mathbf{q})$ [6xn] é o Jacobiano.

2.3.1 JACOBIANO GEOMÉTRICO

O Jacobiano é usualmente determinado através de procedimentos geométricos baseados na contribuição de cada velocidade de junta para as velocidades linear e angular do efetuador final, sendo geralmente função das variáveis das juntas. Esta contribuição é expressa como

$$\dot{\mathbf{p}} = \mathbf{J}_{p1}\dot{\mathbf{q}}_1 + \cdots + \mathbf{J}_{pi}\dot{\mathbf{q}}_i + \cdots + \mathbf{J}_{pn}\dot{\mathbf{q}}_n \tag{2.16a}$$

$$\omega = \mathbf{J}_{o1}\dot{\mathbf{q}}_1 + \cdots + \mathbf{J}_{oi}\dot{\mathbf{q}}_i + \cdots + \mathbf{J}_{on}\dot{\mathbf{q}}_n \tag{2.16b}$$

onde $\mathbf{J}_{pi}$ e $\mathbf{J}_{oi}$, relativos à velocidade linear e angular do efetuador final, compõem a coluna i do Jacobiano. Esta pode ser determinada pela relação

$$\begin{bmatrix} \mathbf{J}_{pi} \\ \mathbf{J}_{oi} \end{bmatrix} = \begin{bmatrix} \mathbf{a}_{i-1} \\ \mathbf{0} \end{bmatrix} \quad \text{, para juntas prismáticas} \tag{2.17a}$$

$$\begin{bmatrix} \mathbf{J}_{pi} \\ \mathbf{J}_{oi} \end{bmatrix} = \begin{bmatrix} \mathbf{a}_{i-1} \times \mathbf{r}_{i-1,n} \\ \mathbf{a}_{i-1} \end{bmatrix} \quad \text{, para juntas rotacionais} \tag{2.17b}$$

onde $\mathbf{a}_{i-1}$ é um vetor unitário na direção do eixo $\mathbf{z}_{i-1}$, e

$$\mathbf{r}_{i-1,n} = \mathbf{p} - \mathbf{o}_{i-1} \tag{2.18}$$

é o vetor entre a origem do sistema do elo i-1 e a posição do efetuador final, expresso segundo o sistema operacional[19]. O vetor $\mathbf{p}$ é obtido da cinemática direta, enquanto os vetores $\mathbf{a}_{i-1}$ e $\mathbf{o}_{i-1}$ podem ser obtidos na matriz de transformação homogênea entre o elo i-1

[19] Para ver a obtenção dos termos para as colunas do Jacobiano, v. SCIAVICCO e SICILIANO[2] e ASADA e SLOTINE[6].

e a base ($\mathbf{a}_{i-1}$ é a 3ª coluna da matriz de rotação, e $\mathbf{o}_{i-1}$ corresponde aos três primeiros elementos da 4ª coluna da matriz).

A matriz jacobiana depende do sistema de coordenadas considerado. Se a velocidade do efetuador final é expressa em um sistema que não o da base, pode-se transformar o Jacobiano através da matriz de rotação **R** relativa entre os dois sistemas, resultando em

$$\mathbf{J}_{\pi} = \begin{bmatrix} \mathbf{R} & \mathbf{0} \\ \mathbf{0} & \mathbf{R} \end{bmatrix} \mathbf{J} \tag{2.19}$$

onde $\mathbf{J}_{\pi}$ é o Jacobiano expresso no novo sistema de coordenadas[2].

2.3.2 JACOBIANO ANALÍTICO

O Jacobiano também pode ser determinado de forma analítica, através da derivação da equação cinemática direta (2.9) em função das variáveis das juntas, ou seja,

$$\dot{\mathbf{x}} = \begin{bmatrix} \dot{\mathbf{p}} \\ \\ \dot{\phi} \end{bmatrix} = \begin{bmatrix} \dfrac{\partial \mathbf{p}}{\partial \mathbf{q}} \\ \text{---} \\ \dfrac{\partial \phi}{\partial \mathbf{q}} \end{bmatrix} \dot{\mathbf{q}} = \mathbf{J}_a \dot{\mathbf{q}} \tag{2.20}$$

sendo $\mathbf{J}_a$ denominado *Jacobiano analítico*, definido como

$$\mathbf{J}_a = \frac{\partial \mathbf{k}(\mathbf{q})}{\partial \mathbf{q}} \tag{2.21}$$

Percebe-se facilmente que o Jacobiano analítico geralmente difere do Jacobiano geométrico, pois o vetor velocidade angular ω não corresponde a $\dot{\phi}$, que depende da representação mínima adotada. Verifica-se, porém, que a parte correspondente às velocidades lineares é a mesma nos dois Jacobianos. A equivalência entre estes depende, então, da obtenção de uma relação entre $\dot{\phi}$ e ω. Para uma representação mínima utilizando ângulos de Euler, por exemplo, esta relação corresponde à matriz de transformação[2]

$$\omega = \begin{bmatrix} 0 & -s_\varphi & c_\varphi s_\theta \\ 0 & c_\varphi & s_\varphi s_\theta \\ 1 & 0 & c_\theta \end{bmatrix} \begin{bmatrix} \dot{\varphi} \\ \dot{\theta\varphi} \\ \dot{\psi} \end{bmatrix} = \mathbf{T}(\phi)\dot{\phi} \tag{2.22}$$

onde s_α e c_α são, respectivamente, o seno e o coseno do ângulo α. A relação entre o Jacobiano geométrico e o Jacobiano analítico é, então, escrita como

$$\mathbf{J} = \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ \mathbf{0} & \mathbf{T}(\phi) \end{bmatrix} \mathbf{J}_a = \mathbf{T}_a(\phi)\mathbf{J}_a \tag{2.23}$$

Esta relação pode ser escrita para qualquer outra representação mínima da orientação, mudando apenas a matriz **T**(.).

O Jacobiano geométrico é mais intuitivo do ponto de vista das grandezas físicas, enquanto o Jacobiano analítico é útil na análise das variáveis do espaço operacional[2,7]. A relação entre os Jacobianos só não é possível quando **T**(.) é singular, sendo as orientações para as quais o determinante de **T** é indeterminado denominadas *singularidades de representação* da orientação[7].

2.3.3 SINGULARIDADES CINEMÁTICAS

No conjunto de configurações de um manipulador, podem existir situações em que o Jacobiano tenha deficiência de posto. Essas são as chamadas *singularidades cinemáticas*, e apresentam as seguintes características[1]:

- mobilidade reduzida, ou seja, certas direções de movimento não são atingíveis;
- podem corresponder a pontos limites no espaço de trabalho;
- velocidades e forças finitas no efetuador final podem corresponder a velocidades e torques infinitos nas juntas;
- nas proximidades não há uma solução única para a cinemática inversa, podendo existir infinitas soluções ou nenhuma solução possível;

sendo portanto de grande importância sua determinação. O meio mais simples de análise de singularidades é o estudo do determinante de **J**, onde diferentes tipos de singularidades podem ser detectadas[7].

Analogamente à cinemática inversa, para manipuladores com n=6 utilizando punhos esféricos é possível *desacoplar a singularidade* em singularidade do braço e singularidade do punho[1]. Particularmente, o punho é singular quando os eixos $\mathbf{z}_3$ e $\mathbf{z}_5$ estão alinhados (v. Figura 2.6). As singularidades do braço são características da estrutura do manipulador.

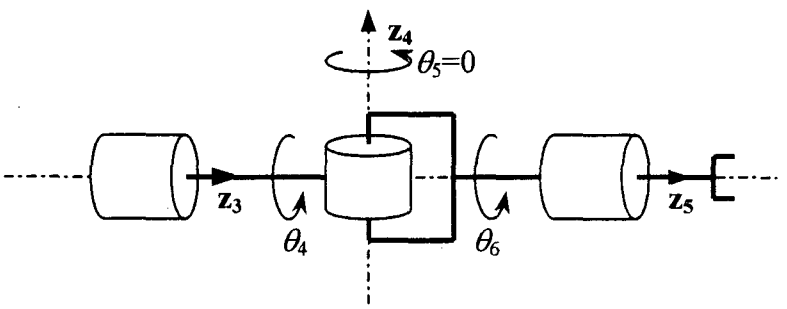


Figura 2.6 Punho esférico em configuração singular

As singularidades aumentam a dimensão do *espaço nulo* de **J** (conjunto de velocidades de junta que causam velocidade nula no efetuador final). Em casos de redundância, esse valor é naturalmente diferente de 0, significando que existe mais de uma solução para a cinemática inversa. Assim, pelo aumento da dimensão do espaço nulo, a singularidade pode levar a infinitas soluções para a cinemática inversa.

2.3.4 Cinemática Diferencial Inversa

A relação inversa entre velocidades é mais simples que a cinemática inversa. No caso de manipuladores não redundantes, em que o Jacobiano tem posto completo, obtém-se as velocidades das juntas relativas à velocidade no efetuador final através da expressão

$$\dot{\mathbf{q}} = \mathbf{J}(\mathbf{q})^{-1}\mathbf{v} \tag{2.24}$$

Para manipuladores redundantes, a inversa tem solução se **v** encontra-se na imagem de **J**(**q**). Nesse caso, obtém-se uma solução ótima através do uso da pseudoinversa de **J** [2,6], resultando em

$$\dot{\mathbf{q}} = \mathbf{J}(\mathbf{q})^{\dagger}\mathbf{v} \tag{2.25}$$

$$\mathbf{J}^{\dagger} = \mathbf{J}^{\mathrm{T}}\left(\mathbf{J}\mathbf{J}^{\mathrm{T}}\right)^{-1} \tag{2.26}$$

Para manipuladores não redundantes, (2.25) reduz-se à (2.24).

A relação entre acelerações também é obtida pelo uso do Jacobiano. Derivando (2.15) ao longo do tempo, obtém-se

$$\dot{\mathbf{v}} = \mathbf{J}(\mathbf{q})\ddot{\mathbf{q}} + \dot{\mathbf{J}}(\mathbf{q})\dot{\mathbf{q}} \tag{2.27}$$

e a aceleração nas juntas é determinada pela relação inversa

$$\ddot{\mathbf{q}} = \mathbf{J}(\mathbf{q})^{-1}\left[\dot{\mathbf{v}} - \dot{\mathbf{J}}(\mathbf{q})\dot{\mathbf{q}}\right] \tag{2.28}$$

A cinemática diferencial inversa pode ser usada para se obter cinemática inversa do manipulador. Conhecendo o perfil de velocidade do efetuador final, utiliza-se (2.24) para determinar as velocidades das juntas, que integradas no tempo a partir de uma configuração inicial $\mathbf{q}_0$ levam a $\mathbf{q}$. A vantagem dessa solução é a generalidade do método.

Um problema que aparece ao se implementar numericamente este método está relacionado à discretização, como ilustrado na equação abaixo, em que se utiliza o método de Euler para a solução da equação.

$$\mathbf{q}(t_k) = \mathbf{q}(t_{k-1}) + \mathbf{J}_a^{-1}(\mathbf{q}(t_{k-1}))\dot{\mathbf{x}}(t_{k-1})\Delta t \tag{2.29}$$

Observa-se que a obtenção de $\mathbf{q}(t_k)$ depende do conhecimento de $\mathbf{q}(t_{k-1})$. O cálculo do Jacobiano também é feito em relação a t_{k-1}, não correspondendo ao valor esperado para o instante t_k. Este fenômeno, conhecido como *deriva*[20], leva a solução para valores das juntas que não correspondem à posição desejada para o efetuador final[2].

Para resolver esse problema, pode-se empregar o erro de posição no espaço operacional em uma malha de realimentação para se determinar as velocidades das juntas, em um esquema conhecido como cinemática inversa em malha fechada[16]. Neste, $\dot{\mathbf{q}}$ é definido como

[20] Do inglês *drift*.

$$\dot{\mathbf{q}} = \mathbf{J}_a^{-1}(\mathbf{q})(\dot{\mathbf{x}}_d + \mathbf{K}\mathbf{e}) \tag{2.30}$$

levando ao sistema equivalente

$$\dot{\mathbf{e}} + \mathbf{K}\mathbf{e} = \mathbf{0} \tag{2.31}$$

onde $\mathbf{e} = \mathbf{x}_d - \mathbf{x}$. Para uma matriz $\mathbf{K}$ definida positiva, o sistema é assintoticamente estável. Para sistemas redundantes, utiliza-se a pseudo-inversa no lugar da inversa. A Figura 2.7 esquematiza essa solução.

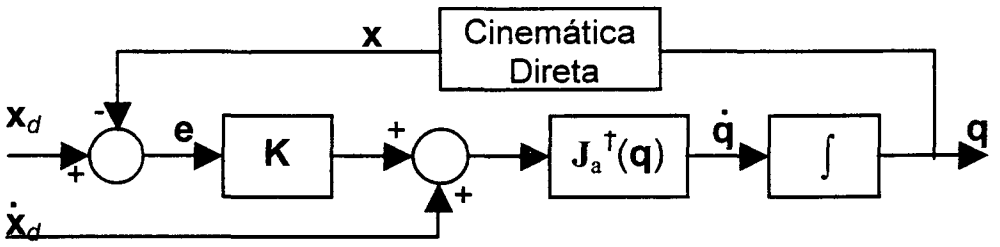


Figura 2.7 Esquema da cinemática inversa em malha fechada

2.4 CONCLUSÃO

Neste capítulo, foram analisados os aspectos inerentes à modelagem cinemática de um manipulador. Inicialmente foi apresentado um roteiro para obtenção da cinemática direta, sendo feitas as definições de espaço das juntas e espaço operacional. Seguiu-se uma discussão dos problemas da cinemática inversa, especialmente para a implementação genérica de modelos. Foram apresentadas as relações entre velocidades e acelerações no espaço das juntas e no espaço operacional. Por fim, foi demonstrado que a cinemática inversa pode ser obtida a partir da solução da cinemática diferencial inversa.

3. MODELAGEM DINÂMICA

O modelo dinâmico de um manipulador é importante para a simulação da execução de tarefas, pois este define a relação entre o movimento do manipulador e as forças e torques aplicados nos seus atuadores e no efetuador final. Assim, este capítulo tem por objetivo fazer uma revisão dos aspectos dinâmicos dos manipuladores necessários para a sua modelagem, complementando o resumo dos textos clássicos, como ASADA e SLOTINE[6], SPONG e VIDYASAGAR[1], SCIAVICCO e SICILIANO[2] e DE WIT et al.[7], entre outros, iniciado no capítulo de modelagem cinemática.

As *equações do movimento* que estabelecem essa relação entre o movimento e os esforços aplicados podem ser deduzidas através da *formulação Lagrangeana*, na qual o sistema dinâmico é definido a partir do trabalho e energia para um conjunto de coordenadas generalizadas (as variáveis das juntas, por exemplo), ou pela *formulação de Newton-Euler*, onde as equações são obtidas pelo balanço das forças e momentos atuantes nos elos do manipulador. Enquanto a primeira formulação é simples e sistemática, a segunda é considerada mais eficiente do ponto de vista computacional [2,7].

Neste capítulo são apresentadas ambas as formulações, seguidas de uma análise da implementação computacional do modelo dinâmico.

3.1 FORMULAÇÃO LAGRANGEANA

Na formulação Lagrangeana, as equações do movimento de um sistema são deduzidas em função de um conjunto de *coordenadas generalizadas* que o descrevem de

forma completa. Em um manipulador, as equações de Lagrange podem ser expressas em função do conjunto formado pelas variáveis das juntas, resultando em

$$\frac{d}{dt}\frac{\partial L}{\partial \dot{q}_i} - \frac{\partial L}{\partial q_i} = \tau_i, \qquad i = 1,\ldots,n \tag{3.1}$$

onde o Lagrangeano L é definido como a diferença entre a energia cinética e a energia potencial do sistema, ou

$$L = T\text{-}U \tag{3.2}$$

e τ_i é uma *força generalizada*[21] não conservativa na junta i. A formulação Lagrangeana da equação do movimento começa pela determinação das expressões da energia cinética e da energia potencial do sistema em função das variáveis das juntas.

A energia cinética total do sistema é dada pelo somatório das parcelas de energia cinética de cada elo. Para um elo i, a energia cinética T_i é igual a

$$T_i = \tfrac{1}{2}\left(m_i \dot{\mathbf{p}}_{ci}{}^{\mathrm{T}} \dot{\mathbf{p}}_{ci} + \omega_i{}^{\mathrm{T}} \mathbf{I}_i \omega_i\right) \tag{3.3}$$

onde m_i é a massa do elo i, $\dot{\mathbf{p}}_{ci}$ é a velocidade linear do seu centro de massa, ω_i é a sua velocidade angular e $\mathbf{I}_i$ é o tensor de inércia relativo ao centro de massa do elo, expressos segundo um sistema de referência inercial[22] (v. Figura 3.1).

Sendo expresso segundo um sistema de referência inercial, o tensor de inércia é dependente da configuração do manipulador. Assim, é preferível que o termo rotacional da energia cinética seja definido segundo o sistema de coordenadas do elo, uma vez que o tensor de inércia então é constante[23]. A velocidade angular do elo neste sistema relaciona-se com a sua representação no sistema inercial pela matriz de rotação entre os dois sistemas,

---

[21] Correspondente a força em juntas prismáticas ou torque em juntas revolutas. Em geral, as forças generalizadas são referidas como *torques*, visto que a maioria das juntas nos manipuladores são rotacionais[7].

[22] Será considerado como sistema inercial o sistema de coordenadas da base.

[23] Os valores dos termos translacional e rotacional da energia cinética de um elo não são afetados pelo sistema de referência utilizado. O único cuidado necessário é que o tensor de inércia e a velocidade angular do elo sejam expressos segundo o mesmo sistema referencial[1].

$$\omega_i^i = \mathbf{R}_i^{\mathrm{T}} \omega_i \tag{3.4}$$

e a energia cinética pode ser reescrita como

$$T_i = \tfrac{1}{2}\left(m_i \dot{\mathbf{p}}_{ci}^{\mathrm{T}} \dot{\mathbf{p}}_{ci} + \omega_i^{\mathrm{T}} \mathbf{R}_i \mathbf{I}_i^i \mathbf{R}_i^{\mathrm{T}} \omega_i\right) \tag{3.5}$$

onde $\mathbf{I}_i^i$ é o tensor de inércia definido segundo o sistema de coordenadas do elo.

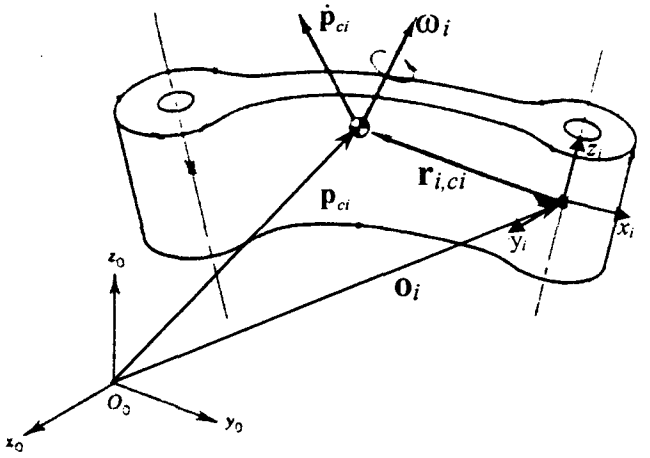


Figura 3.1 Representação cinemática de um elo *i* genérico

Para expressar a energia cinética em função das variáveis das juntas, as velocidades dos elos podem ser relacionadas às velocidades das juntas através da utilização do método de determinação do Jacobiano geométrico nos elos intermediários:

$$\begin{bmatrix} \dot{\mathbf{p}}_{ci} \\ \omega_i \end{bmatrix} = \begin{bmatrix} \mathbf{J}_p^{(i)} \\ \mathbf{J}_o^{(i)} \end{bmatrix} \dot{\mathbf{q}} \tag{3.6}$$

onde

$$\begin{bmatrix} \mathbf{J}_p^{(i)} \\ \mathbf{J}_o^{(i)} \end{bmatrix} = \begin{bmatrix} \mathbf{J}_{p1}^{(i)} & \cdots & \mathbf{J}_{pi}^{(i)} & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{J}_{o1}^{(i)} & \cdots & \mathbf{J}_{oi}^{(i)} & \mathbf{0} & \cdots & \mathbf{0} \end{bmatrix} \tag{3.7}$$

cujas colunas são definidas como

$$\begin{bmatrix} \mathbf{J}_{pj}^{(i)} \\ \mathbf{J}_{oj}^{(i)} \end{bmatrix} = \begin{bmatrix} \mathbf{a}_{j-1} \\ \mathbf{0} \end{bmatrix} \quad \text{, para juntas prismáticas} \tag{3.8a}$$

$$\begin{bmatrix} \mathbf{J}_{pj}^{(i)} \\ \mathbf{J}_{oj}^{(i)} \end{bmatrix} = \begin{bmatrix} \mathbf{a}_{j-1} \times \left(\mathbf{p}_{ci} - \mathbf{p}_{j-1}\right) \\ \mathbf{a}_{j-1} \end{bmatrix} \quad \text{, para juntas rotacionais} \tag{3.8b}$$

onde $\mathbf{p}_{ci}$ e $\mathbf{p}_{j-1}$ são os vetores posição do centro de massa do elo i e da origem do sistema do elo j-1, respectivamente e $\mathbf{a}_{j-1}$ é o vetor unitário na direção do eixo $\mathbf{z}_{j-1}$.

Assim, (3.5) é reescrita como

$$T_i = \tfrac{1}{2}\left(m_i \dot{\mathbf{q}}^{\mathrm{T}} \mathbf{J}_p^{(i)^{\mathrm{T}}} \mathbf{J}_p^{(i)} \dot{\mathbf{q}} + \dot{\mathbf{q}}^{\mathrm{T}} \mathbf{J}_o^{(i)^{\mathrm{T}}} \mathbf{R}_i \mathbf{I}_i^i \mathbf{R}_i^{\mathrm{T}} \mathbf{J}_o^{(i)} \dot{\mathbf{q}}\right) \tag{3.9}$$

Sendo a energia cinética total do manipulador o somatório das parcelas definidas pela equação (3.9), pode-se expressar esse resultado na forma quadrática

$$T = \frac{1}{2} \dot{\mathbf{q}} \mathbf{H}(\mathbf{q}) \dot{\mathbf{q}} \tag{3.10}$$

onde

$$\mathbf{H} = \sum_{i=1}^{n} \left(m_i \mathbf{J}_p^{(i)^{\mathrm{T}}} \mathbf{J}_p^{(i)} + \mathbf{J}_o^{(i)^{\mathrm{T}}} \mathbf{R}_i \mathbf{I}_i^i \mathbf{R}_i^{\mathrm{T}} \mathbf{J}_o^{(i)}\right) \tag{3.11}$$

é a chamada *matriz de inércia do manipulador*. Esta matriz é simétrica e definida positiva.

A energia potencial total armazenada no manipulador é composta pelas parcelas armazenadas nos elos. Tais parcelas, considerando elos rígidos, consistem apenas de esforços gravitacionais, cuja soma resulta em

$$U = -\sum_{i=1}^{n} m_i \mathbf{g}_0^{\mathrm{T}} \mathbf{p}_{ci} \tag{3.12}$$

onde $\mathbf{g}_0$ é a aceleração da gravidade.

Utilizando (3.10) e (3.12) na definição do Lagrangeano, substituindo-o em (3.1) e efetuando suas derivadas, obtém-se as equações do movimento

$$\mathbf{H}(\mathbf{q})\ddot{\mathbf{q}} + \mathbf{C}(\mathbf{q},\dot{\mathbf{q}})\dot{\mathbf{q}} + \mathbf{g}(\mathbf{q}) = \tau \tag{3.13}$$

onde τ é o vetor de torques nas juntas, $\mathbf{g(q)}$ é o vetor de torques gravitacionais, com

$$g_i(\mathbf{q}) = \frac{\partial U}{\partial q_i} \tag{3.14}$$

e

$$\mathbf{C}(\mathbf{q},\dot{\mathbf{q}})\dot{\mathbf{q}} = \dot{\mathbf{H}}(\mathbf{q})\dot{\mathbf{q}} - \frac{1}{2}\left(\frac{\partial}{\partial \mathbf{q}}\dot{\mathbf{q}}^{\mathrm{T}}\mathbf{H}(\mathbf{q})\dot{\mathbf{q}}\right)^{\mathrm{T}} \tag{3.15}$$

é o vetor de torques centrífugos e de Coriolis. Este termo é quadrático nas velocidades das juntas, podendo ter seus elementos expressos como

$$C_i = \sum_{j=1}^{n}\sum_{k=1}^{n} c_{ijk}\dot{q}_k\dot{q}_j \tag{3.16}$$

A escolha dos elementos de $\mathbf{C}(\mathbf{q},\dot{\mathbf{q}})\dot{\mathbf{q}}$ não é única. Em geral, os coeficientes c_{ijk} são definidos pelos *símbolos de Christoffel do primeiro tipo*,

$$c_{ijk} = \frac{1}{2}\left(\frac{\partial H_{ij}}{\partial q_k} + \frac{\partial H_{ik}}{\partial q_j} - \frac{\partial H_{jk}}{\partial q_i}\right) \tag{3.17}$$

onde, por simetria, $c_{ijk} = c_{ikj}$. Esta escolha define uma propriedade das equações do movimento muito útil no projeto de controladores, a ser discutida posteriormente.

Os esforços nas juntas são compostos por

$$\tau = \mathbf{u} - \tau_m - \tau_v + \tau_F \tag{3.18}$$

onde $\mathbf{u}$ é o vetor de torques exercidos pelos atuadores, τ_m corresponde aos torques inerciais devidos aos rotores dos atuadores[24], τ_v é o vetor dos torques causados por atritos nas juntas e τ_F é a parcela de torque causada pelos esforços aplicados ao efetuador final.

[24] Este termo é válido para atuadores elétricos.

Considerando a matriz diagonal $\mathbf{I}_m$ de inércias dos rotores, o torque τ_m é igual a

$$\tau_m = \mathbf{I}_m \ddot{\mathbf{q}} \tag{3.19}$$

O atrito nas juntas é de difícil modelagem. Uma aproximação bastante utilizada[25] é

$$\tau_v = \nu_v \dot{\mathbf{q}} + \nu_s \mathbf{sgn}(\dot{\mathbf{q}}) \tag{3.20}$$

onde ν_s e ν_v são matrizes de coeficientes de atrito estático e viscoso, respectivamente, e **sgn**(.) é uma função *sinal*.

Por fim, os torques causados pelos esforços aplicados no efetuador final são obtidos através do princípio dos trabalhos virtuais. Assim, para o manipulador em equilíbrio, igualam-se os trabalhos virtuais realizados pelos torques nas juntas e pelos esforços no efetuador final,

$$\tau_F{}^T \delta\mathbf{q} = \mathbf{f}^T \delta\mathbf{p} + \mu^T \omega dt \tag{3.21}$$

onde $\delta\mathbf{q}$, $\delta\mathbf{p}$ e ωdt são os deslocamentos virtuais das juntas e do efetuador final, respectivamente[26]. Considerando (2.15), de (3.21) obtém-se

$$\tau_F = \mathbf{J}^T \begin{bmatrix} \mathbf{f} \\ \mu \end{bmatrix} = \mathbf{J}^T \mathbf{F} \tag{3.22}$$

Efetuando as substituições dos termos dos torques em (3.18) e substituindo-a em (3.13), obtém-se o modelo dinâmico no espaço das juntas

$$\mathbf{H}(\mathbf{q})\ddot{\mathbf{q}} + \mathbf{I}_m \ddot{\mathbf{q}} + \mathbf{C}(\mathbf{q}, \dot{\mathbf{q}})\dot{\mathbf{q}} + \nu_v \dot{\mathbf{q}} + \nu_s \mathbf{sgn}(\dot{\mathbf{q}}) + \mathbf{g}(\mathbf{q}) = \mathbf{u} + \mathbf{J}^T \mathbf{F} \tag{3.23}$$

[25] GOMES[17] apresenta um modelo mais preciso, porém de maior complexidade.
[26] δ é o símbolo usual para indicar quantidades virtuais [6].

3.2 FORMULAÇÃO DE NEWTON-EULER

A formulação de Newton-Euler baseia-se no equilíbrio de forças e momentos atuantes nos elos do manipulador, resultando em um conjunto de equações cuja estrutura permite uma solução recursiva. Ao contrário da formulação de Lagrange, esta não explicita os termos das equações de movimento, o que dificulta sua análise. Por outro lado, diversos autores evidenciam a eficiência computacional desta abordagem [2,6,7,14].

Seja o diagrama de corpo livre de um elo genérico mostrado na Figura 3.2. Usando a equação de Newton, obtém-se o balanço de forças para este elo,

$$\mathbf{f}_i^i - \mathbf{f}_{i+1}^i + m_i \mathbf{g}_i^i = m_i \ddot{\mathbf{p}}_{ci}^i \tag{3.24}$$

onde $\mathbf{f}_i^i$ e $\mathbf{f}_{i+1}^i$ são as forças exercidas pelos elos i-1 e i+1 sobre o elo i, respectivamente, $\mathbf{g}_i^i$ é o vetor aceleração da gravidade e $\ddot{\mathbf{p}}_{ci}^i$ é a aceleração do centro de massa do elo. Essas grandezas referem-se ao sistema de coordenadas do elo[27], assim como as demais que aparecem neste desenvolvimento.

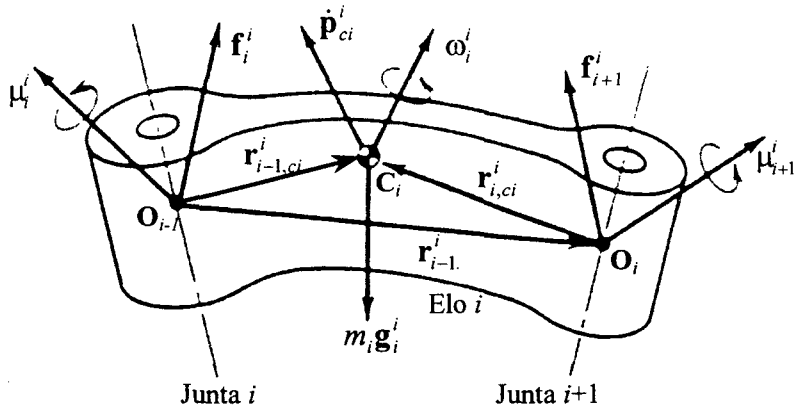


Figura 3.2 Forças e momentos atuantes em um elo genérico i

[27] Várias grandezas envolvidas nas equações tornam-se constantes quando referidas ao sistema do elo, tornando o procedimento recursivo mais eficiente pela simplificação das equações [1,2].

Para a obtenção do equilíbrio dos momentos, utiliza-se a equação de Euler,

$$\mu_i^i - \mu_{i+1}^i + \mathbf{f}_i^i \times \mathbf{r}_{i-1,ci}^i - \mathbf{f}_{i+1}^i \times \mathbf{r}_{i,ci}^i = \mathbf{I}_i^i \dot{\omega}_i^i + \omega_i^i \times (\mathbf{I}_i^i \omega_i^i) \tag{3.25}$$

onde μ_i^i e μ_{i+1}^i são os torques exercidos pelos elos i-1 e i+1 sobre o elo i, respectivamente, ω_i^i é a velocidade angular, $\dot{\omega}_i^i$ é a aceleração angular, $\mathbf{r}_{i-1,ci}^i$ é o vetor da junta i ao centro de massa e $\mathbf{r}_{i,ci}^i$ é o vetor da junta i+1 ao centro de massa.

A parte direita da igualdade em (3.25) é o momento inercial do elo, dado pela taxa de variação do momento angular deste ao longo do tempo. O primeiro termo corresponde à variação da velocidade angular do elo, enquanto o segundo representa o *torque giroscópico* induzido no elo devido a mudança de orientação deste em relação a um sistema inercial[28].

Para expressar as equações de equilíbrio em função das variáveis das juntas, deve-se determinar as relações destas com as velocidades e acelerações dos elos:

$$\omega_i^i = \begin{cases} \mathbf{R}_i^{i-1\mathrm{T}} \omega_{i-1}^{i-1} \\ \mathbf{R}_i^{i-1\mathrm{T}} \left(\omega_{i-1}^{i-1} + \dot{q}_i \mathbf{z}_0 \right) \end{cases} \tag{3.26}$$

$$\dot{\omega}_i^i = \begin{cases} \mathbf{R}_i^{i-1\mathrm{T}} \dot{\omega}_{i-1}^{i-1} \\ \mathbf{R}_i^{i-1\mathrm{T}} \left(\dot{\omega}_{i-1}^{i-1} + \ddot{q}_i \mathbf{z}_0 + \omega_{i-1}^{i-1} \times \dot{q}_i \mathbf{z}_0 \right) \end{cases} \tag{3.27}$$

$$\ddot{\mathbf{p}}_i^i = \begin{cases} \mathbf{R}_i^{i-1\mathrm{T}} \left(\ddot{\mathbf{p}}_{i-1}^{i-1} + \ddot{q}_i \mathbf{z}_0 \right) + 2\omega_i^i \times \dot{q}_i \mathbf{R}_i^{i-1\mathrm{T}} \mathbf{z}_0 + \dot{\omega}_i^i \times \mathbf{r}_{i-1,i}^i + \omega_i^i \times \left(\omega_i^i \times \mathbf{r}_{i-1,i}^i \right) \\ \mathbf{R}_i^{i-1\mathrm{T}} \ddot{\mathbf{p}}_{i-1}^{i-1} + \dot{\omega}_i^i \times \mathbf{r}_{i-1,i}^i + \omega_i^i \times \left(\omega_i^i \times \mathbf{r}_{i-1,i}^i \right) \end{cases} \tag{3.28}$$

$$\ddot{\mathbf{p}}_{ci}^i = \ddot{\mathbf{p}}_i^i + \dot{\omega}_i^i \times \mathbf{r}_{i,ci}^i + \omega_i^i \times (\omega_i^i \times \mathbf{r}_{i,ci}^i) \tag{3.29}$$

$$\dot{\omega}_{mi}^{i-1} = \dot{\omega}_{i-1}^{i-1} + \ddot{q}_i k_{ri} \mathbf{z}_{mi}^{i-1} + \dot{q}_i k_{ri} \omega_{i-1}^{i-1} \times \mathbf{z}_{mi}^{i-1} \tag{3.30}$$

onde $\mathbf{z}_0=[0\ 0\ 1]^{\mathrm{T}}$.

Pelas equações acima observa-se a natureza recursiva da formulação de Newton-Euler, pois as velocidades e acelerações dos elos dependem das velocidades e acelerações

[28] Em relação a um sistema inercial, o tensor de inércia do elo varia de acordo com a variação da orientação deste. A dedução da parte direita de (3.25) é apresentada detalhadamente em SPONG e VIDYASAGAR[1].

dos elos anteriores. Além disso, a recursividade também está presente nas equações de equilíbrio dos elos, pois estas dependem das forças e momentos dos elos seguintes.

A solução das equações de Newton-Euler consiste em duas etapas. Na primeira determinam-se as velocidades e acelerações dos elos 1...*n*, a partir da velocidade e da aceleração da base (condições iniciais). Na segunda determinam-se os esforços nos elos e, destes, os torques nas juntas, a partir das condições terminais (esforços no efetuador final).

Aproveitando a recursividade, uma simplificação interessante consiste em definir a aceleração linear da base como $\ddot{\mathbf{p}}_0 - \mathbf{g}_0$, incorporando a aceleração da gravidade nos cálculos das acelerações dos centros de massa e reduzindo em um termo a equação do equilíbrio das forças. Assim, as equações (3.24) e (3.25) podem ser reescritas para explicitar $\mathbf{f}_i^i$ e μ_i^i como

$$\mathbf{f}_i^i = \mathbf{R}_{i+1}^i \mathbf{f}_{i+1}^{i+1} + m_i \ddot{\mathbf{p}}_{ci}^i \tag{3.31}$$

$$\begin{aligned}\mu_i^i = {} & \mathbf{R}_{i+1}^i \mu_{i+1}^{i+1} + \mathbf{R}_{i+1}^i \mathbf{f}_{i+1}^{i+1} \times \mathbf{r}_{i,ci}^i - \mathbf{f}_i^i \times \mathbf{r}_{i-1,ci}^i + \mathbf{I}_i^i \dot{\omega}_i^i + \omega_i^i \times (\mathbf{I}_i^i \omega_i^i) + \\ & + \ddot{q}_{i+1} k_{ri+1} I_{mi+1} \mathbf{z}_{mi+1}^i + \dot{q}_{i+1} k_{ri+1} I_{mi+1} \omega_i^i \times \mathbf{z}_{mi+1}^i\end{aligned} \tag{3.32}$$

Os torques nas juntas, por fim, são determinados pelas projeções dos esforços calculados sobre os eixos das juntas, além das contribuições do atrito nas juntas e dos efeitos das inércias dos rotores:

$$\tau_i = \begin{cases} \mathbf{f}_i^{i\mathrm{T}} \mathbf{R}_i^{i-1\mathrm{T}} \mathbf{z}_0 + v_{vi} \dot{q}_i + v_{si} \operatorname{sgn}(\dot{q}_i) + k_{ri} \dot{\omega}_{mi}^{i-1\mathrm{T}} z_{mi}^{i-1} I_{mi} & \text{, para juntas prismaticas} \\ \mu_i^{i\mathrm{T}} \mathbf{R}_i^{i-1\mathrm{T}} \mathbf{z}_0 + v_{vi} \dot{q}_i + v_{si} \operatorname{sgn}(\dot{q}_i) + k_{ri} \dot{\omega}_{mi}^{i-1\mathrm{T}} z_{mi}^{i-1} I_{mi} & \text{, para juntas rotacionais} \end{cases} \tag{3.33}$$

A Figura 3.3, mostrada a seguir, esquematiza a solução para a formulação de Newton-Euler.

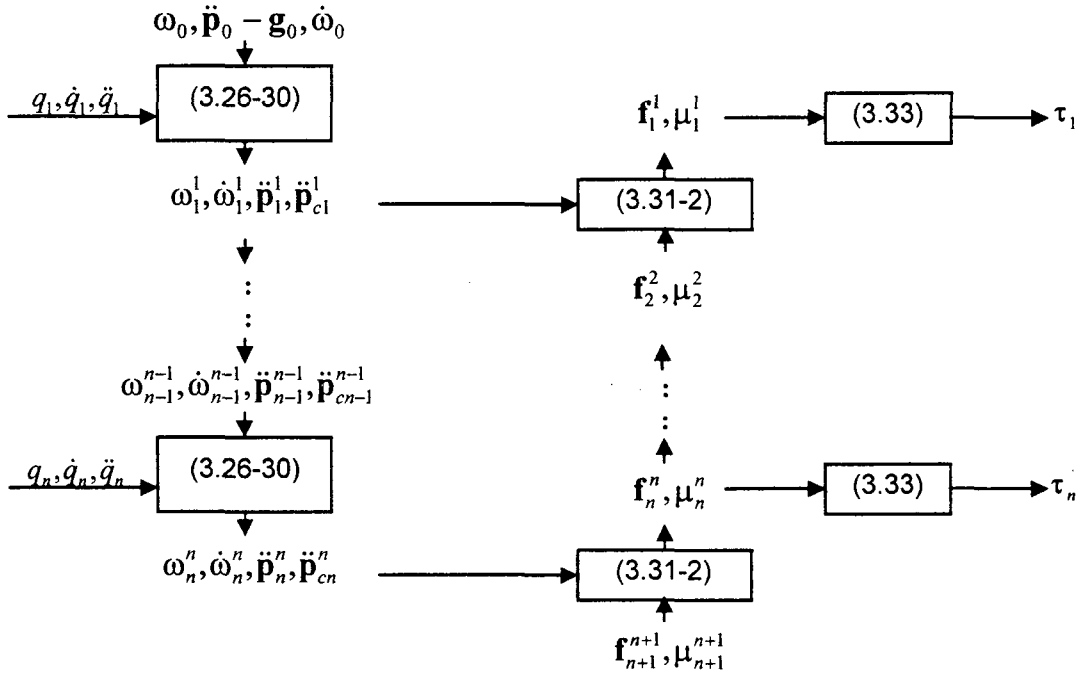


Figura 3.3 Esquema da solução recursiva das equações de Newton-Euler

3.3 IMPLEMENTAÇÃO DE MODELOS DINÂMICOS

Independente da formulação utilizada em sua obtenção, o modelo dinâmico pode ser empregado tanto na determinação da *dinâmica direta* quanto da *dinâmica inversa* do manipulador.

Usada em simulação, a dinâmica direta consiste na determinação das posições, velocidades e acelerações das juntas a partir dos torques exercidos pelos atuadores e dos esforços exercidos no efetuador final. As acelerações nas juntas são obtidas de (3.23):

$$\ddot{\mathbf{q}} = \mathbf{H}^{-1}(\mathbf{q})\big(\mathbf{u} - \mathbf{D}(\mathbf{q}, \dot{\mathbf{q}})\big) \tag{3.34}$$

$$\mathbf{D}(\mathbf{q}, \dot{\mathbf{q}}) = \mathbf{C}(\mathbf{q}, \dot{\mathbf{q}})\dot{\mathbf{q}} + \mathbf{g}(\mathbf{q}) + \nu_v \dot{\mathbf{q}} + \nu_s \mathbf{sgn}(\dot{\mathbf{q}}) - \mathbf{J}^{\mathrm{T}}\mathbf{F} \tag{3.35}$$

desconsiderando-se, por simplificação, os torques de inércia dos rotores. Esta equação, no espaço de estados,

$$\begin{bmatrix} \dot{\mathbf{q}} \\ \ddot{\mathbf{q}} \end{bmatrix} = \begin{bmatrix} \dot{\mathbf{q}} \\ -\mathbf{H}^{-1}(\mathbf{q})\mathbf{D}(\mathbf{q},\dot{\mathbf{q}}) \end{bmatrix} + \begin{bmatrix} \mathbf{0} \\ \mathbf{H}^{-1}(\mathbf{q})\mathbf{u} \end{bmatrix} \tag{3.36}$$

forma um sistema de $2n$ equações diferenciais ordinárias não lineares, passível de solução por algum método numérico, conhecido seu estado inicial ($\mathbf{q}(t_0)$ e $\dot{\mathbf{q}}(t_0)$), para $t > t_0$ [29][2].

Os termos de (3.34) também podem ser podem ser determinados pela formulação de Newton-Euler. $\mathbf{D}(\mathbf{q},\dot{\mathbf{q}})$ é obtido calculando $\mathbf{u}$ considerando $\ddot{\mathbf{q}} = \mathbf{0}$, enquanto as colunas da matriz de inércia $\mathbf{H}(\mathbf{q})$ são determinadas calculando $\mathbf{u}$ considerando $\mathbf{g}_0$=0, $\dot{\mathbf{q}} = \mathbf{0}$, $\ddot{q}_i = 1$ e $\ddot{q}_j = 0$, para $i \neq j$ [2,7].

A dinâmica inversa é útil em algoritmos de controle. Nestes, interessa a determinação dos torques nas juntas necessários para gerar o movimento especificado pelas posições, velocidades e acelerações das juntas, conhecidos os esforços aplicados no efetuador final. Uma vez que o modelo dinâmico fornece naturalmente a relação de posições, velocidades e acelerações para torques, a solução do modelo dinâmico resulta na dinâmica inversa.

É sabido que a formulação de Newton-Euler é mais eficiente que a formulação Lagrangeana, tanto para a dinâmica direta quanto para a dinâmica inversa, quando se avalia a carga computacional e a velocidade de resposta[30] [14]. Para algoritmos de controle que utilizam o cálculo da dinâmica inversa em tempo real, acrescenta-se a necessidade de otimização das equações do movimento, através de simplificações devidas a parâmetros nulos, do uso de variáveis intermediárias para termos que aparecem repetidamente nas equações e do pré-cálculo de termos independentes das variáveis das juntas [7].

[29] Maiores detalhes sobre métodos de solução de equações diferenciais ordinárias e sua implementação podem ser vistos em GOMES[18] e CHAPRA e CANALE[19].

[30] Segundo SCIAVICCO e SICILIANO[2], é possível demonstrar que a formulação Lagrangeana resulta em algoritmos computacionalmente eficientes, a partir de um esforço de reformulação no modelo dinâmico.

3.4 CONCLUSÃO

Neste capítulo foram apresentadas as duas formulações empregadas na determinação de modelos dinâmicos de manipuladores. A seguir, foram analisados os aspectos envolvidos na implementação das equações de movimento tanto para a obtenção da dinâmica direta quanto da dinâmica inversa, buscando a melhor performance computacional.

4. DESENVOLVIMENTO DO SIMULADOR DE ROBÔS MANIPULADORES

Com base nas características cinemáticas e dinâmicas dos manipuladores apresentadas nos capítulos anteriores, é possível definir o sistema a ser simulado.

Neste capítulo, descreve-se o desenvolvimento do *software* de simulação. Inicia-se com a análise e especificação do problema de projeto. A seguir, apresentam-se o paradigma da orientação a objetos, utilizado no desenvolvimento, e as ferramentas empregadas para a implementação. A partir da visão geral do simulador, são detalhados os componentes do sistema, implementados como classes de objetos. A análise dos resultados exige traçadores de gráficos e visualizadores de ambientes virtuais 3D, cuja implementação e uso são mostrados. Por fim, apresenta-se a interface com o usuário.

4.1 ANÁLISE E ESPECIFICAÇÃO DO PROBLEMA DE PROJETO

Na análise e especificação de um sistema, devem ser definidos tanto o domínio quanto os limites deste, bem como os requisitos necessários para a solução do problema de projeto. No desenvolvimento do simulador de robôs manipuladores para a execução de tarefas, pode-se representar de forma genérica os elementos do sistema e seus relacionamentos através do esquema mostrado na Figura 4.1.

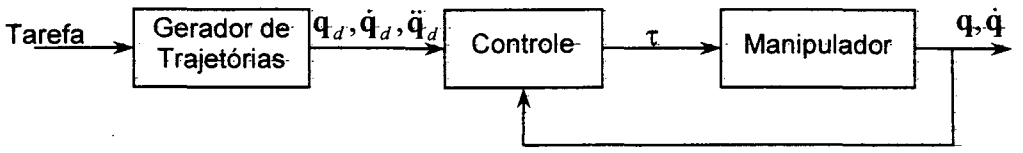


Figura 4.1 Esquema geral de um robô manipulador com controle por realimentação

Neste esquema, são identificados como componentes básicos do sistema o *manipulador*, o *controlador*, a *tarefa* a ser executada e o *gerador de trajetórias*. Este último é empregado para a decomposição da tarefa em movimentos suaves ao longo do tempo de execução, dependendo da especificação desta. Pode-se ainda desprezar o controlador, se for desejado analisar apenas a cinemática do manipulador e os torques resultantes da execução de uma tarefa.

As *tarefas* executadas por um robô industrial podem ser divididas em tarefas nas quais há contato com o meio e tarefas nas quais não há contato com o meio. Estas ainda podem ser subdivididas em tarefas de regulação (ponto a ponto) ou seguimento de trajetória (contínua), sendo especificadas tanto no espaço das juntas como no espaço operacional[7,20].

Nas tarefas em que há contato com o meio, uma série de problemas devem ser considerados, como a modelagem do meio, a dinâmica do contato, e a especificação das forças no efetuador final, alguns dos quais ainda em aberto[20,21]. Neste trabalho, as tarefas que envolvem contato com o meio não são implementadas.

A especificação de tarefas de posição costuma ser feita através dos pontos pelos quais o efetuador final do manipulador deve passar. Para que o manipulador realize a tarefa proposta sem que esforços excessivos nos atuadores sejam alcançados nem sejam excitados modos de ressonância da estrutura do manipulador, a transição entre os pontos deve ser feita de forma suave. A função dos *geradores de trajetória* é gerar referências de movimento ao longo do tempo a partir das configurações que o manipulador deve alcançar, assegurando a execução da tarefa através de um movimento aceitável pela estrutura do manipulador[2].

Um gerador de trajetórias deve, então, receber como entradas os pontos do caminho a ser percorrido, gerando como saída uma seqüência temporal de pontos, interpolados

através de funções polinomiais ou aproximações destas. Em trajetórias contínuas, a velocidade e a aceleração desejada em cada ponto podem ser necessárias.

Na modelagem de um *manipulador*, devem ser levadas em conta as características estruturais, cinemáticas e dinâmicas do mesmo. Além disso, o modelo deve ser o mais parametrizável possível, a fim de possibilitar a representação de diferentes estruturas de manipuladores sem modificações no código. Para possibilitar uma visualização do manipulador, deve-se ainda levar em conta seu modelo físico.

Em relação à cinemática, problemas podem aparecer na inversão, que não possui uma solução analítica geral. Nesse caso, métodos numéricos podem resolver o problema, embora com um custo na performance. Além disso, é importante adotar uma representação mínima comum para especificar posição e orientação do efetuador final.

A dinâmica do manipulador é fundamental para definir a relação entre os torques aplicados e o movimento resultante. Para uma modelagem fiel, podem ser considerados tanto o atrito nas juntas quanto a dinâmica dos atuadores. Embora a dinâmica direta seja de importância para a simulação, a dinâmica inversa pode ser interessante de se obter para uso em controladores. Para ser o mais genérico possível sem necessitar alteração de código, a dinâmica pode ser determinada através do método de Newton-Euler.

Como requisito final da modelagem de um manipulador, devem ser levados em conta os limites de posição, velocidade e aceleração nas juntas, bem como os limites de torque dos atuadores.

Na modelagem de *controladores*, deve-se observar as características comuns entre os diversos tipos. Em uma visão mais geral, o controlador determina a ação de controle (torques dos atuadores) a partir de uma referência e dos valores reais das grandezas do manipulador (posição, velocidade e aceleração). A definição de ganhos dos controladores é facilmente parametrizável. Seus algoritmos, porém, dificilmente o serão, devendo ser descritos em alguma linguagem. Tal implementação, porém, não deve implicar na alteração de código do simulador. Assim, deve-se considerar a possibilidade de implementar os controladores em módulos externos ao simulador, provendo um mecanismo de *acoplamento* destes módulos à simulação.

A visualização dos resultados das simulações é um requisito fundamental para o sistema. Para tanto, devem estar disponíveis gráficos das variáveis de estado do manipulador ao longo do tempo, além de outras grandezas como erros e trajetória no espaço operacional. Além dos gráficos, é interessante fazer uso de animações, a fim de dar ao usuário uma noção mais realista da execução da tarefa. Para tanto, deve-se contar com recursos gráficos tridimensionais que possibilitem a construção de um ambiente virtual onde um modelo do manipulador possa executar a tarefa com realismo.

Para atender a todos esses requisitos do sistema, é necessária uma base de entidades e ferramentas matemáticas, como matrizes, vetores e métodos de solução de equações diferenciais. Dependendo da linguagem e do ambiente utilizado na implementação, essa base está disponível na forma de módulos de código fonte ou bibliotecas já implementadas. Porém, a maioria das ferramentas utilizadas no desenvolvimento de aplicações[31] não provê tais facilidades, sendo necessária a sua implementação.

Os diversos elementos envolvidos na simulação, além das bases matemáticas necessárias, podem ser criados de forma a serem independentes entre si, permitindo a reutilização dos mesmos em outras aplicações, além de facilitar o desenvolvimento independente dos módulos e sua posterior manutenção e atualização. Segundo RUMBAUGH et al.[10], a simulação dinâmica é um dos casos mais simples de sistemas a se projetar utilizando a abordagem baseada em objetos, pois os componentes do sistema podem ser modelados quase diretamente através de objetos. Por esses motivos, esse paradigma foi adotado como metodologia de trabalho, sendo apresentado a seguir.

4.1.1 METODOLOGIA DE DESENVOLVIMENTO

O processo de desenvolvimento de *software* é composto por várias etapas, como a análise do problema de projeto, a determinação e especificação da solução, sua implementação, testes e implantação. A crescente complexidade dos sistemas e a velocidade com que o *software* deve ser produzido motiva a criação e adoção de novas metodologias e ferramentas de desenvolvimento de aplicações[22,23,24].

[31] Considerando aquelas ferramentas que geram código executável que não necessite da ferramenta de desenvolvimento. Esse não é o caso do MATLAB, por exemplo, que contém a base matemática necessária.

O paradigma da **orientação a objetos** é uma tendência para o desenvolvimento de *software*, tendo provado seu valor para a construção de sistemas em todos os tipos de domínios dos problemas, independente de tamanho ou complexidade[23]. Segundo este paradigma, o desenvolvimento é feito de forma a tratar o *software* como uma abstração do mundo real, onde a essência do método é a identificação e organização de conceitos do domínio da aplicação, trazendo como principal benefício a facilidade de expressão e comunicação entre desenvolvedores e clientes[10].

Segundo esta filosofia, *objetos* são os blocos de construção das aplicações, correspondendo a módulos auto-suficientes[32] formados tanto por dados quanto pelo código que os manipulam[25]. Os atributos e comportamentos pertinentes aos objetos são definidos formalmente em classes, geralmente modeladas a partir do vocabulário do espaço do problema ou do espaço da solução, representando o estado, comportamento e identidade de elementos do domínio da aplicação[23,24]. Assim, todo objeto sempre é criado com base em uma classe.

Um conceito fundamental na programação orientada a objetos é a *herança*, que consiste na transmissão de características através de uma hierarquia de classes. As classes nos níveis mais baixos de uma hierarquia (denominadas classes derivadas) recebem todas as características das classes de níveis superiores (chamadas de classes base), o que permite economia no processo de desenvolvimento, pois apenas as modificações necessárias nas características já herdadas (acrescentando novos dados e métodos ou modificando métodos já existente) são necessárias[25].

Outro conceito importante é o de *polimorfismo*, que consiste na possibilidade de um método associado a um objeto executar de forma diferente, dependendo da classe de onde o objeto foi instanciado, sendo bastante comum em hierarquias de classes[33][10,23,25].

WIENER e PINSON[26] resumem as características e conceitos apresentados afirmando que *objetos são instâncias de classes que respondem a mensagens de acordo com métodos determinados nas descrições de classes*. A Tabela 4.1 estabelece uma

[32] A característica citada é conhecida por *encapsulamento*.

[33] Um exemplo de polimorfismo através de hierarquia de classes de elementos de desenho é mostrado em RUMBAUGH et al.[10], onde são modelados objetos de desenho com um mesmo método Exibir, cuja execução é diferente de acordo com a natureza do objeto (linha, círculo, polígono).

comparação entre a programação orientada a objetos e a programação procedural.

Tabela 4.1 Comparação entre programação orientada a objetos e programação estruturada

Programação Orientada a Objetos	Programação Estruturada e Modular
Classe	Tipo de dado estruturado
Instância	Variável
Instanciar	Criar uma variável
Atributo	Dado
Método	Procedimento ou função
Objeto	Variáveis+Procedimentos e Funções

Embora opositores afirmem que as vantagens da programação orientada a objetos podem ser obtidas através da programação estruturada e modular tradicional, deve-se observar que aquela é uma forma evoluída desta, com regras mais formais que favorecem características como *reusabilidade*, *intercambiabilidade*, *padronização* e a adoção de uma *arquitetura evolucionária de software*, aumentando a qualidade e produtividade no desenvolvimento de *software*[25,27]. Além disso, deve-se observar que o paradigma da orientação a objetos vai além da programação, inserindo-se em todas as fases do desenvolvimento de *software*, em particular na fase de análise e especificação do sistema[10,23,25].

4.1.2 Ferramentas de Desenvolvimento

Uma vez definida a plataforma sobre a qual o *software* irá rodar, pode-se passar à avaliação e escolha da linguagem de programação e da ferramenta mais adequada para o desenvolvimento do simulador.

Devido a características como flexibilidade, velocidade de execução do código gerado, padronização e portabilidade, além de suporte a classes e objetos, a linguagem C++ foi escolhida. Esta é uma versão orientada a objetos da linguagem C, que se tornou popular por adicionar a capacidade de programação orientada a objetos às características originais da linguagem, além de corrigir certas fragilidades da linguagem que a originou, como a fraca verificação de tipos [10,25,27]. Ressalta-se, ainda, a existência de muitas bibliotecas

de funções, tipos e classes existentes para essa linguagem, além do farto material bibliográfico disponível.

Dentre os diversos ambientes de desenvolvimento C++ voltados para Windows existentes no mercado, o que mais se destacou no rápido desenvolvimento de aplicações visuais foi o Borland C++ Builder. Esta ferramenta consiste em um ambiente de desenvolvimento visual bidirecional[34], além de contar com uma biblioteca de classes que ocultam as dificuldades inerentes à programação de elementos visuais no Windows (VCL - Visual Component Library), que pode ser expandida de acordo com as necessidades do usuário, seja por desenvolvimento próprio, seja por aquisição de classes desenvolvidas por terceiros. O Borland C++ Builder possui um compilador 32-bits com otimizador de código e ferramentas que auxiliam na depuração, como execução passo a passo, observação do conteúdo da memória e do estado da CPU integrados ao editor de código, contando também com vasta documentação *on-line*[28].

Uma das características desejadas para o simulador é a visualização 3D do robô em execução de tarefas e do ambiente em que ele se encontra. Para atender a esse requisito, uma das possíveis soluções consiste no desenvolvimento de uma biblioteca de visualização 3D. Outra possibilidade é a utilização de uma ferramenta ou biblioteca existente. Apesar de já haver uma implementação de um conjunto de rotinas 3D para simulação de robôs[9], verificou-se que a performance das mesmas era baixa em relação às novas necessidades, além de existirem pequenas falhas nas rotinas de renderização das cenas e ausência de elementos que trouxessem realidade, como iluminação, por exemplo. Haveria necessidade de uma revisão geral das rotinas, que demandaria tempo, além de não haverem garantias de uma boa performance.

Optou-se, então, pela utilização do OpenGL, que consiste em uma biblioteca gráfica independente de plataforma de *hardware* e *software* projetada essencialmente para ser portável e de rápida execução[29]. Originalmente desenvolvida pela SGI[35], essa biblioteca e suas derivadas estão se tornando rapidamente um padrão da indústria de *software* e *hardware*, existindo para a maioria dos sistemas Unix utilizados em

[34] Este tipo de ambiente permite "desenhar" a parte de interface da aplicação, enquanto o código necessário para a interface desenhada é gerado. O termo bidirecional refere-se ao fato da interface mudar automaticamente quando o código é alterado e vice-versa.
[35] Empresa fabricante de *workstations* e servidores. Antigamente conhecida como Silicon Graphics.

workstations de alta performance em aplicações gráficas, além de versões para microcomputadores rodando Windows, Linux ou MacOS[25,30].

Embora uma placa gráfica aceleradora 3D seja desejável para alta performance, o OpenGL executa com boa velocidade no *hardware* gráfico comumente encontrado nos microcomputadores PC, estando disponível desde a versão Windows 95 OSR2[30].

Entre as características gráficas desta biblioteca, destacam-se a texturização de superfícies, a remoção de áreas ocultas, os efeitos de iluminação, a tonalização suave das cores em função da incidência de luz e do material das superfícies, a transparência de objetos e o uso de matrizes de transformação para modificar objetos no espaço 3D[29].

4.2 VISÃO GERAL DO *SOFTWARE*

A Figura 4.2 corresponde a uma visão geral da arquitetura do simulador, representado segundo a UML (Unified Modeling Language), uma linguagem gráfica que será utilizada para ilustrar a modelagem do sistema ao longo do texto[36].

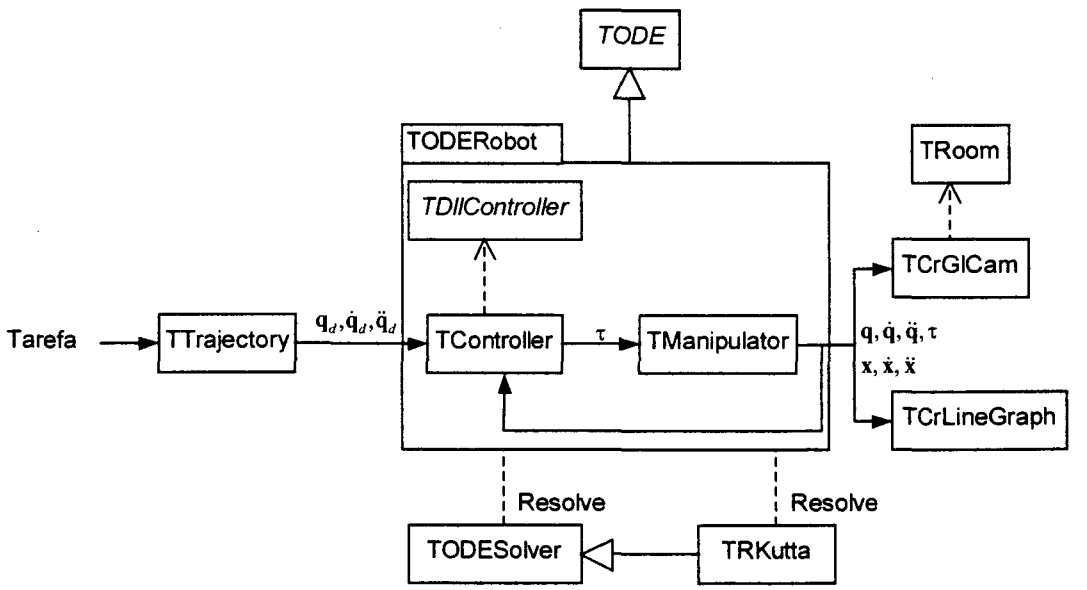


Figura 4.2 Visão geral da arquitetura do simulador

[36] Para uma referência sobre esta linguagem de modelagem, consultar BOOCH, RUMBAUGH e JACOBSON[23].

A entrada de dados é a tarefa a ser executada. Esta pode ser uma trajetória no espaço das juntas, formada por valores de posição, velocidade e aceleração destas ao longo do tempo. Os pontos são separados por um intervalo fixo de tempo, sendo armazenados em uma instância da classe `TTask`.

Outra forma de especificar uma tarefa é pela definição de um caminho no espaço das juntas ou no espaço operacional por valores de posição, velocidade e aceleração armazenados em um objeto `TTask`. Nesse caso, torna-se necessário o uso de um gerador de trajetórias, para interpolar uma trajetória suave entre os pontos do caminho. Se este for definido no espaço operacional, os pontos ainda devem ser transformados em posições das juntas correspondentes através de um algoritmo de inversão cinemática, que é implementado na classe `TManipulator`. Os geradores de trajetória ponto a ponto e contínua são modelados em uma hierarquia de classes cuja raiz é a classe `TTrajectory`.

O sistema é constituído por um manipulador acionado por um controlador em malha fechada. Este é modelado pela classe `TController`, que faz a interface com o algoritmo de controle implementado pelo usuário em uma biblioteca de ligação dinâmica (dll). Esta consiste em um método de criação de uma instância de uma classe derivada da classe abstrata `TDLLController`, que define o comportamento geral que um algoritmo de controle por realimentação deve ter. O modelo do manipulador é feito pela classe `TManipulator`. O resultado da simulação consiste nos valores obtidos para posição, velocidade, aceleração e torques das juntas ao longo da execução da tarefa. Para visualização dos resultados, pode-se obter os valores equivalentes no espaço operacional, através dos métodos definidos em `TManipulator`.

O sistema pode ser formado apenas por uma instância de `TManipulator`. Neste caso, a simulação determina os torques gerados pela execução da trajetória desejada, considerando que esta é perfeitamente seguida.

Para modelar o sistema, deve-se criar uma classe derivada da classe abstrata `TODE`, que define o comportamento geral de um sistema de equações diferenciais ordinárias. A solução desses sistemas pode ser feita por uma instância de `TODESolver` ou por uma instância de `TRKutta`. A primeira classe implementa o método de Euler, enquanto a

segunda, derivada de `TODESolver`, implementa um algoritmo genérico do método de Runge-Kutta, cuja ordem pode ser definida de acordo com a precisão desejada na resposta.

O resultado das simulações é visualizado em gráficos das variáveis das juntas e do espaço operacional, mostrados em janelas definidas pela classe `TFrmGraph`. Os gráficos são traçados por componentes da classe `TCrLineGraph`, criada para o simulador.

Para visualização tridimensional, foi desenvolvida a classe de componentes `TCRGlCam`, baseada na biblioteca OpenGL. Esta implementa uma câmera sintética, que mostra um ambiente 3D do ponto de vista de um observador posicionado neste espaço. Pode-se utilizar esta câmera para traçar a trajetória executada no espaço operacional, ou para mostrar uma animação de um modelo virtual do manipulador executando a tarefa especificada. Para modelar um ambiente virtual onde o manipulador está localizado, foi criada a classe `TRoom`, que representa uma sala retangular. O corpo do manipulador deve ser formado por sólidos geométricos, representados por instâncias de `TModel3D`. Os sólidos que formam cada elo são relacionados em `TManipulator`.

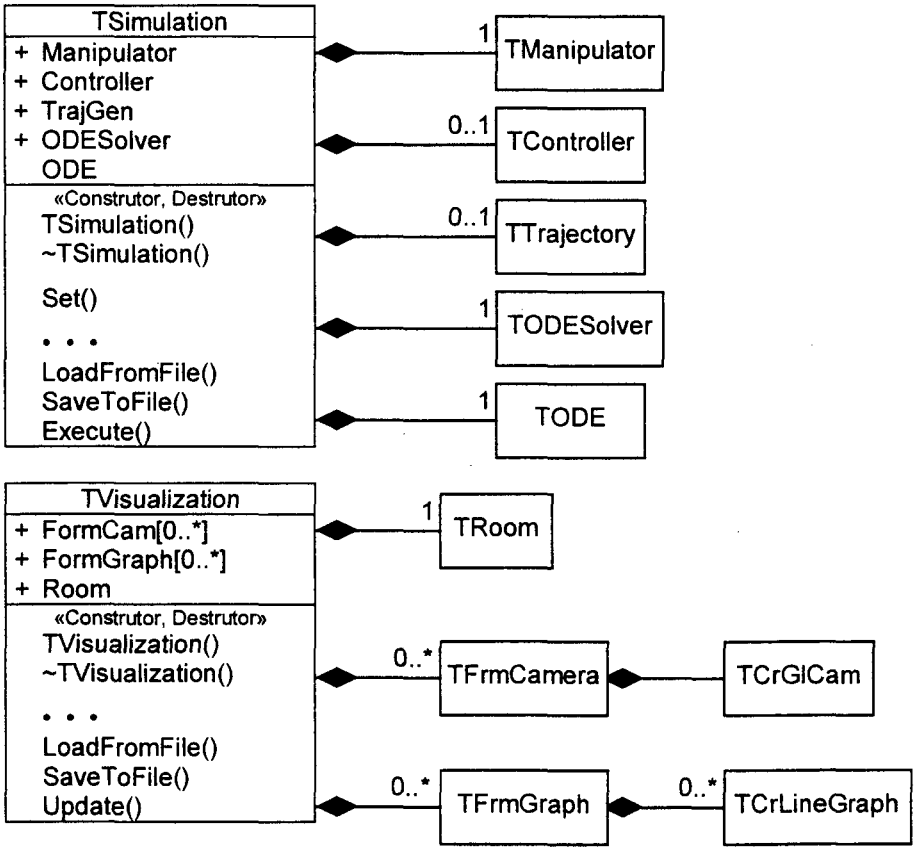


Figura 4.3 Modelo das classes `TSimulation` e `TVisualization`

As classes que modelam os elementos do sistema simulado compõem a classe `TSimulation`, que relaciona os objetos entre si e armazena os seus parâmetros. Os elementos de visualização são agregados em `TVisualization`. A Figura 4.3 ilustra essa modelagem.

4.3 ENTIDADES MATEMÁTICAS BÁSICAS

Na modelagem dos componentes do simulador, matrizes e vetores são entidades fundamentais. As matrizes aparecem na representação dos sistemas de coordenadas dos elos e do efetuador final do manipulador, nas equações dinâmicas destes e na definição dos ganhos dos controladores. Para representar posições, velocidades, acelerações, forças e momentos, usadas particularmente na formulação de Newton-Euler, são utilizados vetores. Estes também são empregados na representação da forma do manipulador, feita através de poliedros, e para a determinação da posição e da orientação da câmera sintética no ambiente virtual da simulação.

A simulação dinâmica exige a solução dos modelos matemáticos do sistema, que no caso dos manipuladores consistem em equações diferenciais ordinárias. A obtenção da cinemática inversa a partir da cinemática diferencial também exige o mesmo tipo de solução. Para fins computacionais, os métodos numéricos são mais empregados, pois muitas vezes os modelos apresentam comportamento não linear, podendo não existir solução analítica[18]. Entre os métodos mais empregados, destacam-se o de Euler, por sua simplicidade, e o de Runge-Kutta.

Embora na linguagem C++ exista o conceito de matrizes, estas carecem da flexibilidade e das operações apresentadas em outros *software* como o MATLAB. Além disso, é necessário implementar métodos numéricos para a solução dos modelos matemáticos. Estes elementos estão modelados na biblioteca de classes `CrExtMth`, desenvolvida para atender às necessidades do simulador, como mostrado na Figura 4.4. Essas classes são apresentadas a seguir.

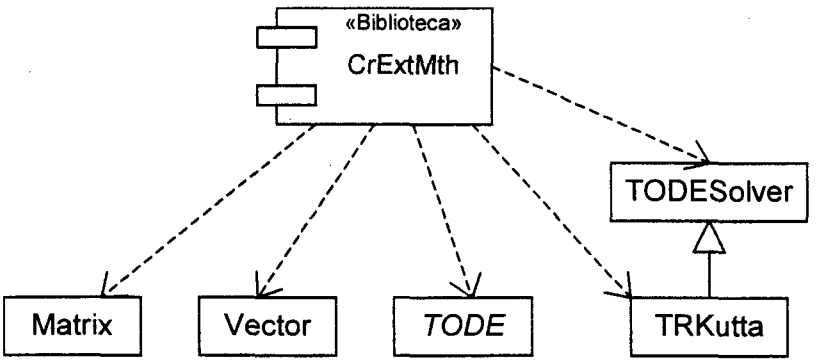


Figura 4.4 Biblioteca `CrExtMth` e suas classes

4.3.1 CLASSE MATRIX

A classe `Matrix` representa matrizes de números reais e implementa os métodos e operações básicas sobre estas. Ao contrário da definição de matriz nativa da linguagem C++ e outras, instâncias de `Matrix` podem variar suas dimensões de acordo com o resultado de expressões envolvendo matrizes ou conforme as definições do usuário.

Os elementos das matrizes podem ser acessados individualmente, através do operador `(l,c)`, onde `l` e `c` correspondem a linha e a coluna do elemento, respectivamente. Pedaços da matriz podem ser acessados através do operador `(l,c,nl,nc)` e do método `Set(l,c)`[37].

Para dar um aspecto mais natural às expressões com matrizes, instâncias de `Matrix` podem ser acessadas como um todo. Para tanto, foram redefinidos os operadores de atribuição (=), comparação (== e !=) e aritméticos (+,-, *, /). Estes podem ser utilizados tanto para aritmética entre matrizes como para aritmética entre escalares e matrizes. A operação de divisão (/) corresponde à multiplicação da primeira matriz com a inversa da segunda matriz.

Também estão implementadas operações específicas de matrizes, como traço (`Trc()`), norma (`Norm()`), determinante (`Det()`), inversa (`Inv()`) e pseudo-inversa (`PInv()`). Para saber se uma matriz apresenta uma característica particular, estão implementados métodos como `IsSingular()` (matriz singular), `IsOrthogonal()` (matriz ortogonal) e `IsSymmetric()` (matriz simétrica). A Figura 4.5 mostra a modelagem desta classe.

4.3.2 CLASSE VECTOR

Vetores são representados pela classe `Vector`. Estes são formados pelas componentes nas direções dos eixos coordenados `X`, `Y` e `Z`. Além de representar grandezas vetoriais, objetos da classe `Vector` podem representar pontos no espaço 3D, se considerar o vetor com origem na origem do sistema de coordenadas de referência. Os componentes de uma instância podem ser acessados através do operador `(i)`.

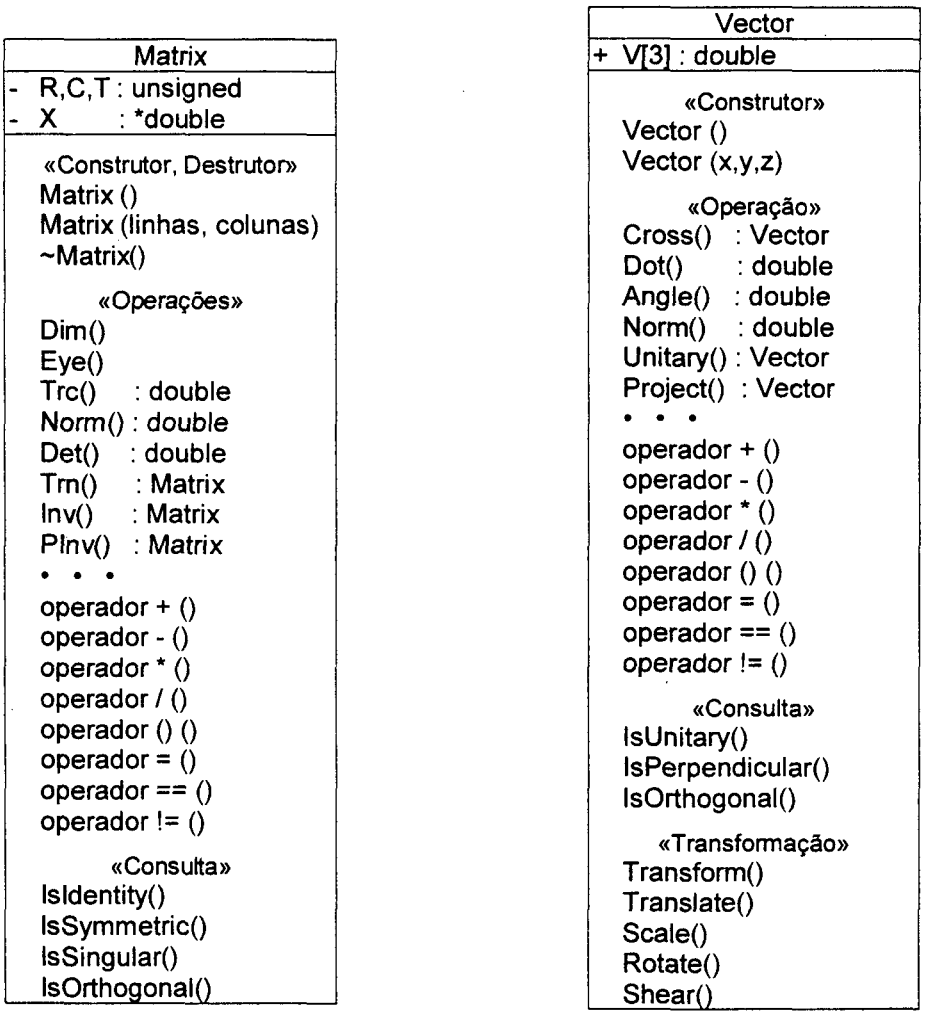


Figura 4.5 Modelos das Classes `Matrix` e `Vector`

Os operadores aritméticos (+,-, *, /) são redefinidos para expressões envolvendo vetores e escalares, bem como operadores de comparação (== e !=) e atribuição (=). Estão

[37] O primeiro elemento da matriz encontra-se na posição (0,0).

implementados métodos específicos para vetores, como produto escalar (`Dot()`), produto vetorial (`Cross()`, operador `*`), ângulo entre vetores (`Angle()`), projeção sobre um vetor (`Project()`), norma (`Norm()`) e vetor unitário (`Unitary()`).

Por fim, também estão disponíveis métodos para a transformação de coordenadas, como rotação (`Rotate()`), translação (`Translate()`), escala (`Scale()`), distorção (`Shear()`) e transformações homogêneas (`Transform()`). O modelo desta classe é apresentado na Figura 4.5

4.3.3 Resolução de Sistemas de Equações Diferenciais

Para a solução de sistemas de equações diferenciais ordinárias foram criadas as classes `TODESolver` e `TRKutta`. A primeira implementa o método de Euler, enquanto a segunda é uma implementação genérica do método de Runge-Kutta de passo fixo, cuja ordem pode ser definida dinamicamente. Para representar os sistemas de equações diferenciais ordinárias foi definida a classe abstrata `TODE`. Essas classes tem seus modelos mostrados na Figura 4.6.

Para seu funcionamento, uma instância de `TODESolver` precisa ter definidos o número de equações do sistema a resolver, o passo de integração e o passo de amostragem. O passo de amostragem especifica com que freqüência os valores das variáveis de estado do sistema são armazenados ao longo da sua solução[38]. A definição desses parâmetros pode ser feita ao criar um objeto `TODESolver`, ou através do método `Define()`.

Definidos os parâmetros iniciais, utiliza-se o método `Solve()` para a solução do sistema de equações diferenciais ordinárias. Estas equações devem estar modeladas em uma instância de classe descendente de `TODE`, sendo passada como parâmetro do método, junto com o intervalo da solução (definido pelos valores inicial e final da variável independente) e uma instância de `Matrix`, onde o resultado será armazenado[39]. A matriz de

[38] O uso de um passo de amostragem possibilita armazenar um menor número de pontos que os calculados ao se empregar um passo de integração pequeno, normalmente empregado para um aumento de precisão na solução do sistema. Assim, pode-se desprezar pontos intermediários aos pontos de interesse da solução, além de economizar memória.

[39] Opcionalmente, pode-se indicar que os valores da variável independente devem ser armazenados na matriz de resultados, além de se especificar quais variáveis de estado do sistema e suas derivadas serão armazenadas. Por *default*, são armazenadas todas as variáveis de estado, não são armazenadas as derivadas e nem os valores da variável independente.

resultados terá tantas colunas quanto variáveis do sistema armazenadas, e o número de linhas será igual ao número de pontos coletados no intervalo da solução com o passo de amostragem especificado.

Outra variação do método `Solve()` precisa apenas do sistema de equações e de um valor limite para a variável independente como parâmetros, considerando o valor inicial igual a 0. O método é executado até a variável independente atingir o valor limite ou a condição definida pelo usuário na classe que representa o sistema resultar falso. O resultado retorna em uma matriz com as variáveis de estado.

A classe `TRKutta` é descendente de `TODESolver`, com a mesma interface de uso. A única diferença consiste na definição da ordem do método de Runge-Kutta a utilizar, podendo ser de segunda, terceira, quarta, quinta ou oitava ordem. Quanto maior a ordem do Runge-Kutta, mais acurada é a resposta, e maior é a carga computacional do método[19].

Como há correspondência total entre as duas classes, os métodos de `TRKutta` sobrepõem os métodos da sua classe base, `TODESolver`, definindo características de polimorfismo. Isso significa que uma referência a instâncias de `TODESolver` pode referenciar instâncias de `TRKutta`. Em tempo de execução, a chamada a um método de um objeto referenciado implicará na determinação da sua natureza, para que o método correto seja posto em execução, em um processo denominado *vinculação tardia.*

Como exemplo de aplicação desta característica da hierarquia de classes, pode-se implementar apenas um trecho de código para resolver o sistema de equações no simulador, sendo o método de solução determinado em tempo de execução através da referência a uma instância de `TODESolver` ou `TRKutta`.

Para utilizar essas classes na solução de sistemas de equações diferenciais ordinárias, o usuário deverá modelar esse sistema em uma classe derivada da classe `TODE`[40]. O construtor da classe derivada deve definir as dimensões das matrizes `X0` e `dX`, que correspondem aos valores iniciais das variáveis de estado e à derivada, respectivamente. Devem ser implementados obrigatoriamente os métodos `Evaluate()` (que calcula as derivadas das variáveis de estado segundo o sistema de equações diferenciais),

[40] Esta classe não pode ser utilizada diretamente por ser abstrata.

`SampleAction()` (ação a ser executada a cada conjunto de variáveis armazenado na matriz de resultados) e `Continue()` (utilizado pelo método `Solve()` para testar a continuidade da execução da solução do sistema).

O Anexo 1 deste trabalho apresenta um exemplo de uso das classes acima descritas na modelagem e solução de um sistema de equações diferenciais, para determinar a cinemática inversa de um manipulador plano de dois graus de liberdade a partir da cinemática diferencial (parâmetros definidos no ítem 5.1 do Capítulo 5).

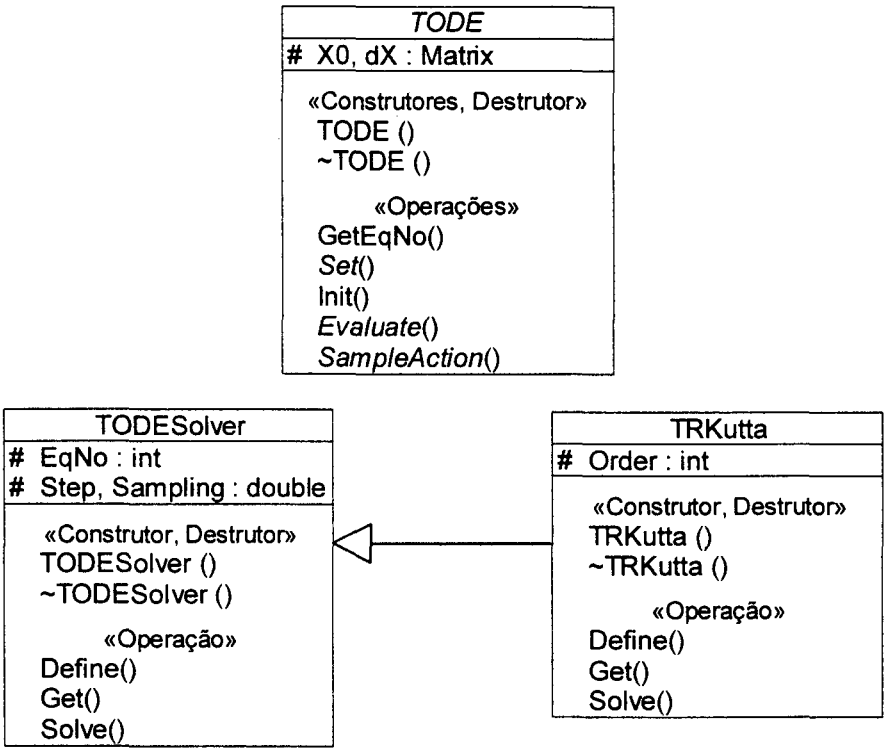


Figura 4.6 Modelos das Classes `TODE`, `TODESolver` e `TRKutta`

Além dessas classes, estão implementadas na biblioteca `CrExtMth` funções para cálculo de matrizes de transformação homogênea correspondendo à translações, rotações e escalas. Estão também definidas constantes como Pi, múltiplos e submultíplos comuns, funções de conversão de unidades de ângulos e outras úteis em expressões matemáticas.

O Anexo 1 deste trabalho contém uma descrição completa das classes, tipos e métodos presentes na biblioteca `CrExtMth`, além de exemplos de uso.

4.4 MODELAGEM DE ROBÔS MANIPULADORES

A modelagem dos robôs manipuladores é feita pela classe `TManipulator`. Nela são armazenados os parâmetros cinemáticos e dinâmicos, além de definições da forma dos robôs. Também são implementados métodos para determinação da cinemática e da dinâmica dos manipuladores, tanto direta como inversa, e métodos para desenho do modelo do manipulador em um ambiente 3D. Para armazenar as definições dos manipuladores, a classe conta com métodos que geram e lêem arquivos texto em um padrão de fácil entendimento e modificação. A Figura 4.7 mostra a definição da classe `TManipulator` e suas componentes.

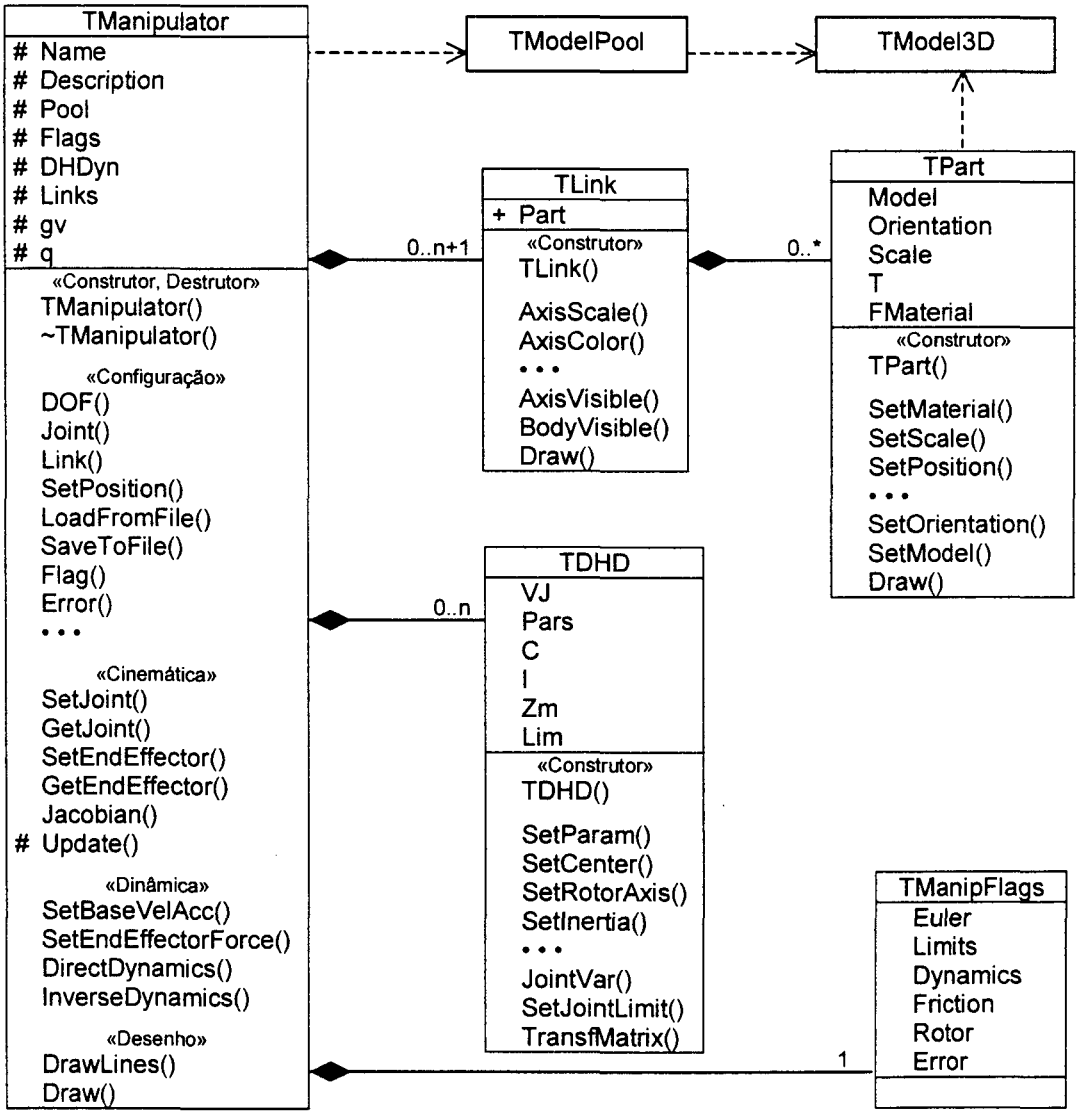


Figura 4.7 Modelo da classe `TManipulator` e suas componentes

4.4.1 PARÂMETROS CINEMÁTICOS E DINÂMICOS

O número de graus de liberdade do manipulador pode ser definido pelo método `DOF()`. O método `Add()` adiciona um grau de liberdade ao manipulador (por default, no final da cadeia cinemática, mas pode ser em outro ponto). `Del()` remove um grau de liberdade, junto com o elo correspondente. `Move()` e `Exchange()` podem ser usados para mudar a ordem das juntas na cadeia cinemática, modificando a estrutura do manipulador.

Para representar a cinemática do manipulador, utilizam-se os parâmetros Denavit-Hartenberg (a, α, d, θ) e uma variável que indica qual destes é a variável de junta. Para cada elo, os parâmetros dinâmicos de interesse são a massa, o centro de massa e o tensor de inércia, estes últimos definidos segundo o sistema de coordenadas fixo do elo. Se forem considerados efeitos do atrito nas juntas, devem ser indicados os coeficientes de atrito estático e dinâmico, e para levar em conta os efeitos do rotor, necessita-se da inércia deste, a direção do seu eixo e da relação de transmissão usada no atuador.

Os parâmetros são armazenados em uma lista de objetos da classe `TDHD`, cujo número de elementos corresponde aos graus de liberdade do manipulador. Esses elementos podem ser acessados através do método `Joint()`, que recebe como parâmetro o número do grau de liberdade (entre `1` e `DOF()`) e retorna uma referência ao elemento desejado.

A classe `TDHD` implementa os métodos `SetParam()` e `GetParam()` para acessar qualquer parâmetro cinemático ou dinâmico. Existem métodos mais específicos a alguns tipos de parâmetros, como `SetCenter()` e `GetCenter()`, que utilizam instâncias de `Vector` para definir o centro de massa do elo, `SetInertia()` e `GetInertia()`, que trabalham com instâncias de Matrix para definir o tensor de inércia, e `SetRotorAxis()` e `GetRotorAxis()`, que definem o eixo do rotor do atuador através de instâncias de `Vector`[41]. A variável de junta é especificada através do método `JointVar()`[42].

Além dos parâmetros cinemáticos e dinâmicos, instâncias de `TDHD` armazenam os limites de posição, velocidade, aceleração e torque de uma junta, acessados através dos métodos `SetJointLimit()` e `GetJointLimit()`, que são considerados se a opção `mfLimits` do manipulador estiver ativa.

---

[41] Este é definido segundo o sistema de coordenadas do elo anterior, por usualmente estar vinculado a este[2].
[42] Pode assumir valores entre 0 e 3, de acordo com a ordem dos parâmetros Denavit-Hartenberg apresentada.

4.4.2 CINEMÁTICA

A configuração inicial do manipulador é estabelecida pelos parâmetros cinemáticos armazenados na lista de objetos `TDHD`. As variáveis das juntas são acessadas pelos métodos `SetJoint()` e `GetJoint()`, que além de posições manipulam velocidades, acelerações e torques nas juntas, de acordo com o parâmetro passado na chamada do método.

A cada modificação na configuração, ativa-se um sinal que indica aos demais métodos a necessidade de atualização das matrizes de transformação que definem os sistemas de coordenadas dos elos. Em função desse sinal, o método `Update()` pode ser invocado automaticamente quando da execução dos métodos para determinação da posição do efetuador final, cálculo do jacobiano ou da dinâmica do manipulador.

A posição e orientação do efetuador final pode ser obtida pelo método `GetEndEffector()`, que pode retornar a matriz de transformação resultante da cinemática do manipulador ou uma representação mínima utilizando ângulos de Euler. Como muitas vezes há necessidade de se especificar essa posição, ao invés de uma configuração no espaço das juntas, foi criado o método `SetEndEffector()`, que recebe uma representação mínima da posição do efetuador final (usando ângulos de Euler) e procura determinar uma configuração das variáveis das juntas através do algoritmo de cinemática inversa em malha fechada apresentado no capítulo 2. Se há uma configuração possível, o resultado é armazenado nas variáveis das juntas, sendo acessível através de `GetJoint()`. Caso contrário, o método retorna um erro, indicando a impossibilidade devido a uma singularidade ou especificação de posição fora do espaço de trabalho do manipulador.

O algoritmo de inversão cinemática necessita do jacobiano, que é calculado pelo método `DetJacobian()`. Se desejado, o acesso à matriz jacobiana é possível pelo uso do método `Jacobian()`.

A qualquer instante, a estrutura cinemática do manipulador pode ser desenhada em um contexto de renderização OpenGL (como o da câmera sintética implementada para o simulador) através do método `DrawLines()`. Se desejado, um modelo mais realista pode ser visualizado através do método `Draw()`, exigindo porém a modelagem do corpo do manipulador.

4.4.3 DINÂMICA

Os métodos `DirectDynamics()` e `InverseDynamics()` calculam a dinâmica direta e a dinâmica inversa do manipulador, respectivamente. Para tanto, a matriz de inércia e os termos não lineares da equação dinâmica são calculados através da formulação de Newton-Euler, conforme o item 3.3 do capítulo de Modelagem Dinâmica.

Ambos os métodos recebem como parâmetro uma referência a uma matriz, que contém as posições, velocidades, acelerações e torques nas juntas, nessa ordem, em forma de linha. Para `DirectDynamics()`, as acelerações são calculadas em função das posições, velocidades e torques passados na matriz de parâmetros. Em `InverseDynamics()`, os torques são determinados a partir das posições, velocidades e acelerações passados na matriz de parâmetros. Nos dois casos, o resultado é armazenado nessa mesma matriz.

Se desejado, pode-se obter tanto a matriz de inércia quanto os termos não lineares diretamente através de `InertiaMatrix()` e `NonLinearTerm()`, respectivamente. Estes calculam os termos da equação dinâmica em função da configuração armazenada no manipulador, não exigindo parâmetros.

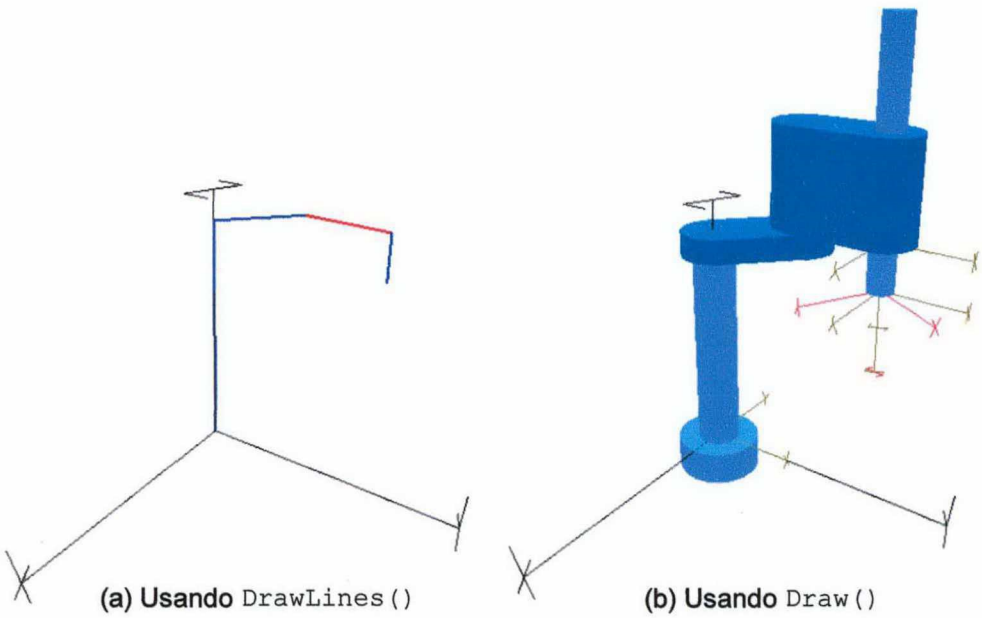
Toda a modelagem do manipulador é estabelecida em relação ao sistema de coordenadas da sua base, tomado como sistema inercial. Assim, é fundamental a definição da posição e orientação do manipulador através de `SetPosition()`, para que a gravidade seja expressa em relação ao sistema de coordenadas da base. Além disso, para a determinação da dinâmica segundo Newton-Euler, devem ser definidas a aceleração linear e a velocidade e a aceleração angulares da base, bem como os esforços no efetuador final. Estes termos são definidos através de `SetBaseVelAcc()` e `SetEndEffectorForce()`, respectivamente.

O atributo `Flags` define o comportamento do modelo do manipulador, afetando diretamente a determinação da dinâmica deste. Seu acesso é feito pelo método `Flag()`, que recebe como parâmetro o tipo de sinal e o valor (0 se desativado e 1 se ativado). Se o sinal `mfFriction` está ativo, são calculados os efeitos do atrito nas juntas. Se o sinal `mfRotor` estiver ativo, a inércia do rotor é considerada. Os limites das juntas são obedecidos se o sinal `mfLimits` for ativado. Caso seja analisada apenas a cinemática do manipulador, sem definição dos parâmetros dinâmicos, é aconselhável desativar o sinal `mfDynamics`, para

que nenhuma operação que os envolva seja realizada. Se passado apenas o parâmetro de tipo de sinal, o método `Flag()` retorna seu status, sem modificá-lo.

4.4.4 DESENHANDO O MANIPULADOR

Existem dois métodos para desenhar o manipulador. O método `DrawLines()` desenha a estrutura cinemática deste através de uma série de linhas. O método `Draw()` desenha os eixos dos sistemas de coordenadas dos elos do manipulador e, se estiver disponível um modelo do corpo do manipulador, desenha as formas deste. Ambos os métodos usam primitivas de desenho da biblioteca OpenGL, sendo necessário o uso de um contexto de renderização para a geração da imagem. A Figura 4.8 ilustra o uso dos dois métodos.



(a) Usando `DrawLines()` (b) Usando `Draw()`

Figura 4.8 Uso de `DrawLines()` e `Draw()` para desenhar um manipulador

Para modelar o corpo do manipulador, utiliza-se um conjunto de sólidos geométricos, representados por instâncias da classe `TModel3D`. Esta define os sólidos através de malhas poligonais (ou *mesh*, em inglês), que aproximam a forma da superfície pela especificação de uma série de pontos pertencentes a esta, que relacionados entre si formam faces poligonais[13]. Entre métodos existentes para esta classe, destacam-se `SaveToFile()` e `LoadFromFile()`, que permitem armazenar e recuperar as definições de

sólidos em um arquivo texto com um formato específico (extensão padrão .m3d). A modelagem de `TModel3D` é mostrada na Figura 4.9. Um exemplo de sólido é mostrado na Figura 4.10.

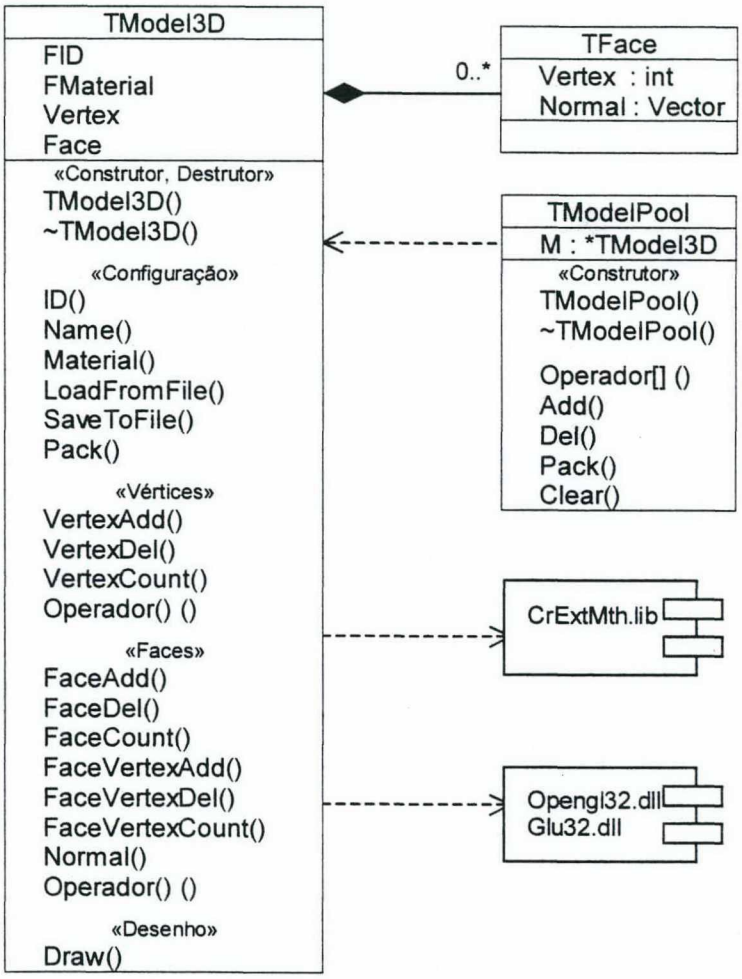


Figura 4.9 Definição da classe `TModel3D`

```
BeginModel
NAME=Cubo
ID=2
VERTEXNUM=8
FACENUM=6
AMBIENT=1.0,0.341,0.349,1.0
DIFFUSE=1.0,0.341,0.349,1.0
SPECULAR=1.0,0.95,0.95,1.0
SHININESS=50.0
EMISSION=0,0,0,1.0
V=1,1,1;1,-1,1;1,-1,-1;1,1,-1
V=-1,1,-1;-1,-1,-1;-1,-1,1;-1,1,1
F=4:0,1,2,3
F=4:4,5,6,7
F=4:0,3,4,7
F=4:0,7,6,1
F=4:1,6,5,2
F=4:5,4,3,2
EndModel
```

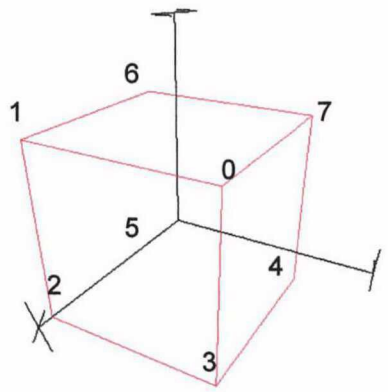


Figura 4.10 Modelo de um cubo segundo o formato de definição m3d[43]

Cada elo é representado por um objeto da classe `TLink` em uma lista cujos elementos são acessados pelo método `Link()`, que recebe como parâmetro a posição do elo e retorna uma referência ao elemento desejado, considerando-se a base como elo 0.

Um objeto `TLink` contém definições para o desenho dos eixos de um elo, como visibilidade, cor e escala. As partes que formam o desenho do elo são especificadas em uma lista de objetos `TPart`, que armazena a identificação do sólido, sua posição e sua orientação em relação ao sistema de coordenadas do elo. Como um mesmo poliedro pode ser usado em tamanhos e materiais diferentes, são armazenados também fatores de escala e definição do material do elemento, se este for diferente do originalmente especificado pela instância correspondente de `TModel3D`. Os objetos `TModel3D` utilizados no desenho dos elos são armazenados em uma lista, que deve ser referenciada pela instância de `TManipulator`.

[43] As variáveis `Ambient`, `Diffuse`, `Specular`, `Shininess` e `Emission` definem as características de cor ambiente, difusa e especular, além da emissividade e brilhância, respectivamente, que compõem o material das superfícies, de acordo com o OpenGL[29].

4.4.5 ARMAZENAMENTO DA MODELAGEM

As definições dos parâmetros cinemáticos e dinâmicos do manipulador e dos elementos que definem a forma de seu corpo podem ser armazenados em um arquivo texto. Ao invés de se utilizar um formato binário específico, o uso de arquivos texto traz como vantagens a facilidade de análise e edição através de um editor de textos comum.

Para salvar a definição de um manipulador executa-se o método `SaveToFile()`. Uma instância de `TManipulator` pode ser inicializada com o nome de um arquivo que contenha a definição de um robô, para que seus parâmetros sejam carregados deste. Após a criação do objeto, a mesma ação é executada pelo método `LoadFromFile()`.

O arquivo de definição de robôs manipuladores (cuja extensão padrão é .rdf - robot definition file) segue uma organização específica, proposta pelo autor, sendo estruturado em seções. Algumas das seções e parâmetros são opcionais, de acordo com o tipo de modelagem feita para o manipulador.

A seção [`Config`] é obrigatória, e deve ser a primeira do arquivo. As linhas desta seção têm o formato <`Variável`>=<`Valor`>. A variável `Name` define um nome que identifica o manipulador. `DOF` deve informar o número de graus de liberdade do manipulador. As variáveis `Position` e `Orientation` devem conter a posição e a orientação do manipulador segundo o sistema de coordenadas referencial do ambiente, respectivamente. A orientação deve ser expressa em ângulos de Euler.

A variável `Flags` define os sinais que são considerados durante a simulação. Seus valores são separados por um (|). As palavras-chave ativam ou desativam um sinal. Para considerar|desconsiderar os limites das juntas, adiciona-se a palavra `Limits|NoLimits`. A dinâmica é ativada|desativada pela palavra `Dynamics|NoDynamics`. Os efeitos do atrito e da inércia do atuador são ativados|desativados pelo emprego de `Friction|NoFriction` e `Rotor|NoRotor`, respectivamente. A Figura 4.11 exemplifica as definições dessa seção.

```
[CONFIG]
Name=Manipulador Plano 2 DOF
DOF=2
Flags=Euler|NoLimits|Dynamics|Rotor
Position=0.0,0.0,0.3      # X, Y, Z
Orientation=0.0,pid2,pid2 # ângulos de Euler ZYZ

# Qualquer parte do texto após o sinal (#) na linha
# é considerada comentário e é descartada por LoadFromFile()
```

Figura 4.11 Seção [Config] de um manipulador plano com dois graus de liberdade

Pode-se escrever um texto detalhando o manipulador na seção [Description], que é opcional. Todas as linhas entre este cabeçalho e o da próxima seção serão consideradas como comentários sobre o robô.

A seção [DH-Dynamics] é o local onde os parâmetros cinemáticos e dinâmicos do manipulador são especificados, sendo obrigatória no arquivo. Cada linha desta seção descreve os parâmetros cinemáticos e dinâmicos de um grau de liberdade, sendo necessárias tantas linhas quanto o valor da variável DOF da seção [Config].

Cada linha inicia por um número que indica a variável da junta (2 para juntas de translação, 3 para juntas de rotação). Após um (;), seguem-se os parâmetros Denavit-Hartenberg, na ordem a, α, d, θ, separados por (,)[44]. Se a dinâmica do robô for especificada, os parâmetros seguem na mesma linha, separados por (,), na ordem massa, centro de massa, tensor de inércia, coeficientes de atrito estático e viscoso, inércia do rotor, relação de transmissão e eixo do rotor[45]. A Figura 4.12 ilustra essa seção.

[44] Comprimentos em metros e graus em radianos, podendo ser utilizadas as constantes Pi (π), Pix2 (2π), Pid2 ($\pi/2$), Pid3 ($\pi/3$), Pid4 ($\pi/4$) e Pid6 ($\pi/6$), com sinal + ou -.

[45] Massa em kg, componentes do centro de massa e do eixo do rotor em metros, componentes do tensor de inércia e inércia do rotor em Nm^2, coeficiente de atrito estático em Nm (N) e coeficiente de atrito viscoso em Ns/rad (Ns/m) para juntas de rotação(translação). O eixo do rotor é especificado em relação ao sistema de coordenadas do elo anterior.

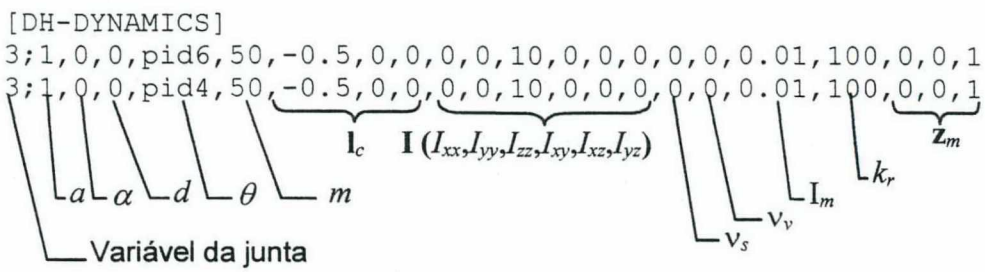


Figura 4.12 Seção [DH-Dynamics] de um manipulador plano com dois graus de liberdade

A seção [Limits] também é opcional, e descreve os limites de cada junta. Cada linha deverá conter, separado por (,), os limites mínimo e máximo de posição, velocidade, aceleração e torque, como mostra a Figura 4.13.

```
[LIMITS]
#qmin, qmax, vmin, vmax, amin, amax, τmin, τmax
-pix2,pix2,0,0,0,0,0,0
-pix2,pix2,0,0,0,0,0,0
```

Figura 4.13 Seção [Limits] de um manipulador plano com dois graus de liberdade

Se for modelado o corpo do manipulador, a seção [ModelList] pode ser utilizada para relacionar os sólidos geométricos utilizados para desenhar os elos. Estes são armazenados em uma lista de modelos, e substituirão os modelos já existentes na lista que tiverem a mesma identificação (atributo ID de TModel3D). Uma forma de especificar um modelo consiste no nome do arquivo que contém a sua descrição através da variável File=<modelo>. Pode-se também incorporar toda a definição do modelo, entre as palavras BeginModel e EndModel, conforme ilustrado na Figura 4.14.

```
[MODELLIST]
File=barra.m3d
BeginModel
NAME=Cubo
ID=2
VERTEXNUM=8
FACENUM=6
AMBIENT=1.0,0.341,0.349,1.0
DIFFUSE=1.0,0.341,0.349,1.0
SPECULAR=1.0,0.95,0.95,1.0
SHININESS=50.0
EMISSION=0,0,0,1.0
V=1,1,1;1,-1,1;1,-1,-1;1,1,-1;-1,1,-1;-1,-1,-1;-1,-1,1;-1,1,1
F=4:0,1,2,3
F=4:4,5,6,7
F=4:0,3,4,7
F=4:0,7,6,1
F=4:1,6,5,2
F=4:5,4,3,2
EndModel
```

Figura 4.14 Seção [ModelList] de um manipulador plano com dois graus de liberdade

A seção [Links] descreve atributos dos eixos coordenados de cada elo e relaciona os sólidos geométricos utilizados no desenho de um elo. Para cada elo, começa-se com uma variável Link=<número do elo>[46]. Segue-se a variável Axis=<atributos dos eixos>, que consiste na definição de visibilidade (Visible ou Invisible), cor (no formato R,G,B,A) e escala. Para cada sólido componente do elo deve ser incluída uma linha no formato Model=<atributos do poliedro>, que consiste na identificação do modelo 3D, sua posição e orientação relativa à origem do sistema de coordenadas do elo, escala e definição do material (se diferente da definição contida no modelo original). O modelo deve estar presente na lista de sólidos geométricos utilizada pelo manipulador (ModelPool). A Figura 4.15 exemplifica essa seção.

[46] O elo 0 corresponde à base do manipulador.

```
[LINKS]
Link=0
Visibility=InVisible
Axis=Visible;0,1,1,1;0.25
Link=1
Visibility=Visible
Axis=Visible;0,1,1,1;0.25
Model=11;0,0,0;0,0,0;1,1,1
Link=2
Visibility=Visible
Axis=Visible;1,0,0,1;0.25    #R=1,G=0,B=0,A=1 ; Escala=0.25
Model=11;0.0,0.0,-0.101;0.0,0.0,0.0;1.0,1.0,1.0
```

Posição (x, y, z) — Orientação (φ, θ, ψ) — Escala (S_x, S_y, S_z)

Id do Modelo

Figura 4.15 Seção [Links] de um manipulador plano com dois graus de liberdade

No Anexo 3 deste trabalho são apresentadas listagens completas de vários manipuladores modelados neste formato.

4.5 MODELAGEM E IMPLEMENTAÇÃO DE CONTROLADORES

A relação entre robótica e teoria de controle é muito forte. Para controle, os robôs são um bom estudo de caso para teste e desenvolvimento de controladores, por serem sistemas multivariáveis altamente não lineares, com as equações fortemente acopladas entre si. A robótica, por sua vez, aproveita as novas técnicas de controle, algumas desenvolvidas especificamente para suas aplicações, para o aumento da performance dos manipuladores[7].

O controle de um manipulador consiste na determinação dos esforços a serem exercidos nos atuadores a fim de garantir a execução de uma tarefa ao longo do tempo, procurando satisfazer certos requisitos de performance. Pode-se classificar o controle de acordo com a natureza da tarefa. Se o efetuador final pode se mover livremente no espaço operacional, aplica-se o *controle do movimento*. Se houver interação com o meio, torna-se necessário acrescentar o *controle de força*[1,2,7,11,20].

O controle de posição pode ser realizado no *espaço das juntas* ou no *espaço operacional*. No primeiro caso, as referências da tarefa são mapeadas do espaço operacional para o espaço das juntas através de inversão cinemática[47], e após são utilizadas como entrada no algoritmo de controle, que determina os torques a serem aplicados nos atuadores. O *controle no espaço operacional*, por sua vez, utiliza como entradas os pontos da tarefa no espaço operacional e a posição do efetuador final, determinando os torques aplicados nos atuadores.

O controle no espaço das juntas apresenta como inconveniente o fato de incertezas na estrutura do manipulador (causadas por elasticidade nas juntas ou elos, folgas ou tolerâncias construtivas) poderem levar a erros na posição do efetuador final, pois esta não é diretamente controlada neste esquema, sendo determinada em função da estrutura do manipulador e das variáveis das juntas (estas são controladas).

Embora o controle no espaço operacional não apresente o problema do esquema anterior, sua complexidade aumenta a carga computacional envolvida. Além disso, normalmente as variáveis do espaço operacional não são medidas diretamente, sendo determinadas com o uso da cinemática direta a partir das variáveis das juntas, estando esse tipo de controle sujeito ao mesmo tipo de inconveniente, na prática[2].

Os problemas de controle, por sua vez, podem ser classificados em *regulação*, onde o manipulador deve atingir uma dada configuração especificada na tarefa a partir de uma configuração inicial, e *seguimento de trajetória*, onde a tarefa consiste em um conjunto de posições, velocidades e acelerações ao longo do tempo, sendo o objetivo de controle seguir essa referência a despeito de perturbações e dinâmicas não modeladas.

O trabalho ora apresentado não tem por finalidade apresentar e analisar as diferentes estratégias de controle, pois estes, no caso de controle de posição, são amplamente discutidos na literatura[1, 2, 7, 11]. Seu objetivo principal, em relação aos controladores, consiste em apresentar uma metodologia para possibilitar ao usuário desenvolver seus próprios algoritmos de controle, incorporando-os dinamicamente ao simulador. O *software* desenvolvido neste trabalho considera apenas algoritmos de

[47] Considerando que, usualmente, as tarefas são definidas no espaço operacional.

controle de posição no espaço das juntas, podendo executar tarefas tanto de regulação quanto de seguimento de trajetórias no espaço das juntas ou no espaço operacional.

O mecanismo adotado para permitir a implementação dos algoritmos de controle independente do simulador, possibilitando porém seu uso durante as simulações, é o das *bibliotecas de ligação dinâmica* (*dynamic-link libraries*, ou dll). Estas consistem em módulos armazenados em arquivos que contém funções e dados, que podem ser carregadas e descarregadas em tempo de execução de acordo com as necessidades das aplicações. Uma das principais vantagens no uso dessas bibliotecas está na facilidade de desenvolvimento, atualização e reusabilidade dos módulos, sem precisar modificações no *software* principal[32].

Para desenvolver um controlador, o usuário deve criar uma dll, que implementará uma classe derivada da classe abstrata `TDllController`. Esta modela o comportamento desejado para o controlador. A dll também deverá conter uma função com o nome `CreateInstance()`, que criará no simulador uma instância do controlador implementado. Qualquer compilador C++ que suporte a criação de bibliotecas dinâmicas para Windows poderá ser utilizado.

Todos os métodos da classe derivada deverão ser implementados. O método `Reset()` é utilizado para definir as condições iniciais do controlador. Os torques de controle são determinados pelo método `Evaluate()`, que recebe como parâmetros uma matriz com os valores de referência, uma matriz com as variáveis das juntas e uma matriz que receberá os torques nas juntas. O método `Set()` deve ser utilizado para definir os ganhos e outros parâmetros inerentes ao método de controle, enquanto `Get()` retorna esses valores.

Para auxiliar a criação de uma interface amigável com o usuário, os métodos `ParamNum()`, `ParamSize()`, `ParamName()` e `ParamHelp()` devem estar disponíveis. O primeiro deve retornar o número de parâmetros ajustáveis do controlador. O segundo retorna as dimensões das matrizes de ganhos. `ParamName()` retorna um nome para cada parâmetro e `ParamHelp()` retorna um comentário para esses parâmetros.

Para facilitar a carga dos módulos dos controladores, foi criada a classe `TController`. Esta automatiza os processos de carga e descarga dos módulos, criando uma

instância do controlador interna ao objeto `TController` durante o processo de carga. Para tanto, utilizam-se os métodos `Load()` e `UnLoad()`. Os parâmetros de um controlador, bem como o nome da dll que o contém, podem ser salvos em um arquivo texto através de `SaveFromFile()` e recuperados usando `LoadFromFile()`. Essa classe conta com métodos equivalentes aos de `TDllController`, que verificam se o controlador está carregado e executam os métodos correspondentes deste. A Figura 4.16 apresenta a modelagem destas classes e sua relação.

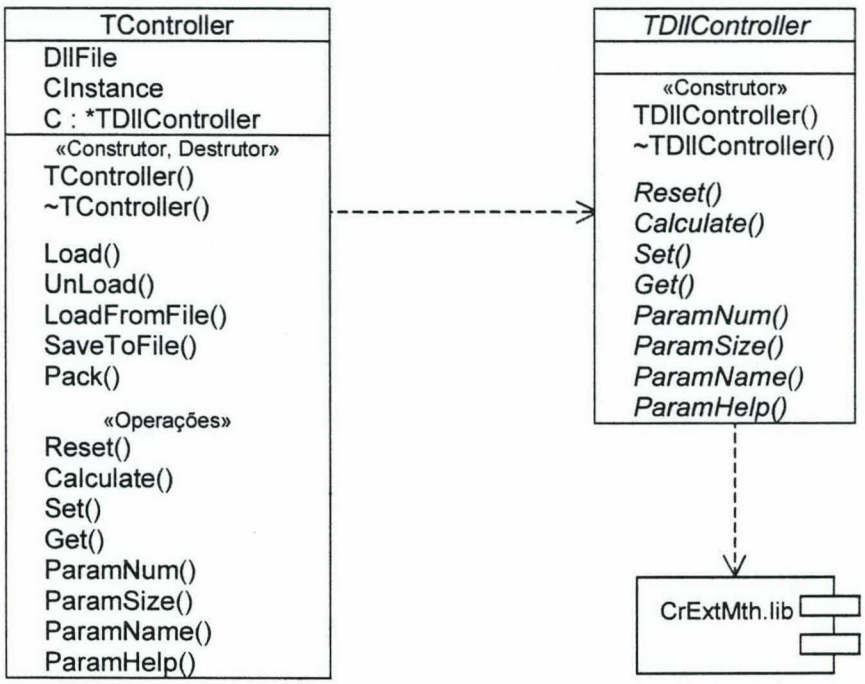


Figura 4.16 Classes `TController` e `TDllController`

O Anexo 3 deste trabalho exemplifica a implementação de controladores em bibliotecas de ligação dinâmica.

4.6 Especificação de Tarefas

Uma tarefa de posição consiste na definição do movimento do manipulador. Este pode ser definido tanto por um caminho quanto por uma trajetória. Um *caminho* denota os pontos que o manipulador deverá alcançar na execução do movimento, enquanto uma

trajetória é um caminho no qual é especificada uma função temporal, que define, além de posições, velocidades e acelerações ao longo do caminho seguido[2].

Em geral, as tarefas são definidas em termos de caminhos, sendo estes utilizados para a geração das trajetórias a serem seguidas. Estas, então, também são definidas por vários pontos, interpolados entre os pontos do caminho em intervalos de tempo menores que os intervalos entre os pontos dos caminhos.

A classe `TTask` é utilizada para representar uma tarefa, contendo como atributos o tipo de tarefa (caminho ou trajetória), em que espaço está especificada (espaço das juntas ou espaço operacional), uma matriz para armazenar os pontos do caminho ou da trajetória e uma matriz para armazenar os valores dos instantes em que os pontos devem ser alcançados (no caso da tarefa ser um caminho) ou o passo entre os pontos e os instantes inicial e final (no caso de uma trajetória). A Figura 4.17 apresenta o modelo dessa classe.

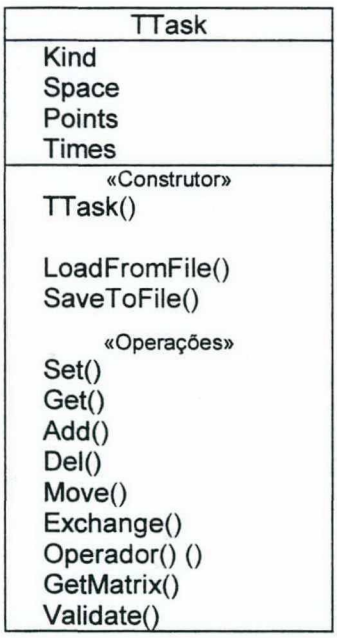


Figura 4.17 Modelo da classe `TTask`

Para definir as características da tarefa, como tipo e em que espaço está definida, utiliza-se o método `Set()`. O método `Get()` retorna essas características. No caso de trajetórias, esses métodos também são utilizados para acessar os valores do passo e dos instantes inicial e final. Trajetórias somente podem ser especificadas no espaço das juntas,

nesta versão do *software*, enquanto caminhos podem ser definidos tanto no espaço das juntas como no espaço operacional.

Para manipular os pontos da tarefa, utilizam-se os métodos `Add()`, `Del()`, `Move()` e `Exchange()`, além do operador`()`. O primeiro permite adicionar um ponto, dado um determinado instante de tempo. O segundo remove um ponto cujo instante for passado como parâmetro. `Move()` e `Exchange()` permite mudar os pontos de posição na relação de pontos, mantendo porém os valores dos instantes intactos. O operador `()` retorna uma referência de ponto a partir do instante especificado, para acesso a este ponto.

Para manipular o conjunto de pontos, pode-se utilizar o método `GetMatrix()`, que retorna referências para as matrizes de pontos e instantes de tempo. Para armazenar a tarefa em um arquivo texto, num formato específico, utiliza-se o método `SaveToFile()`, enquanto sua recuperação é feita pelo método `LoadFromFile()`.

Se a tarefa especificada for um caminho, torna-se necessário o uso de um gerador de trajetória para que o movimento desejado não cause esforços excessivos nos atuadores. Os geradores determinarão os pontos intermediários entre os pontos do caminho, em um intervalo de tempo equivalente ao passo utilizado na resolução do sistema do manipulador.

Se o caminho for especificado no espaço operacional, os seus pontos serão mapeados para o espaço das juntas, antes de serem informados ao gerador de trajetórias. Para não aumentar a carga computacional da solução do sistema, a geração de trajetórias será feita antecipadamente, sendo gerada uma matriz com os valores de posição, velocidade e aceleração de referência.

Existem diversos algoritmos de geração de trajetória, sendo classificados de acordo com o tipo de movimento especificado na tarefa. No movimento *ponto a ponto*, especificam-se apenas os pontos inicial e final do caminho, não havendo preocupação com os pontos intermediários do caminho. Em movimento ao *longo do caminho* (*path motion*), são especificados vários pontos entre o início e o fim da trajetória, sendo este importante no caso de tarefas complexas onde o efetuador final deva manter uma determinada postura ou para evitar obstáculos presentes no espaço de trabalho[2].

Os geradores de trajetória ponto a ponto usualmente consideram que as velocidade e acelerações nos instantes inicial e final da trajetória são nulos. Se simplesmente

estendidos para uso em caminhos formados por mais de dois pontos, estes métodos gerarão uma trajetória formada por sub-trajetórias entre cada dois pontos do caminho, causando descontinuidade nas acelerações e velocidades para esses pontos intermediários. Por esse motivo, é interessante o uso de algoritmos voltados para movimento ao longo do caminho.

Os geradores de trajetória disponíveis no simulador são modelados em uma hierarquia de classes onde a classe `TTrajectory` é a classe raiz, como mostra a Figura 4.18.

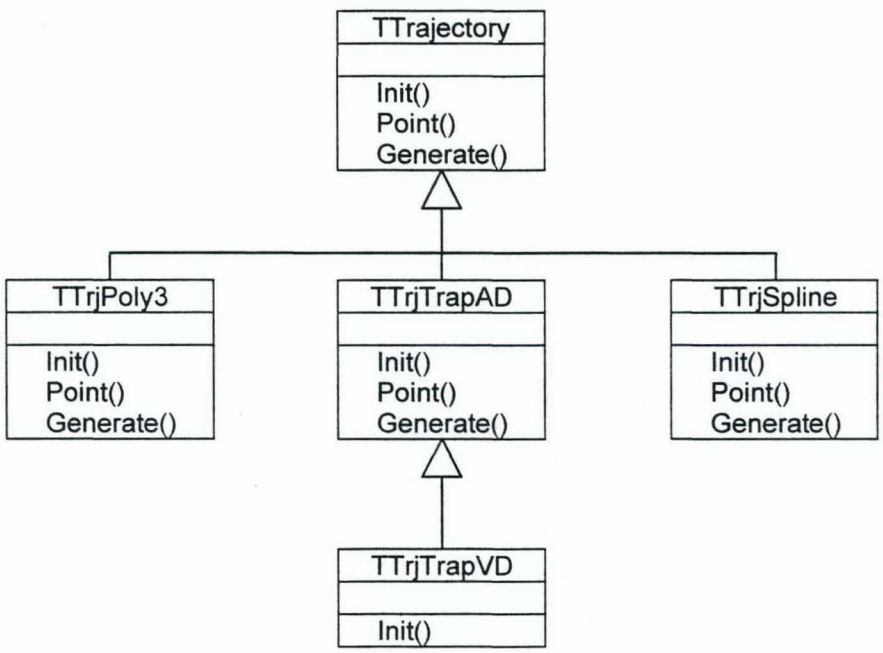


Figura 4.18 Hierarquia de classes de geradores de trajetória

Em todas as classes, são três os métodos que caracterizam o tipo de gerador de trajetórias. O método `Init()` recebe como parâmetros uma matriz com os pontos do caminho e uma matriz com os instantes correspondentes a cada ponto, inicializando o gerador de trajetórias. Para obtenção de um único ponto da trajetória, utiliza-se o método `Point()`. A geração de trajetória em um intervalo é feita pelo método `Generate()`, que também recebe como parâmetros o intervalo de tempo entre os pontos e uma referência a uma matriz que armazenará os valores de posição, velocidade e aceleração da trajetória.

A classe `TTrajectory` não implementa nenhum gerador de trajetória. Na verdade, a idéia inicial era torná-la uma classe base abstrata para somente definir o comportamento

das demais. Havia, porém, a necessidade de um gerador de pontos para tarefas de regulação, onde uma referência de posição é em um determinado intervalo, com valores de velocidade e aceleração desejados nulos. Assim, foi implementado este algoritmo de "geração de trajetórias" nessa classe. Um exemplo de resultado é mostrado na Figura 4.19, onde foram especificadas uma posição inicial $\pi/2$, uma posição π (mantida até 2.5s) e uma posição final $\pi/2$.

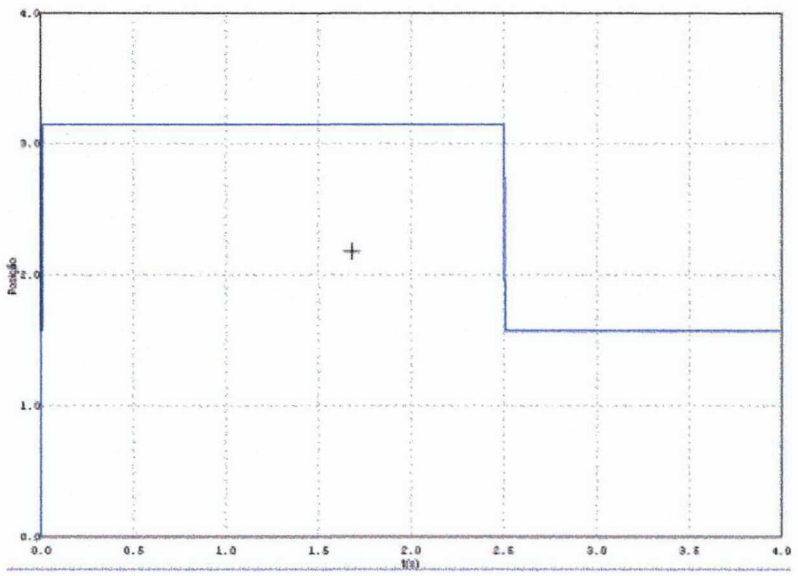


Figura 4.19 Referências de posição constantes entre 0-2.5s e 2.5-4s

Para garantir suavidade e continuidade no movimento entre um ponto inicial e um ponto final, pode-se utilizar uma função polinomial para a interpolação dos pontos intermediários. A classe `TTrjPoly3` implementa um gerador de trajetória ponto a ponto polinomial de ordem 3, que corresponde à ordem mínima para o polinômio garantir continuidade nas acelerações. As equações que definem as posições, velocidade e acelerações são, então[48]

$$\mathbf{q}(t) = \mathbf{a}_0 + \mathbf{a}_1 t + \mathbf{a}_2 t^2 + \mathbf{a}_3 t^3 \tag{4.1}$$

$$\dot{\mathbf{q}}(t) = \mathbf{a}_1 + 2\mathbf{a}_2 t + 3\mathbf{a}_3 t^2 \tag{4.2}$$

$$\ddot{\mathbf{q}}(t) = 2\mathbf{a}_2 t + 6\mathbf{a}_3 t \tag{4.3}$$

[48] As equações referem-se ao espaço das juntas.

Onde os coeficientes podem ser obtidos resolvendo as equações para os instantes inicial e final, sendo informadas as posições e velocidades inicial e final. Em relação às velocidades, considera-se nessa classe que, como o movimento é feito ponto a ponto, as velocidades inicial e final são iguais a zero. Assim, os coeficientes dos polinômios são definidos como

$$\mathbf{a}_0 = \mathbf{q}_i \tag{4.4a}$$

$$\mathbf{a}_1 = \mathbf{0} \tag{4.4b}$$

$$\mathbf{a}_2 = \frac{3(\mathbf{q}_f - \mathbf{q}_i)}{t_f^{\,2}} \tag{4.4c}$$

$$\mathbf{a}_3 = \frac{2(\mathbf{q}_i - \mathbf{q}_f)}{t_f^{\,3}} \tag{4.4d}$$

A Figura 4.20 ilustra os resultados da geração de trajetória polinomial cúbica, para ir de uma posição inicial 0 a uma posição final π em 2.5s.

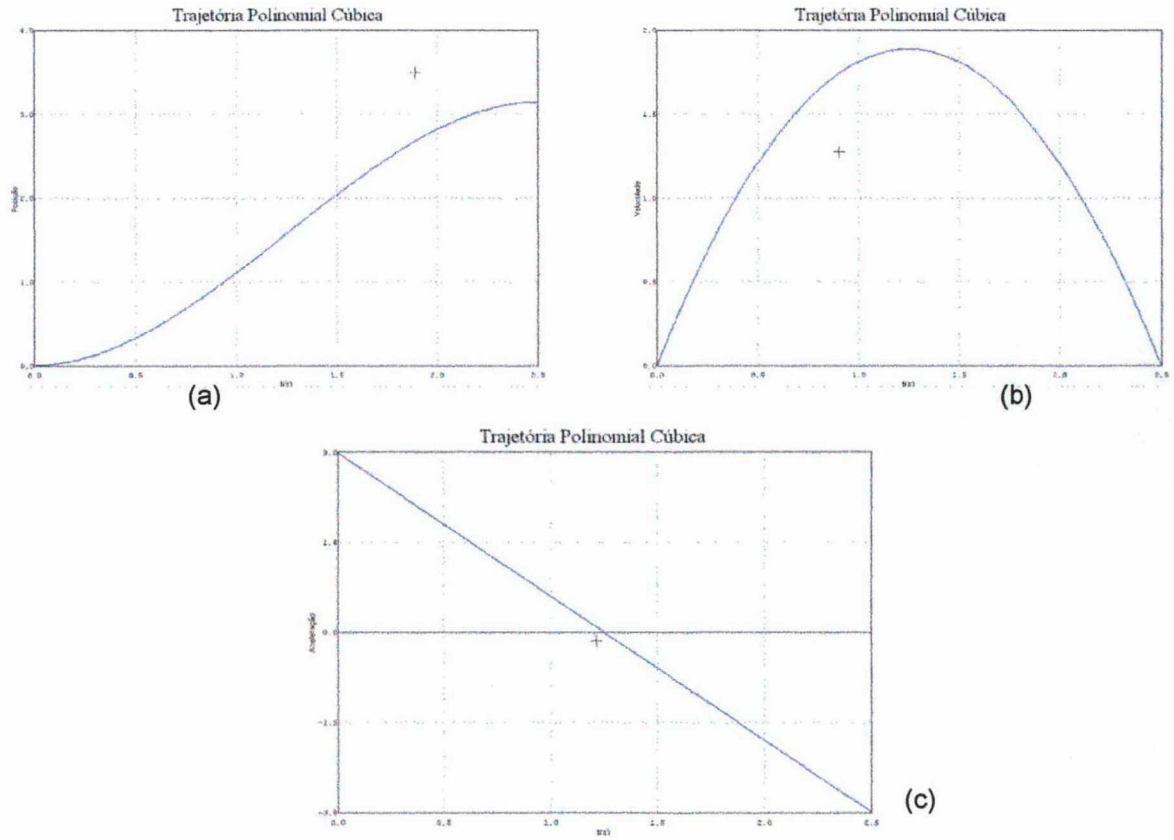


Figura 4.20 (a)Posições, (b)velocidades e (c)acelerações de uma trajetória polinomial cúbica

Um método mais simples de geração, bastante empregado na indústria, consiste no uso de um perfil de velocidade trapezoidal. Este começa o movimento com uma aceleração constante por um determinado tempo t_c até que se atinja uma velocidade "de cruzeiro", sendo esta mantida até faltar um intervalo t_c para o final do movimento, quando o movimento desacelera de forma constante até parar. Assim como na trajetória polinomial cúbica, considera-se que as velocidades inicial e final são iguais a zero. As equações que regem esse movimento são

$$\mathbf{q}(t) = \begin{cases} \mathbf{q}_i + \tfrac{1}{2}\ddot{\mathbf{q}}_c t^2 & ,0 \le t \le t_c \\ \mathbf{q}_i + \ddot{\mathbf{q}}_c t_c \left(t - {t_c}/{2} \right) & ,t_c \le t \le (t_f - t_c) \\ \mathbf{q}_f - \tfrac{1}{2}\ddot{\mathbf{q}}_c (t_f - t)^2 & ,(t_f - t_c) < t \le t_f \end{cases} \tag{4.5}$$

$$\dot{\mathbf{q}}(t) = \begin{cases} \ddot{\mathbf{q}}_c t & ,0 \le t \le t_c \\ \ddot{\mathbf{q}}_c t_c & ,t_c \le t \le (t_f - t_c) \\ \ddot{\mathbf{q}}_c t_c - \ddot{\mathbf{q}}_c (t_f - t) & ,(t_f - t_c) < t \le t_f \end{cases} \tag{4.6}$$

$$\ddot{\mathbf{q}}(t) = \begin{cases} \ddot{\mathbf{q}}_c & ,0 \le t \le t_c \\ \mathbf{0} & ,t_c \le t \le (t_f - t_c) \\ -\ddot{\mathbf{q}}_c & ,(t_f - t_c) < t \le t_f \end{cases} \tag{4.7}$$

As classes `TTrjTrapAD` e `TTrjTrapVD` implementam o mesmo gerador de trajetórias trapezoidais, diferenciando-se na forma de inicialização.

`TTrjTrapAD` necessita da especificação da aceleração inicial e final, estando esta sujeita a restrição

$$|\ddot{\mathbf{q}}_c| \ge \frac{4(\mathbf{q}_f - \mathbf{q}_i)}{t_f^{\,2}} \tag{4.8}$$

onde, se satisfeita a igualdade de (4.8), obtém-se um perfil triangular para a velocidade.

A classe `TTrjTrapVD` utiliza a especificação da velocidade de cruzeiro desejada, sendo esta restrita aos limites

$$\frac{|\mathbf{q}_f - \mathbf{q}_i|}{t_f} < |\dot{\mathbf{q}}_c| \le \frac{2|\mathbf{q}_f - \mathbf{q}_i|}{t_f} \tag{4.9}$$

Novamente, obtém-se perfil triangular se o valor da velocidade de cruzeiro desejada for igual ao limite superior da restrição (4.9).

Ao se especificar um caminho pelo método `Init()`, este verificará se os valores de aceleração (em `TTrjTrapAD`) ou velocidade (em `TTrjTrapVD`) estão dentro dos limites impostos pelas restrições. Caso aconteça, automaticamente assume-se o valor do limite infringido. A Figura 4.21 mostra os resultados obtidos com os geradores de perfis trapezoidais, para ir de uma posição inicial 0 a uma posição final π em 2.5s.

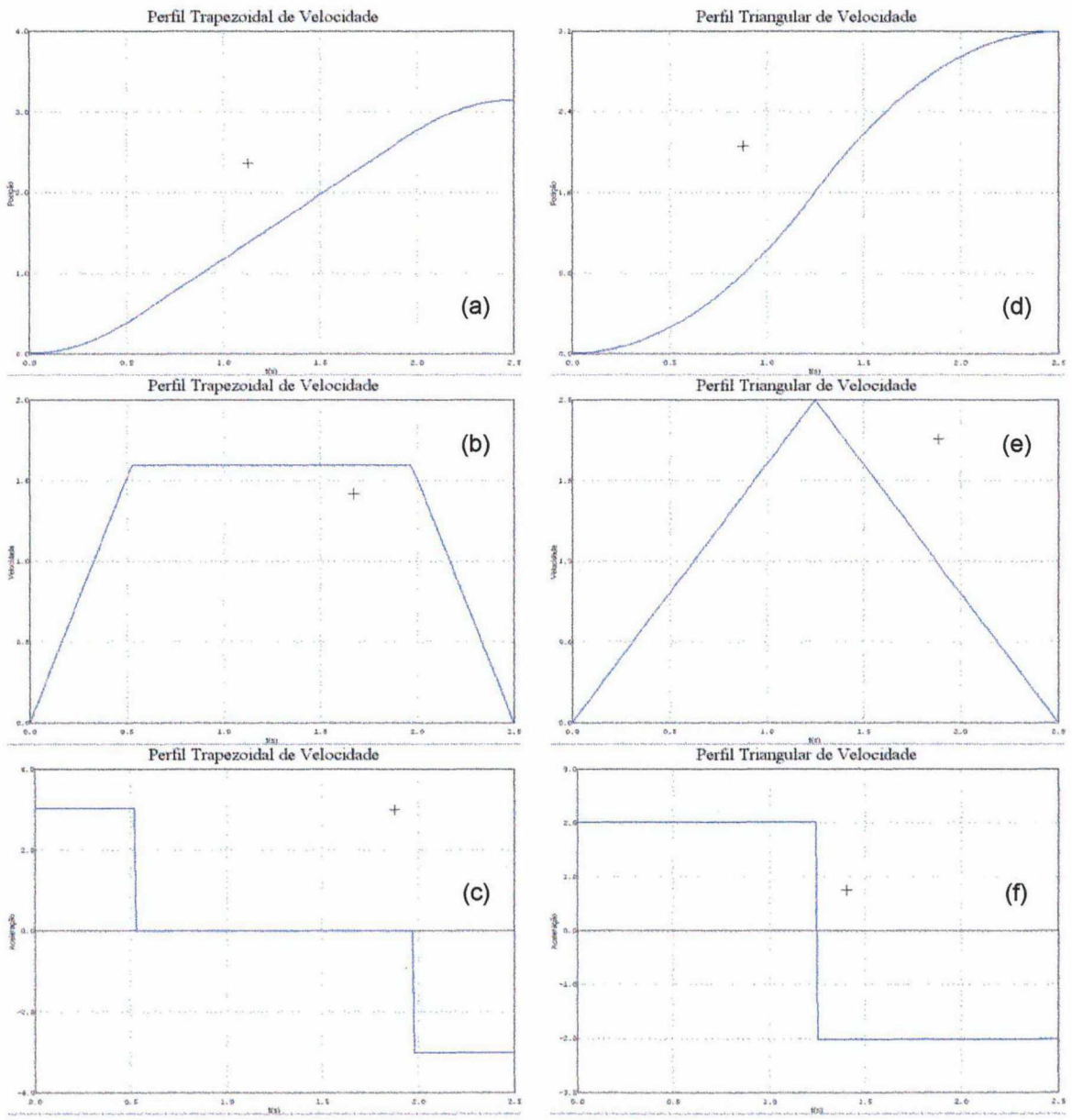


Figura 4.21 (a)Posições, (b)velocidades e (c)acelerações de uma trajetória trapezoidal e de uma (d), (e), (f)trajetória triangular

As classes de geradores de trajetória acima apresentadas podem ser utilizadas em caminhos formados por mais de dois pontos. Nesse caso, serão geradas trajetórias entre cada dois pontos consecutivos do caminho que, embora obedeçam a necessidade de passar por todos os pontos do caminho, geram descontinuidades nos perfis de velocidade e aceleração, perdendo a suavidade.

Para garantir essa continuidade, pode-se aplicar uma interpolação polinomial de ordem n, para n+1 pontos do caminho. Essa abordagem, porém, apresenta problemas à medida que a ordem do polinômio aumenta, como comportamento oscilatório, perda de precisão numérica e carga computacional elevada.

Uma solução mais eficiente consiste no emprego de um conjunto de polinômios de ordem pequena, mantendo continuidade nos pontos do caminho, sendo polinômios cúbicos os menores possíveis para essa aplicação. Entre outras possibilidades de definição dos coeficientes dos polinômios, destaca-se a técnica dos splines cúbicos, por garantirem continuidade das acelerações nos pontos dos caminhos.

Para um caminho formado por n+1 pontos, existirão n polinômios cúbicos, um para cada intervalo entre dois pontos. Assim, existirão $4n$ incógnitas, exigindo um mesmo número de condições para a sua solução. Utilizando splines, as condições são:

1. Os valores das funções devem ser iguais nos nós interiores ($2n$-2 condições)

$$\mathbf{a}_{i-1,0} + \mathbf{a}_{i-1,1} t_i + \mathbf{a}_{i-1,2} t_i^2 + \mathbf{a}_{i-1,3} t_i^3 = \mathbf{a}_{i,0} + \mathbf{a}_{i,1} t_i + \mathbf{a}_{i,2} t_i^2 + \mathbf{a}_{i,3} t_i^3 \;, i = 1 \ldots n-1 \qquad (4.10)$$

2. A primeira e a última função devem passar pelas extremidades (2 condições)

$$\mathbf{a}_{0,0} + \mathbf{a}_{0,1} t + \mathbf{a}_{0,2} t^2 + \mathbf{a}_{0,3} t^3 = \mathbf{q}(t_0) \qquad (4.11a)$$

$$\mathbf{a}_{n-1,0} + \mathbf{a}_{n-1,1} t + \mathbf{a}_{n-1,2} t^2 + \mathbf{a}_{n-1,3} t^3 = \mathbf{q}(t_N) \qquad (4.11b)$$

3. As primeiras derivadas nos nós interiores devem ser iguais (n-1 condições)
4. As segundas derivadas nos nós interiores devem ser iguais (n-1 condições)
5. As segundas derivadas das extremidades são iguais a zero (2 condições)

Com essas condições, é possível montar um sistema de equações cuja solução levará à obtenção das constantes das equações polinomiais. Para reduzir o esforço

computacional, foi desenvolvido um método alternativo, que permite reduzir o número de equações a resolver de $4n$ para n-1[19].

Neste método, parte-se do fato da segunda derivada da equação polinomial ser uma reta, podendo ser representada por um polinômio de interpolação de Lagrange de primeira ordem,

$$\ddot{\mathbf{q}}_i(t) = \ddot{\mathbf{q}}_i(t_{i-1})\frac{t-t_i}{t_{i-1}-t_i} + \ddot{\mathbf{q}}_i(t_i)\frac{t-t_{i-1}}{t_i-t_{i-1}} \tag{4.12}$$

onde $t_{i-1} < t < t_i$, e i=0..n.

Integrando (4.12) duas vezes, obtém-se uma equação para $\mathbf{q}_i(t)$,

$$\begin{aligned}\mathbf{q}_i(t) = &\frac{\ddot{\mathbf{q}}(t_{i-1})(t_i-t)^3 + \ddot{\mathbf{q}}(t_i)(t-t_{i-1})^3}{6(t_i-t_{i-1})} \\ &+ \frac{\mathbf{q}(t_{i-1})(t_i-t) + \mathbf{q}(t_i)(t-t_{i-1})}{t_i-t_{i-1}} - \frac{\left(\ddot{\mathbf{q}}(t_{i-1}) + \ddot{\mathbf{q}}(t_i)\right)(t_i-t_{i-1})}{6}\end{aligned} \tag{4.13}$$

que contém duas incógnitas, correspondentes às derivadas segundas da função nos instantes t_{i-1} e t_i. Estas podem ser determinadas pela condição de igualdade das derivadas de (4.13),

$$\begin{aligned}&(t_i-t_{i-1})\ddot{\mathbf{q}}(t_{i-1}) + 2(t_{i+1}-t_{i-1})\ddot{\mathbf{q}}(t_i) + (t_{i+1}-t_i)\ddot{\mathbf{q}}(t_{i+1}) \\ &= \frac{6\left(\mathbf{q}(t_{i+1}) - \mathbf{q}(t_i)\right)}{t_{i+1}-t_i} + \frac{6\left(\mathbf{q}(t_{i-1}) - \mathbf{q}(t_i)\right)}{t_i-t_{i-1}}\end{aligned} \tag{4.14}$$

que, para todos os pontos interiores do caminho, resulta em n-1 equações para n+1 incógnitas. Sendo as acelerações nos pontos extremos do caminho iguais a zero, tem-se as duas condições que faltam para a resolução do sistema de equações[19]. Com as acelerações determinadas, obtém-se facilmente os coeficientes das equações polinomiais pela substituição destes em (4.13).

A classe `TTrjSplines` implementa esse gerador de trajetórias, utilizando em `Init()` o método de determinação das constantes polinomiais acima explicado. Os métodos `Point()` e `Generate()` são funcionalmente equivalentes aos da classe

`TTrjPoly3`, pois ambos utilizam polinômios de terceira ordem para a interpolação dos pontos intermediários. A Figura 4.22 mostra os resultados obtidos com esta classe.

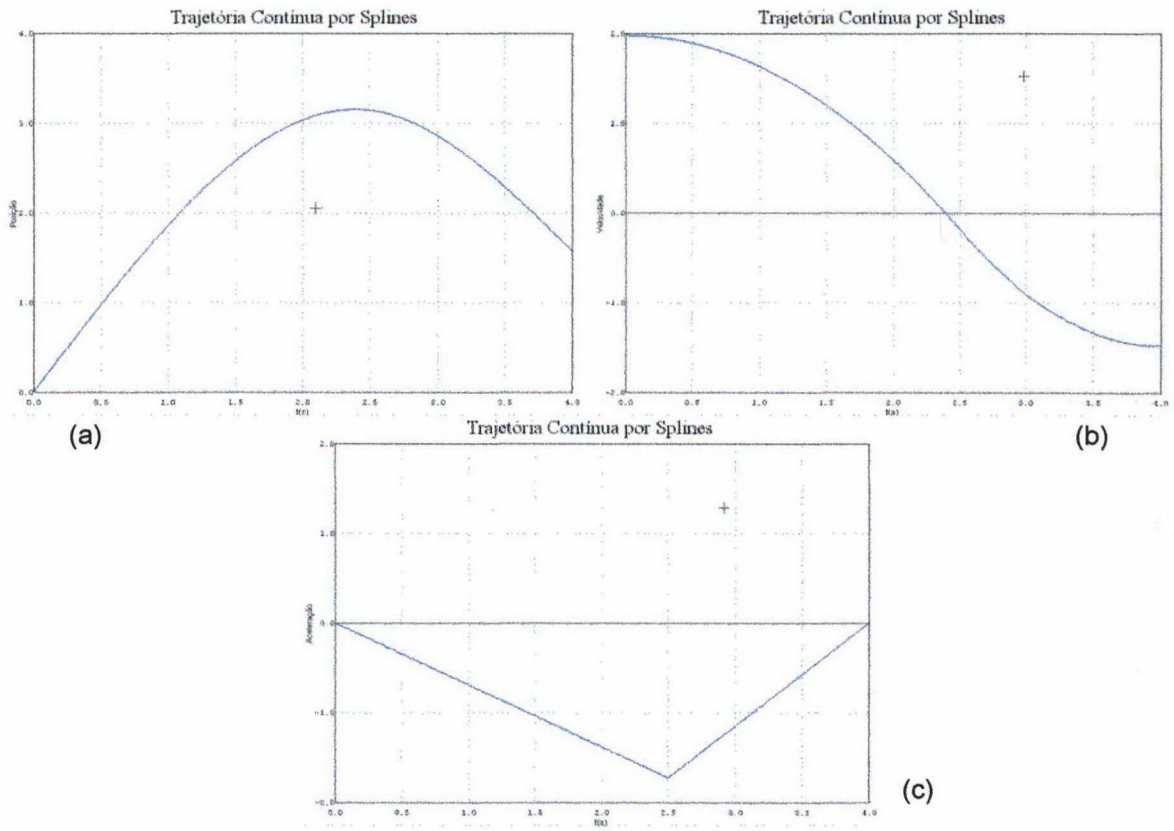


Figura 4.22 (a)Posições, (b)velocidades e (c)acelerações de uma trajetória por splines cúbicos

4.7 VISUALIZAÇÃO DE RESULTADOS

A análise dos resultados obtidos na simulação pode ser feita através de gráficos das variáveis de estado do sistema ao longo do tempo. Como variáveis de interesse, podem ser visualizadas as posições, velocidades, acelerações e torques nas juntas, ou posições, velocidades e acelerações no espaço operacional. Além disso, quando considerado o seguimento de uma trajetória, pode-se desejar analisar o erro de seguimento.

Para facilitar o traçado destes gráficos a partir das matrizes de resultados, foi criado o componente `CrLineGraph`. O objetivo deste componente é traçar automaticamente

gráficos de linha a partir de funções do tipo y=f (x) ou de dados armazenados em matrizes, bastando definir a origem dos dados e os limites de visualização.

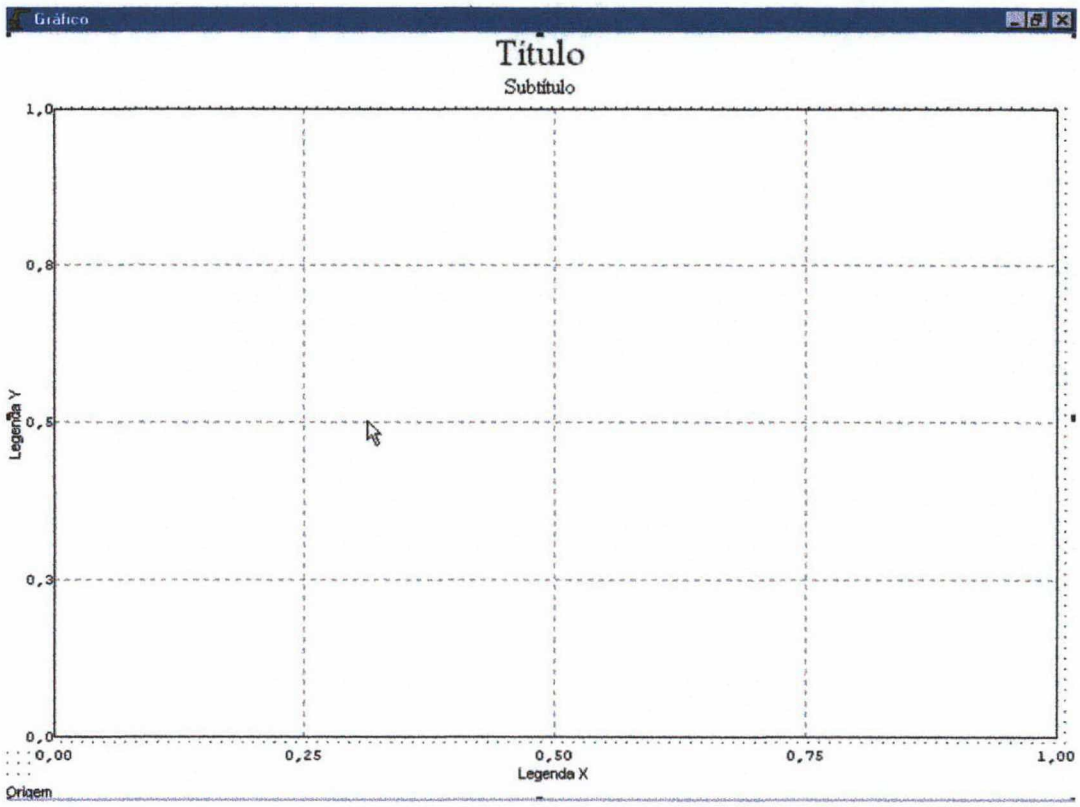


Figura 4.23 Componente `CrLineGraph`

Como qualquer classe derivada da VCL do C++ Builder, a classe TCrLineGraph implementa algumas *propriedades*[49] que podem ser definidas em tempo de projeto através do Object Inspector ou em tempo de execução. Estas possibilitam definir adornos como o título do gráfico, legendas dos eixos horizontal e vertical e suas escalas.

As fontes de dados podem ser adicionadas e retiradas da lista de gráficos a traçar através dos métodos `Add()` e `Del()`. As características das linhas dos gráficos podem ser definidas através do método `Set()`. O método `MoveBy()` permite alterar a ordem dos gráficos na lista, o que resulta na mudança da ordem com que os gráficos são traçados.

[49] Propriedades são extensões da linguagem C++ convencional existente no C++ Builder, correspondendo a abstrações do uso dos métodos de acesso aos atributos de objetos. Uma propriedade se parece e se comporta como um atributo, mas um acesso a ela causa a execução de métodos ocultos (setters/getters) para a sua realização. Como exemplo, uma propriedade ângulo poderia corresponder a um método para retornar um valor de um atributo existente na classe e a um método que, antes de modificar o atributo, verificasse se o valor passado é valido[31].

O eixo horizontal costuma corresponder a uma variável independente. Nesse caso, as propriedades `Initial`, `Final` e `Step` definem a faixa de valores desta variável, que serão usados para determinar as coordenadas verticais correspondentes, se o gráfico corresponder a uma função, ou serão combinados com os valores de uma matriz, se esta for a origem das coordenadas verticais. Pode-se definir uma origem de dados para os valores do eixo horizontal, e nesse caso obtém-se gráficos paramétricos, cujos pontos serão definidos em função da origem dos dados nos dois eixos. Se forem matrizes, os elementos destas corresponderão aos pontos a traçar. Se forem funções, estas serão calculadas para a faixa de valores `Initial-Final`, com passo `Step`, a fim de se determinar os pontos do gráfico. A Figura 4.23 mostra uma instância de `TCrLineGraph` em uma janela, sem terem sido definidas as fontes de dados e as propriedades.

A Figura 4.24 define a classe `TCrLineGraph` e suas componentes. Uma referência desta classe se encontra no Anexo 2 deste trabalho.

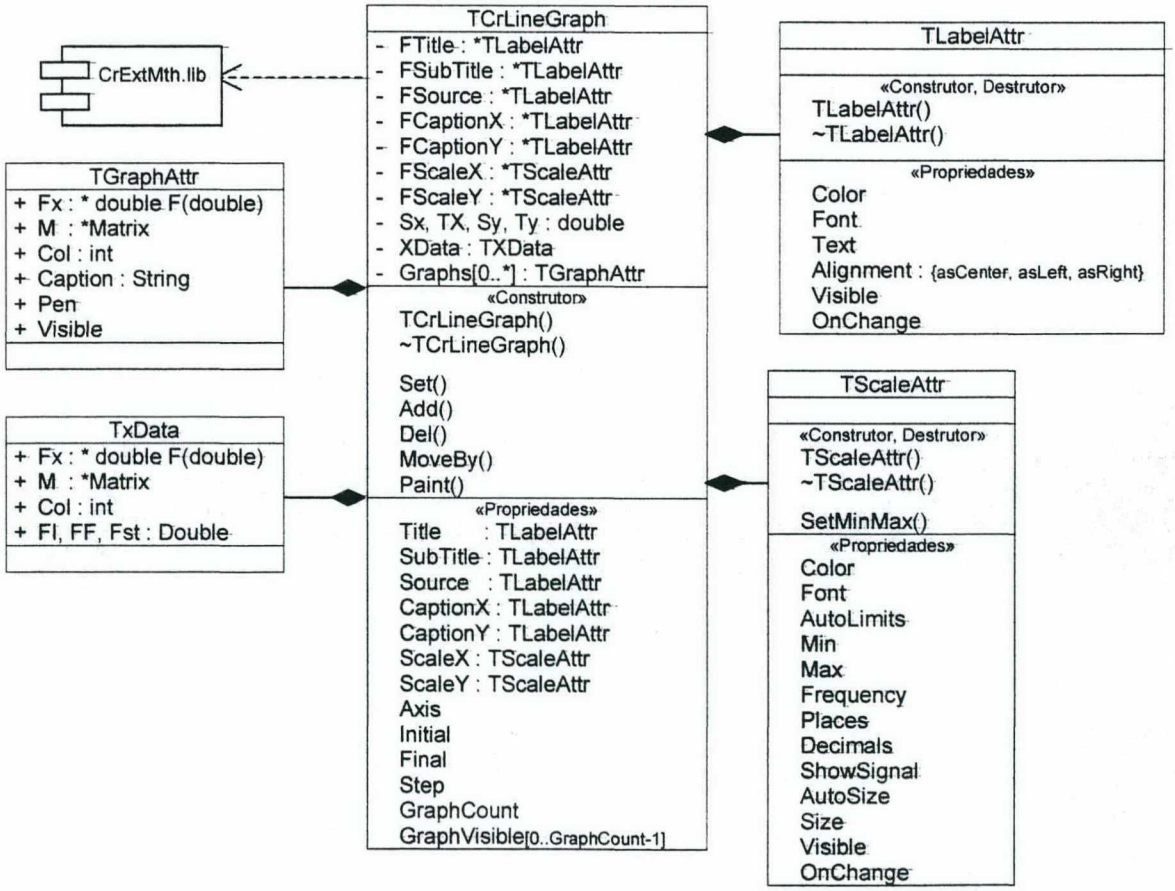


Figura 4.24 Modelo da classe `TCrLineGraph` e suas classes componentes

Outro componente desenvolvido para visualização é o `CrGlCam`. Este implementa uma câmera sintética, que é uma forma de descrever uma câmera posicionada e orientada no espaço 3D. Em relação a outras abordagens de renderização de cenas, esta traz como vantagens a independência da modelagem e a flexibilidade de utilização[13]. Entre outras aplicações, pode-se utilizar a câmera para gerar animações de modelos dos manipuladores executando tarefas em um ambiente virtual ou para traçar gráficos da trajetória no espaço operacional.

Este componente não executa nenhum desenho além do traçado opcional dos eixos do sistema de coordenadas de referência da cena. Os objetos a renderizar devem ser definidos através de métodos externos ao componente, utilizando primitivas da biblioteca OpenGL (a qual serve de base para o componente) ou suas derivadas[50].

A função da câmera implementada é a de renderizar as cenas a partir dos métodos criados pelo usuário, processando as primitivas de desenho que estejam dentro do volume de visão da câmera. Este é definido pela propriedade `Projection`, correspondendo a um paralelepípedo (para projeções ortogonais) ou a um tronco de pirâmide (para projeções em perspectiva). Nesta projeção tem-se o efeito de profundidade, acrescentando realismo às cenas. A Figura 4.25 apresenta os volumes correspondentes aos tipos de projeções.

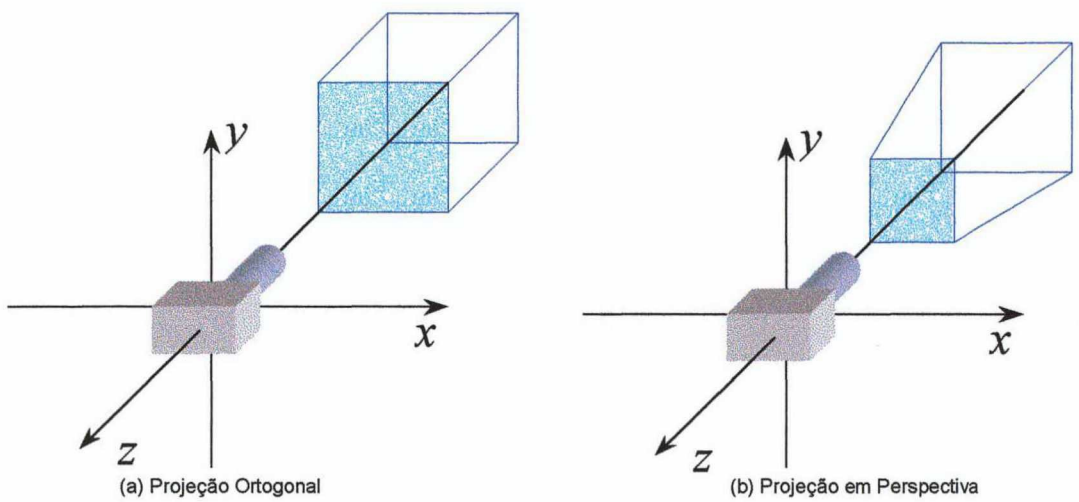


Figura 4.25 Tipos de projeção e seus volumes de visão

[50] Como exemplo, tem-se os métodos `Draw()` e `DrawLines()` da classe `TManipulator`.

Para aumentar o efeito de realismo, pode-se utilizar recursos de iluminação. Se ativada, através da propriedade `Lighting`, a renderização de cenas utiliza as definições da luz ambiente e dos focos de luz ativos (através da propriedade `Light[i]`, onde `i` identifica o foco de luz) na determinação das cores dos objetos. As definições de cores e efeitos das luzes são acessadas através dos métodos `SetLight()` e `GetLight()`. As posições, direções, focos e tipos de atenuação das fontes de luz são acessadas pelos métodos `SetLightPosition()`, `SetLightDirection()`, `SetSpotLight()`.

A posição e orientação da câmera pode ser definida através do método `Move()` ou pelas propriedades `Position` (e suas componentes `X`, `Y` e `Z`) e `Orientation` (e suas componentes `Azimuth` - giro em torno do eixo `Z` da câmera, `Elevation` - giro em torno do eixo `Y` da câmera, e `Inclination` - giro em torno do eixo `X` da câmera, equivalente a um "giro de cabeça", sem alterar a direção da câmera).

Para utilizar uma câmera sintética, pode-se ativá-la através do método `BeginPaint()`. Após, podem ser chamados os métodos responsáveis pelo desenho da cena a renderizar. Ao final, deve-se executar o método `EndPaint()`, para que a cena seja mostrada. Para atualização automática do desenho a cada modificação das características da câmera (como mudança na posição desta ou da projeção empregada), pode-se vincular o método de desenho ao evento `OnPaint`.

A classe `TCrGlCam` e suas componentes são modeladas na Figura 4.26. O Anexo 2 deste trabalho apresenta uma referência completa para uso deste componente.

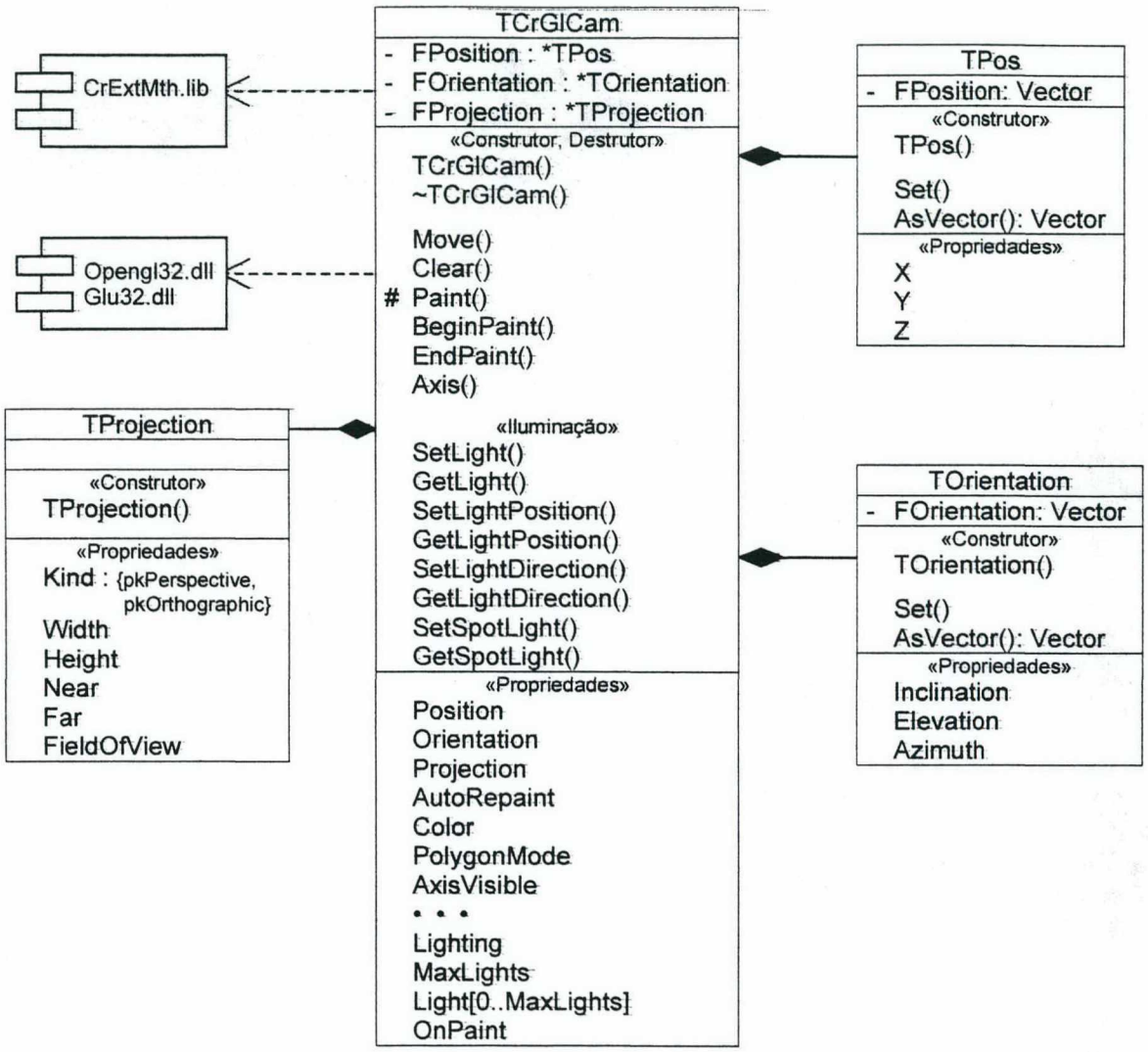


Figura 4.26 Modelo da classe `TCrGlCam` e suas classes componentes

4.8 INTERFACE COM O USUÁRIO

Para o usuário, a interface é um fator relevante para a adoção e uso de um *software*. Um exemplo de tal importância é a aceitação do Microsoft Windows como ambiente operacional, por oferecer uma interface consistente às aplicações, através do conceito de janelas gráficas independentes do *hardware*[51]. O Borland C++ Builder, utilizado no desenvolvimento do simulador, oferece componentes adicionais aos já existentes no Windows, além de permitir a criação de outros componentes, de acordo com a necessidade. Esse foi o caso dos componentes `CrLineGraph` e `CrGlCam`, além de outros componentes auxiliares à interface.

O simulador foi organizado em uma série de janelas independentes, onde cada uma corresponde a um dos elementos do sistema a simular.

A janela principal do simulador corresponde à definição das características gerais da simulação. Nesta pode-se atribuir um nome e escrever uma descrição da simulação, além de serem configuradas características como método de solução do sistema de equações, passo de integração, passo de amostragem e gerador de trajetórias empregado, como mostra a Figura 4.27.

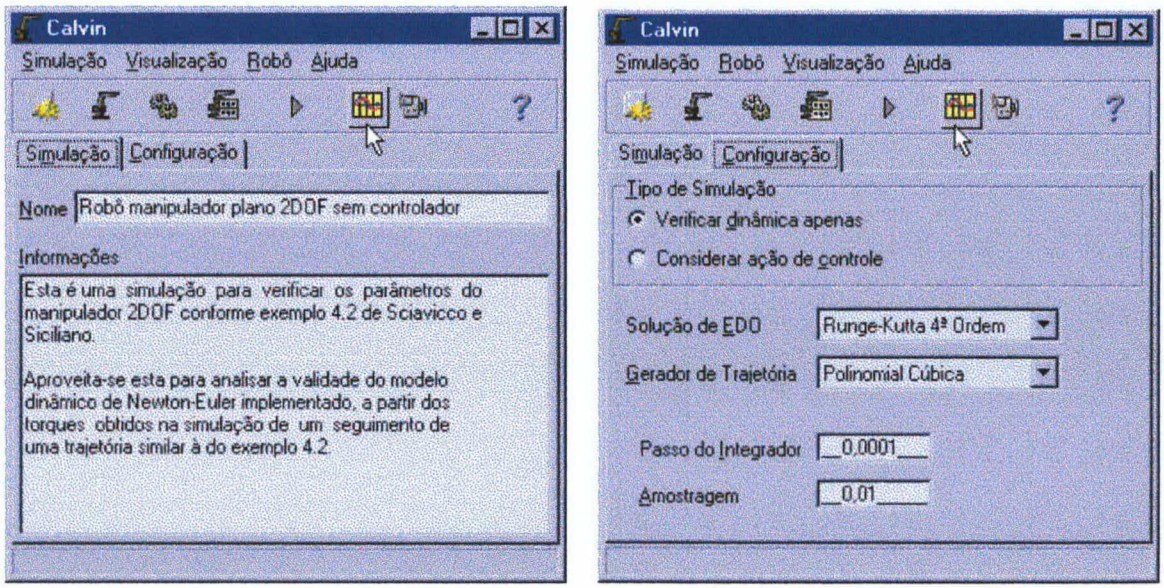


Figura 4.27 Janela principal do simulador

[51] Embora existam outros sistemas operacionais com interfaces gráficas amigáveis e consistentes, o Windows é uma das mais usadas para a plataforma IBM-PC, que corresponde à grande maioria dos hardware utilizados. Tal uso deve-se ao fato desta ser uma das mais antigas interfaces gráficas para este hardware.

O menu principal da aplicação está na janela principal. Este é sensível ao contexto, procurando fazer o usuário seguir uma ordem de definição do sistema, que seria

Simulação → Manipulador → Controlador → Tarefa → Execução → Resultados

onde o controlador pode ser omitido, se for desejado apenas verificar as características dinâmicas do manipulador. As opções do menu aparecem de acordo com essa ordem.

A próxima janela a se considerar é a do manipulador. Esta apresenta, para cada elo, seus parâmetros cinemáticos e dinâmicos, além dos limites das juntas. Todas estas características podem ser modificadas, e graus de liberdade podem ser adicionados ou removidos do modelo do manipulador. Também podem ser ativadas ou desativadas características gerais, como consideração de atrito e da inércia do rotor na dinâmica, e verificação dos limites das juntas nas operações do manipulador. Não estão disponíveis recursos para definição do corpo do manipulador, devendo esta ser feita, se desejado, através da edição do arquivo de modelagem do manipulador. A Figura 4.28 apresenta essa janela.

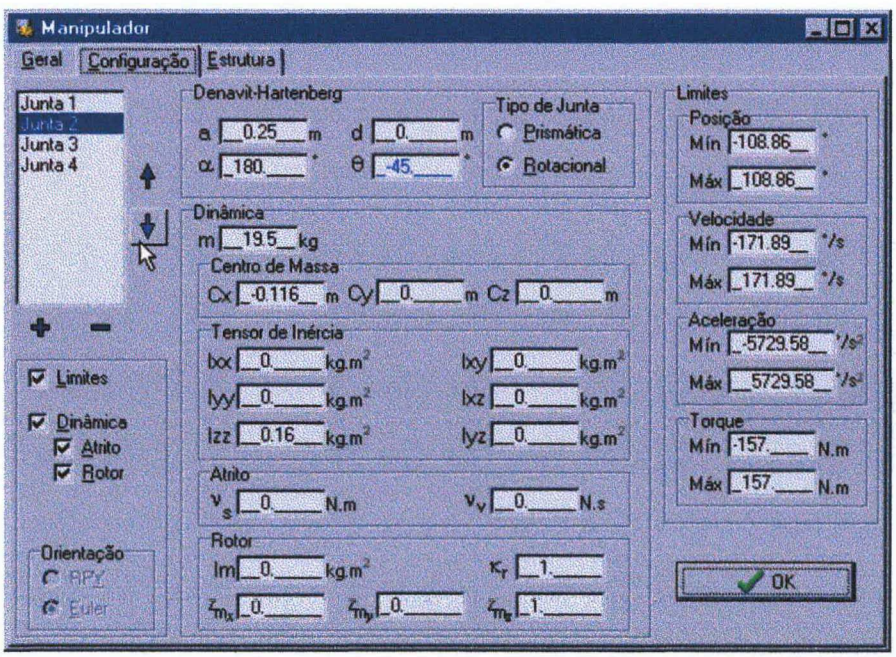


Figura 4.28 Janela de dados do manipulador

O controlador é um dos elementos mais difíceis de gerenciar, devido à flexibilidade de implementação possibilitada ao usuário. Na janela correspondente a esse elemento, a primeira definição é a da biblioteca de ligação dinâmica que o implementa. Uma vez ligada ao simulador, determinam-se os seus parâmetros configuráveis[52], e a janela é reconfigurada para permitir o acesso destes ao usuário, conforme mostra a Figura 4.29.

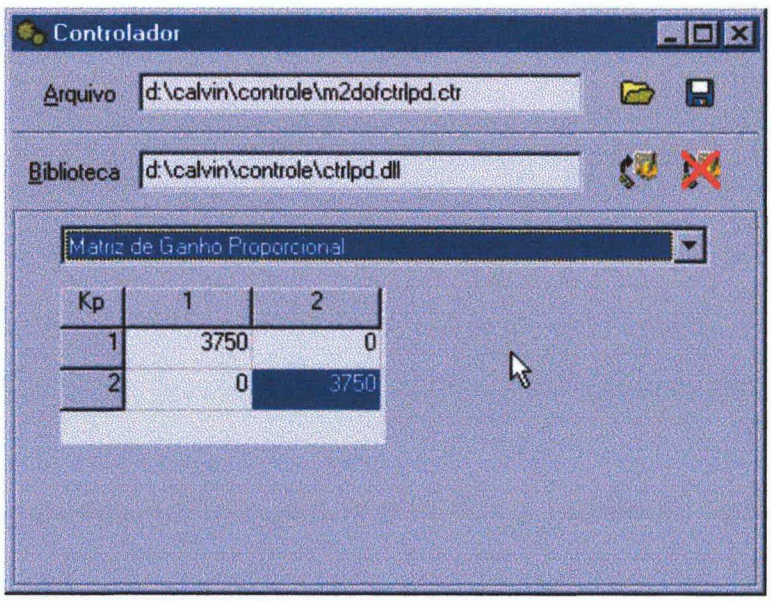


Figura 4.29 Janela de configuração do controlador

A janela de definição da tarefa a executar utiliza-se de tabelas para especificação das posições a serem alcançadas pelo manipulador. Nestas, cada linha corresponde a uma posição, e as colunas são as componentes da posição em um dado instante. Estas podem ser variáveis das juntas ou do espaço operacional, de acordo com a natureza da tarefa, também definida nesta janela. Se a tarefa for um caminho, a cada ponto deverá ser definido o instante em que este ocorre, aparecendo ao lado da Tabela de caminho uma Tabela correspondente de instantes. Se a tarefa for uma trajetória, são definidos o intervalo de tempo desta e um passo, e todos os pontos ocorrem em intervalos fixos. A Figura 4.30 ilustra essa janela em uma definição de caminho no espaço das juntas.

[52] Através dos métodos `ParamNum()`, `ParamSize()`, `ParamName()` e `ParamHelp()` do controlador. Esta abordagem só é válida para os controladores cujos parâmetros configuráveis sejam valores numéricos.

Definidos os elementos da simulação, sua execução é solicitada no menu da janela principal. O status da simulação é mostrado nesta janela, assim como avisos de erros que possam ocorrer durante o processo de solução do sistema.

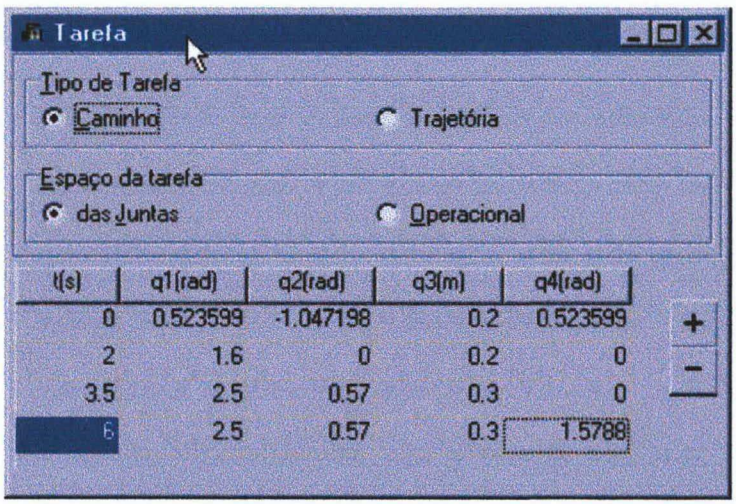


Figura 4.30 Janela de especificação de tarefas

Os resultados de uma simulação bem sucedida podem então ser analisados nas janelas de gráficos. Se o sistema considerado for controlado, são mostradas tanto a trajetória a ser seguida quanto o resultado obtido no mesmo gráfico. As janelas são configuráveis, permitindo escolher o espaço de representação dos resultados (espaço das juntas ou operacional). No caso do espaço das juntas, os torques exercidos nas juntas são traçados, como ilustra a Figura 4.31.

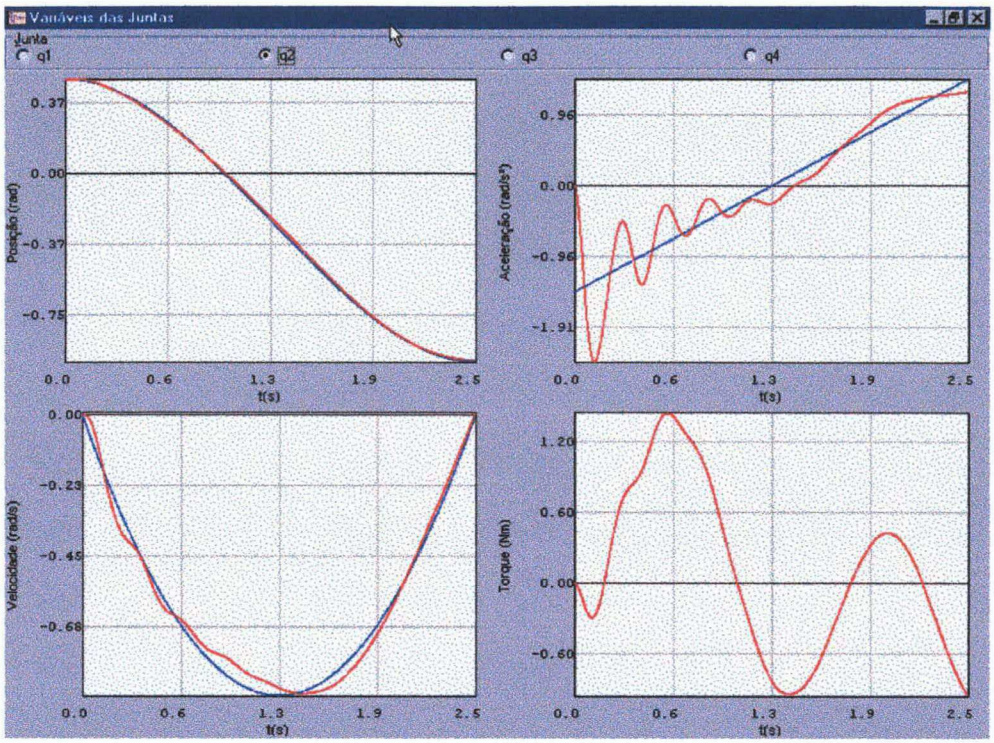


Figura 4.31 Janela de gráficos das variáveis das juntas ao longo da execução da tarefa

Pelo menos uma janela de visualização do ambiente virtual onde a tarefa é executada deve estar disponível no início de uma simulação. Essa consiste em uma câmera sintética, que pode ser controlada pelo teclado numérico ou pelo painel de controle criado especificamente para esse fim. Na câmera, mostra-se uma sala virtual, onde pode-se desenhar a estrutura do manipulador (linhas representando os elos) ou seu corpo, se este foi modelado. A Figura 4.32 mostra um exemplo de visualização desta janela.

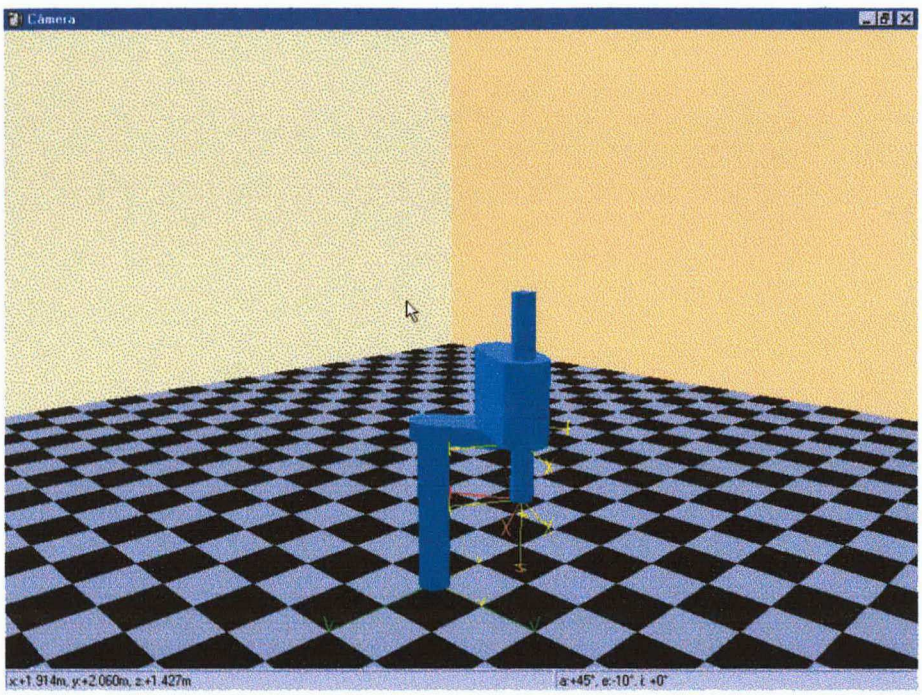


Figura 4.32 Janela de visualização do ambiente virtual do manipulador

O usuário pode criar várias janelas de câmera, cada qual com um ponto de observação e características de visualização distintas. Estas são definidas tanto pelo painel de controle como pela janela de configuração, que permite ainda definir as características de iluminação, como luz ambiente e fontes de luz. O cenário também pode ter suas características definidas nesta janela, como mostra a Figura 4.33.

Todas as configurações podem ser salvas através do menu principal. Deve-se observar que as definições dos manipuladores, do controladores e das tarefas são independentes entre si, o que permite criar várias configurações do sistema a simular. Na simulação também são salvas as características das câmeras.

Figura 4.33 Janela de configuração da câmera e ambiente

4.9 CONCLUSÃO

Este capítulo tratou do desenvolvimento do simulador de robôs manipuladores, objeto deste trabalho. Iniciou-se pela análise do problema de projeto, consistindo na especificação dos requisitos funcionais do *software* e determinação da metodologia e das ferramentas de desenvolvimento. Após uma visão geral do simulador, foram apresentadas classes de entidades matemáticas fundamentais para o desenvolvimento do simulador, como matrizes, vetores e *solvers* de equações diferenciais ordinárias, seguidas das classes que modelam manipuladores, controladores e geradores de trajetórias. Foram mostradas, também, classes de ferramentas de visualização, como traçadores de gráficos e câmeras sintéticas, estas fundamentais para a visualização de um ambiente virtual onde as tarefas são executadas.

Por fim, foi apresentada a interface do *software* com o usuário, cujo projeto deve levar em conta a facilidade de uso, a visualização mais eficiente das informações e possibilidades de configuração dos elementos da simulação.

5. RESULTADOS

Neste capítulo são apresentados os resultados obtidos com o uso do simulador na modelagem cinemática e dinâmica de alguns manipuladores. Para a validação do funcionamento do simulador, realiza-se a modelagem cinemática e dinâmica de um manipulador plano de dois graus de liberdade, simulando a execução de algumas tarefas no espaço das juntas e espaço operacional empregando diferentes tipos de geradores de trajetória e controle. A seguir, utilizam-se os parâmetros de um manipulador tipo SCARA para a modelagem e simulação da execução de tarefas.

5.1 VALIDAÇÃO DO SIMULADOR

O manipulador plano de dois graus de liberdade é um tipo de robô bastante encontrado na literatura, utilizado tanto para exemplificar a modelagem cinemática e dinâmica de manipuladores quanto para estudo de caso de implementação de técnicas de controle[1,2,6,7,11].

Assim, para verificar o correto funcionamento do simulador, optou-se por este tipo de manipulador, para possibilitar a comparação com resultados já existentes. Os parâmetros cinemáticos e dinâmicos foram retirados de SCIAVICCO e SICILIANO[2], e são reproduzidos nas Tabelas 5.1, 5.2 e 5.3.

Tabela 5.1 Parâmetros Denavit-Hartenberg do manipulador plano de dois graus de liberdade

Elo	a_i	α_i	d_i	θ_i
1	1.0	0.0	0.0	q_1
2	1.0	0.0	0.0	q_2

Na Tabela acima, os parâmetros sombreados correspondem às variáveis das juntas. Os valores de a_i e d_i são informados em metros e os valores de α_i e θ_i são informados em radianos[52].

Tabela 5.2 Parâmetros dinâmicos do manipulador plano de dois graus de liberdade

Elo	m_i	$\mathbf{l}_{ci}$	Ixx_i	Iyy_i	Izz_i	Ixy_i	Ixz_i	Iyz_i
1	50.0	-0.5,0.0,0.0	0.0	0.0	10.0	0.0	0.0	0.0
2	50.0	-0.5,0.0,0.0	0.0	0.0	10.0	0.0	0.0	0.0

Nos parâmetros dinâmicos, a massa m_i está em kg, as componentes do vetor centro de massa $\mathbf{l}_{ci}$ estão em metros e as componentes do tensor de inércia $\mathbf{I}_i$ estão em kg·m^2.

Tabela 5.3 Atrito e dinâmica do atuador no manipulador plano de dois graus de liberdade

Junta / Atuador	v_{si}	v_{vi}	I_{mi}	k_{ri}	$\mathbf{z}_{mi}$
1	0.0	0.0	0.01	100.0	0.0,0.0,1.0
2	0.0	0.0	0.01	100.0	0.0,0.0,1.0

Na Tabela 5.3, o coeficiente de atrito estático v_s está em N, o coeficiente de atrito dinâmico v_v está em N·s/rad ou N·s/m, dependendo do tipo de atuador, a inércia do rotor I_i é informada em kg·m^2, a relação de transmissão k_{ri} é adimensional e os componentes do vetor eixo do rotor $\mathbf{z}_{mi}$ estão em metros.

Os parâmetros cinemáticos e dinâmicos são utilizados para a criação de um arquivo de descrição do robô, que é carregado pelo simulador para realizar a modelagem do manipulador. O arquivo de modelo de robô é listado no Anexo 3.

5.1.1 Verificação da Modelagem Cinemática

Para verificar a modelagem cinemática, foi realizada uma simulação onde a entrada consistia em um caminho no espaço operacional, formando um círculo de raio 0.25m e

[52] As unidades utilizadas nas Tabelas 5.1, 5.2 e 5.3 são adotadas nas demais Tabelas de parâmetros cinemáticos e dinâmicos deste capítulo, além de serem empregadas no simulador.

centro em (1.0m,0.25m). Este caminho foi definido por 16 pontos igualmente espaçados, devendo ser realizado em um intervalo de 16s.

A simulação utilizou o algoritmo de inversão cinemática implementado no modelo do manipulador para obter os valores das variáveis das juntas correspondentes aos pontos do caminho. Com estes valores, a trajetória foi gerada com o uso de um algoritmo de splines cúbicos, a fim de manter continuidade de velocidades e acelerações ao longo do caminho, com passo igual a 0.02.

O resultado da simulação foi a execução de uma animação, onde o modelo do manipulador era desenhado por uma câmera sintética após a definição das variáveis das juntas, em cada instante da tarefa. A posição do efetuador final ao longo do tempo foi traçada na animação. O instante final da animação, com a trajetória completa, é mostrado na Figura 5.1.

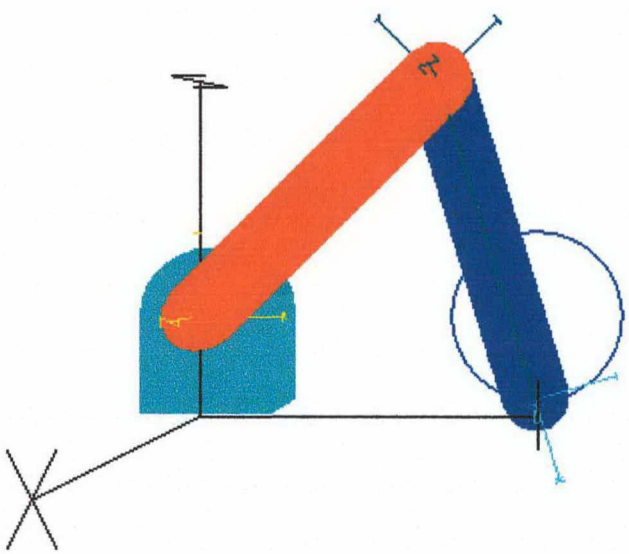


Figura 5.1 Visualização do modelo do manipulador plano de dois graus de liberdade através de uma câmera sintética, após a execução de uma trajetória circular no espaço operacional

Os resultados obtidos na simulação são análogos aos obtidos pelas equações da cinemática direta para este tipo de manipulador, o que valida o modelo cinemático implementado.

Observa-se também o correto funcionamento do componente de câmera sintética, utilizado para a animação do modelo do manipulador. Este, para ser desenhado, também depende do correto funcionamento da cinemática direta, o que serve como auxílio à comprovação do modelo cinemático.

5.1.2 VERIFICAÇÃO DA MODELAGEM DINÂMICA

Para validar o modelo dinâmico, foi realizada a simulação de uma tarefa no espaço das juntas descrita em SCIAVICCO e SICILIANO[2], que consiste na determinação dos torques resultantes do seguimento de uma trajetória no espaço das juntas.

A entrada é formada pelos pontos inicial e final desejados para o manipulador. A configuração inicial é $\mathbf{q} = [-1.47063 \quad 2.94126]^T$, equivalente a uma posição do efetuador final $\mathbf{x} = [0.2 \quad 0.0]^T$. Ambas as juntas realizam uma rotação de $\pi/2$ rad em 0.5s, com perfil de velocidade triangular. O histórico das variáveis das juntas é mostrado na Figura 5.2.

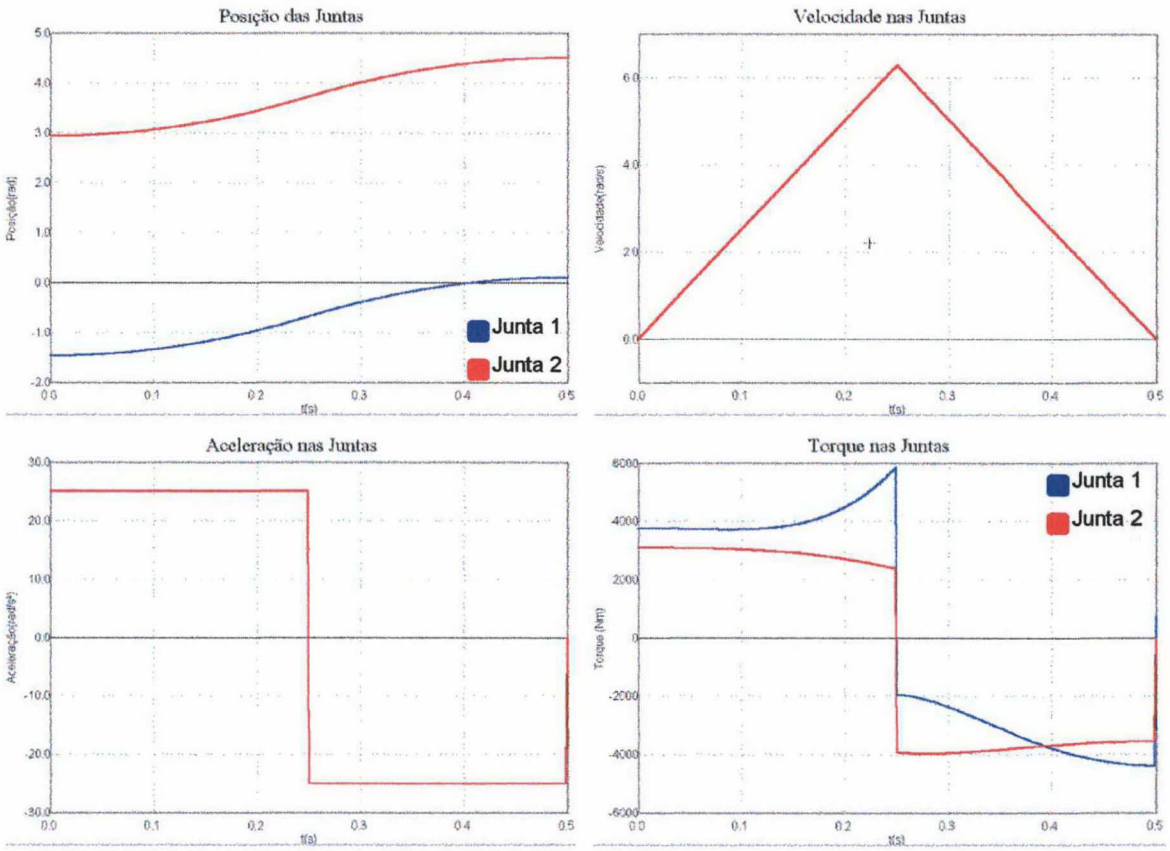


Figura 5.2 Posições, velocidades, acelerações nas juntas e torques resultantes no movimento especificado para o manipulador em uma trajetória de perfil de velocidade triangular

Os resultados obtidos são equivalentes aos apresentados na literatura, o que verifica a funcionalidade do modelo dinâmico do manipulador e dos geradores de trajetória implementados.

5.1.3 Simulação de Tarefas em Malha Fechada

O passo seguinte para verificar o funcionamento do simulador consistiu na execução de tarefas empregando algoritmos de controle em malha fechada.

A primeira tarefa simulada foi mover o efetuador final da posição $\mathbf{x} = [0.2 \;\; 0.0]^T$ a posição $\mathbf{x} = [1.8 \;\; 0.0]^T$ em um tempo de 3s, considerando os efeitos do tipo de geração de trajetória e de controle empregados. Foram utilizadas trajetórias de perfil de velocidade trapezoidal e trajetórias polinomiais cúbicas, e controladores tipo PD e PD com compensação de gravidade.

Para a resolução do sistema, foi empregado o método de Runge-Kutta de 5ª ordem, com passo de integração 0.001 e passo de amostragem 0.01. Os geradores de trajetórias utilizaram um passo igual ao passo de amostragem, sendo que, no caso do perfil trapezoidal, foi considerado um tempo de aceleração de 0.86s e velocidade máxima $\dot{\mathbf{q}}=[0.39859, \; -0.95114]^T$.

A Figura 5.3 mostra o resultado da simulação empregando um perfil de velocidade trapezoidal, com um controlador PD de ganhos $\mathbf{K}_p=3750\mathbf{I}$ e $\mathbf{K}_d=300\mathbf{I}$.

Para melhorar a performance do sistema, empregou-se um controlador PD com compensação de gravidade, desenvolvido especificamente para este modelo de manipulador, com os mesmos ganhos utilizados no controlador PD anterior. A Figura 5.4 mostra os resultados obtidos.

A Figura 5.5 mostra os resultados de simulações onde a trajetória é polinomial cúbica, utilizando-se o controlador PD com compensação de gravidade.

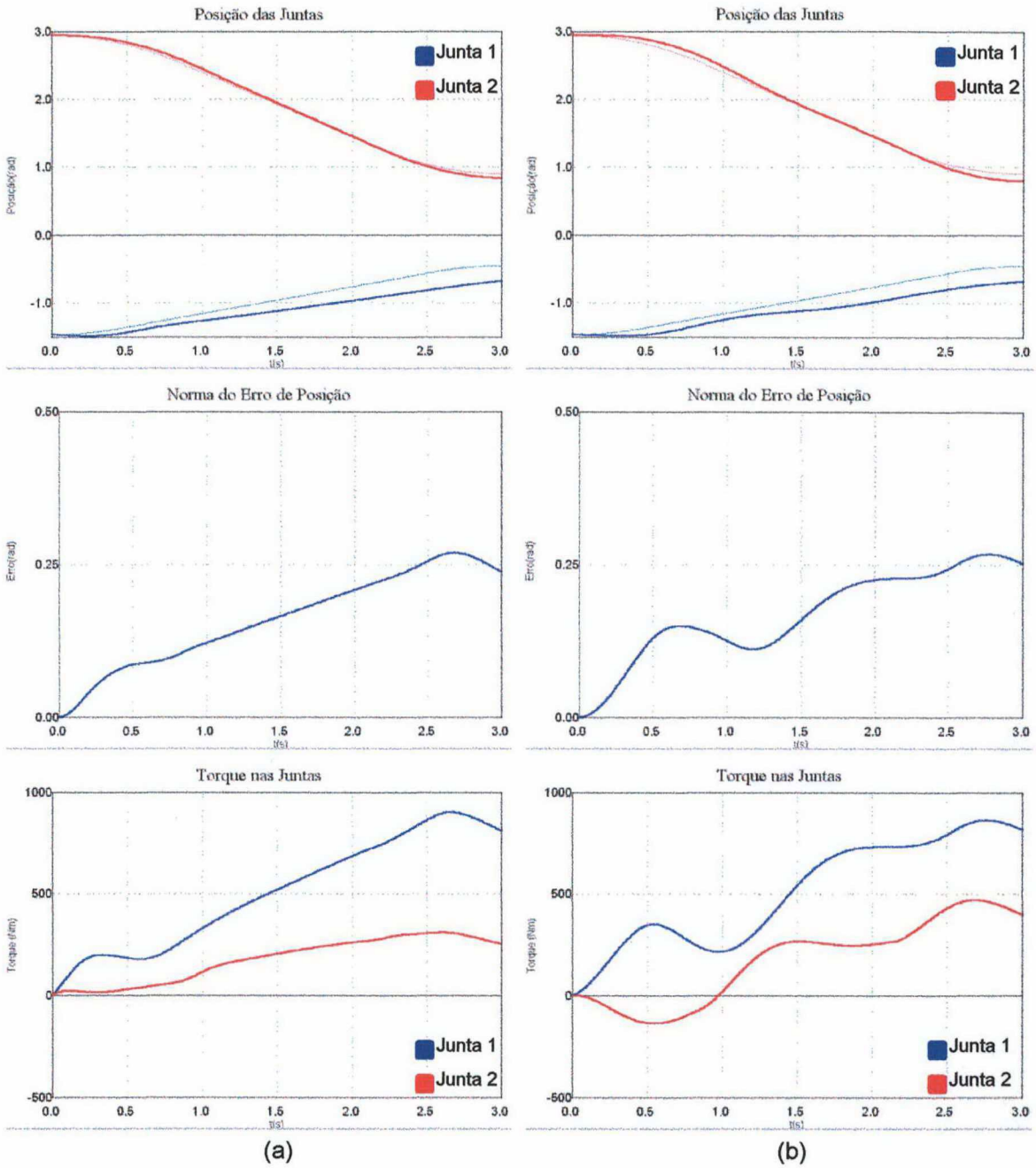


Figura 5.3 Histórico de posições, norma dos erros de posições e torques das juntas em uma tarefa de seguimento de trajetória com perfis de velocidade trapezoidais, utilizando um controlador PD (a) não considerando a inércia dos atuadores e (b) considerando a inércia dos atuadores[53]

[53] Nos gráficos de posição, as curvas de cores mais fracas representam as referências de trajetória.

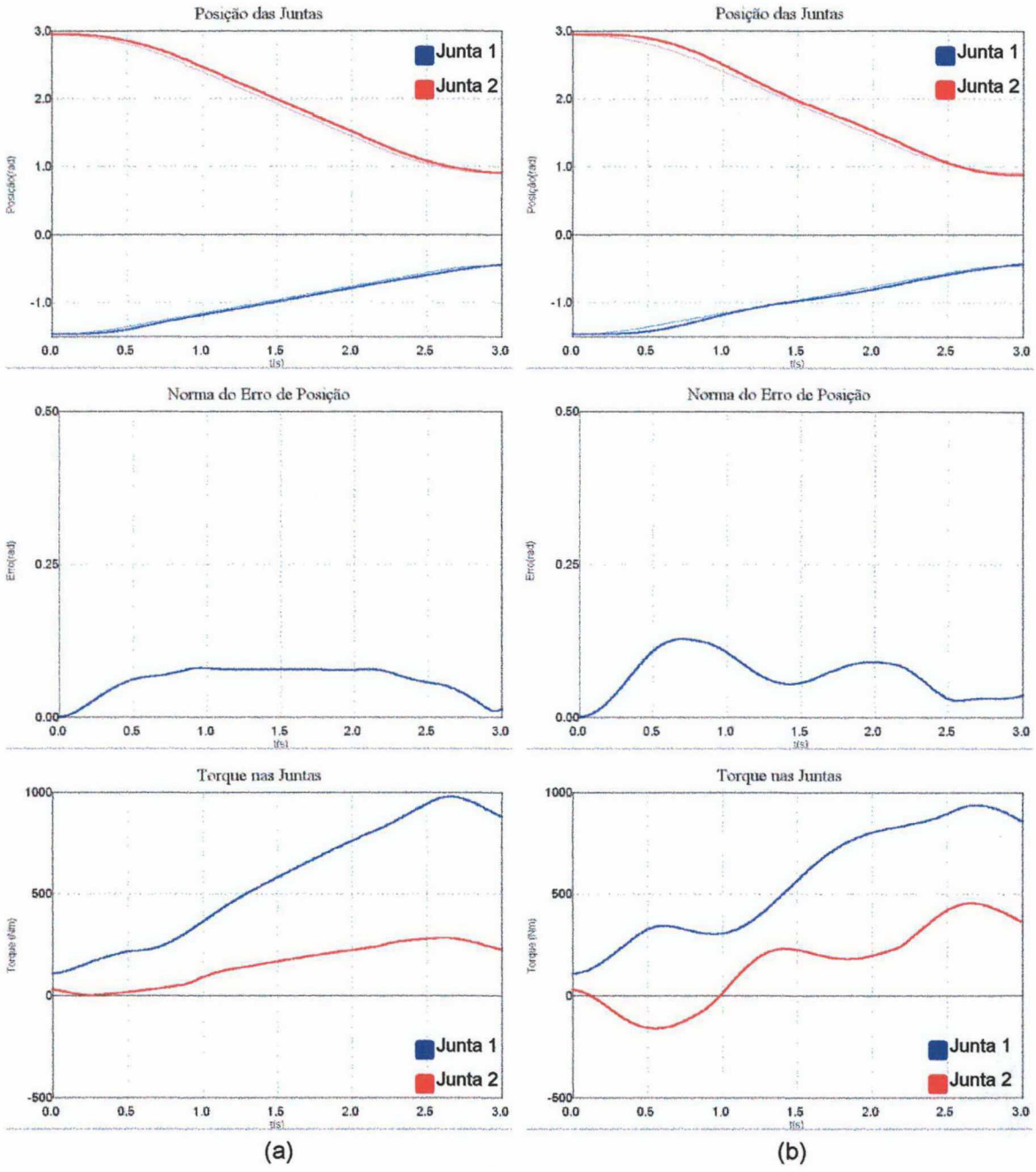


Figura 5.4 Histórico de posições, norma dos erros de posições e torques das juntas em uma tarefa de seguimento de trajetória com perfis de velocidade trapezoidais, utilizando um controlador PD com compensação de gravidade (a) não considerando a inércia dos atuadores e (b) considerando a inércia dos atuadores

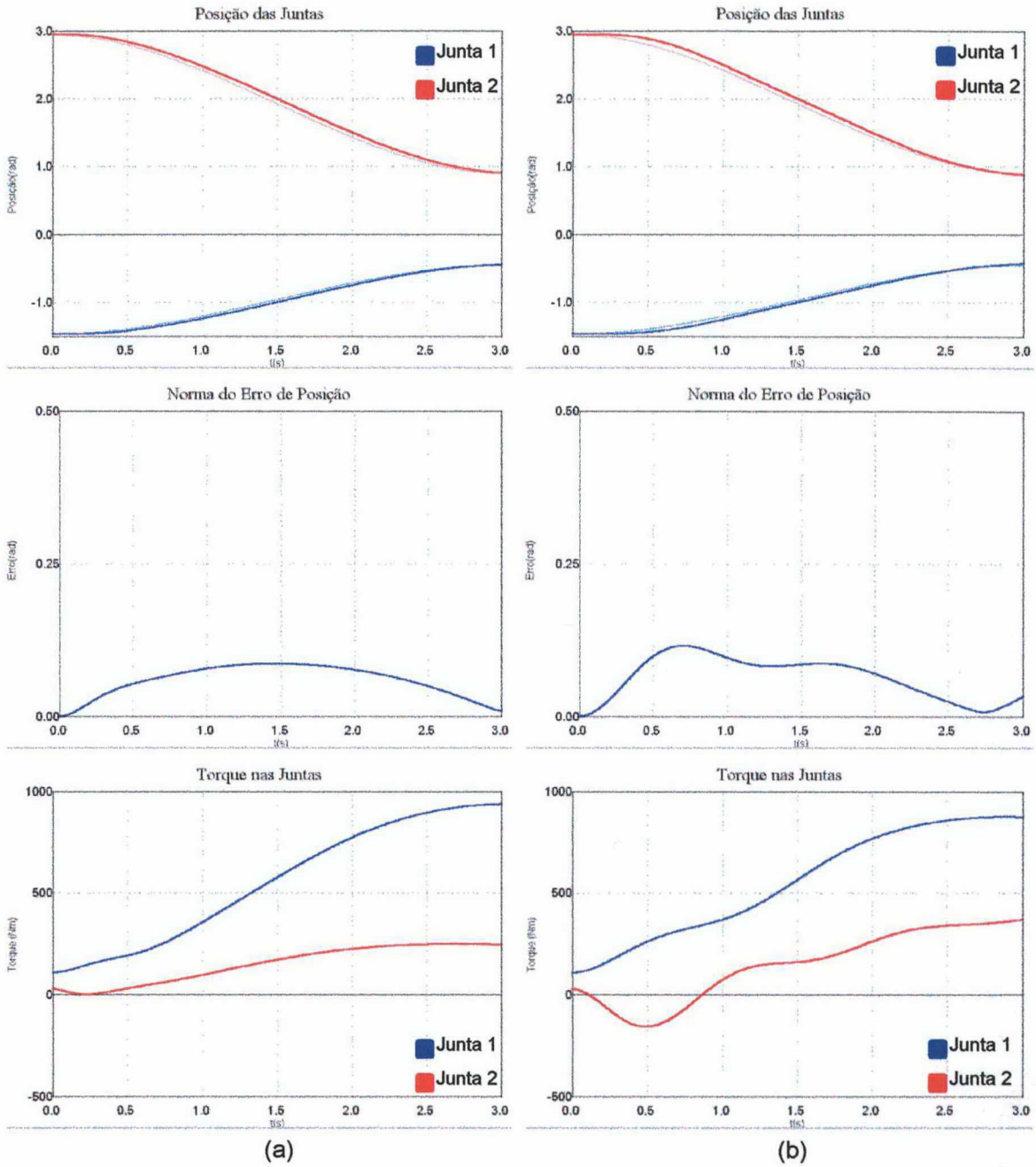


Figura 5.5 Histórico de posições, norma dos erros de posições e torques das juntas em uma tarefa de seguimento de trajetória com perfis polinomiais, utilizando um controlador PD com compensação de gravidade (a) não considerando a inércia dos atuadores e (b) considerando a inércia dos atuadores

A segunda tarefa de controle em malha fechada especificada foi o seguimento da trajetória circular utilizada na validação da modelagem cinemática. O controlador utilizado foi o PD com compensação de gravidade, com ganhos $\mathbf{K}_p$=3750**I** e $\mathbf{K}_d$=300**I**. A resolução do sistema foi feita pelo método de Runge-Kutta de 4ª ordem, com passo de integração 0.001 e passo de amostragem 0.02. Os resultados são mostrados na Figura 5.6.

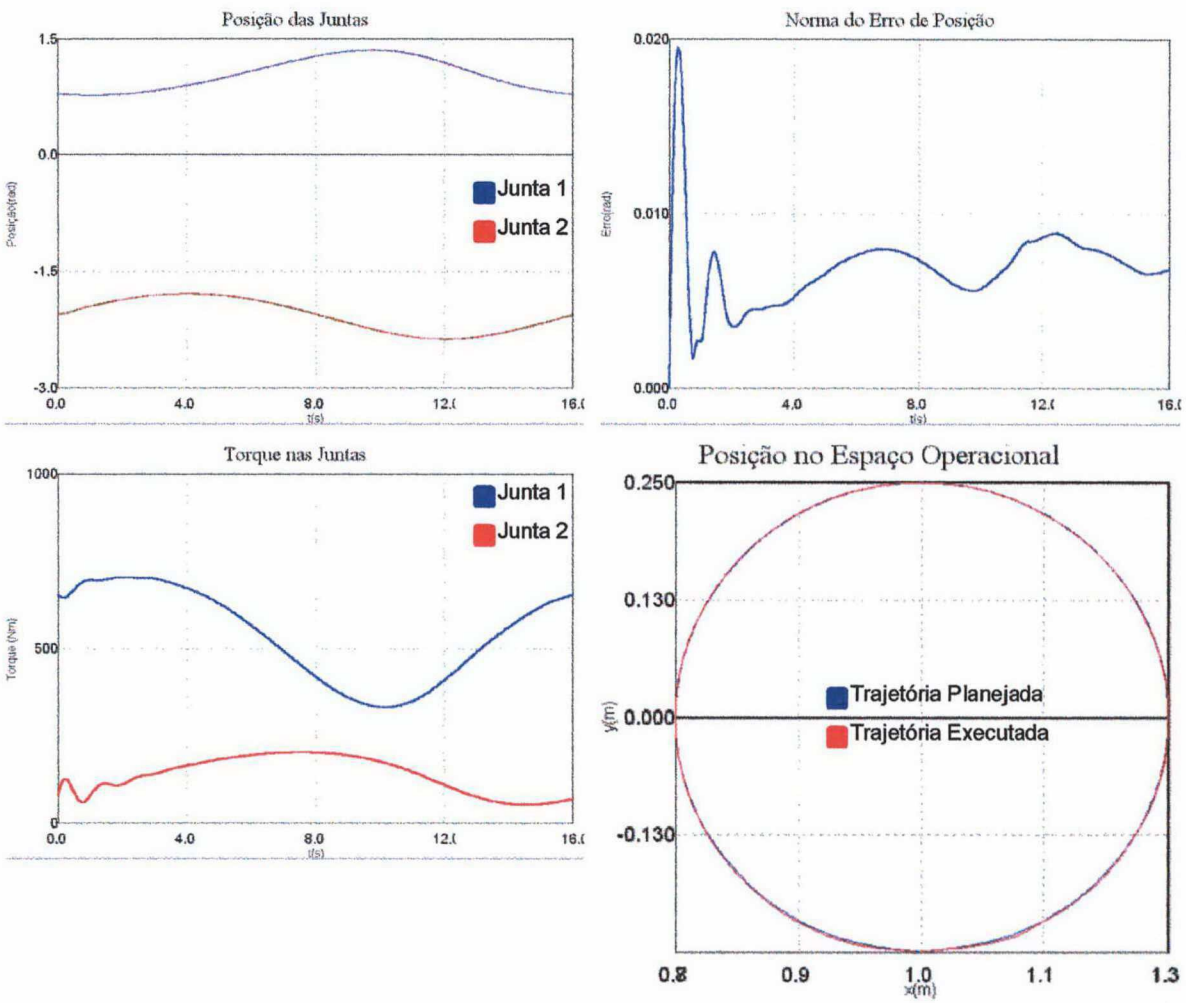


Figura 5.6 Histórico de posições, torques e norma dos erros de posições das juntas no seguimento de uma trajetória circular no espaço operacional, utilizando um controlador PD com compensação de gravidade

5.2 MANIPULADOR SCARA

A fim de utilizar o simulador para modelar um robô real, optou-se por adotar a configuração e os parâmetros do manipulador Inter, instalado no Laboratório de Robótica de uso comum aos Departamentos de Engenharia Mecânica e de Automação e Sistemas da Universidade Federal de Santa Catarina. Este é um robô do tipo SCARA (Selective Compliant Articulated Robot for Assembly), comumente empregado na indústria, e foi construído no Instituto de Robótica da Universidade Técnica de Zurique.

Os parâmetros cinemáticos e dinâmicos utilizados na modelagem foram extraídos de GOLIN, GUENTHER e WEIHMANN[33], e são reproduzidos nas Tabelas 5.4 e 5.5. O arranjo cinemático do robô, de quatro graus de liberdade, é mostrado na Figura 5.7.

Tabela 5.4 Parâmetros Denavit-Hartenberg do manipulador Inter

Elo	A_i	α_i	d_i	θ_i
1	0.25	0.0	0.665	q_1
2	0.25	180.0	0.0	q_2
3	0.0	0.0	q_3	0.0
4	0.0	0.0	0.0	q_4

Tabela 5.5 Parâmetros dinâmicos do manipulador Inter

Elo	m_i	$\mathbf{l}_{ci}$	Ixx_i	Iyy_i	Izz_i	Ixy_i	Ixz_i	Iyz_i
1	11.4	-0.118,0.0,0.0	0.0	0.0	0.23	0.0	0.0	0.0
2	19.5	-0.116,0.0,0.0	0.0	0.0	0.16	0.0	0.0	0.0
3	2.0	0.0,0.0,0.0	0.0	0.0	0.1	0.0	0.0	0.0
4	1.5	0.0,0.0,0.0	0.0	0.0	0.1	0.0	0.0	0.0

A tarefa a executar na simulação é definida no espaço operacional, correspondendo ao seguimento de uma trajetória circular no plano xy de raio igual a 0.10m e centro (0.35m,0.05m) em um período de 8s, com mudança gradual de altura, da ordem de 0.05m, na metade da circunferência (entre 2s e 6s). A entrada da tarefa é formada por um caminho de 16 pontos igualmente espaçados na circunferência.

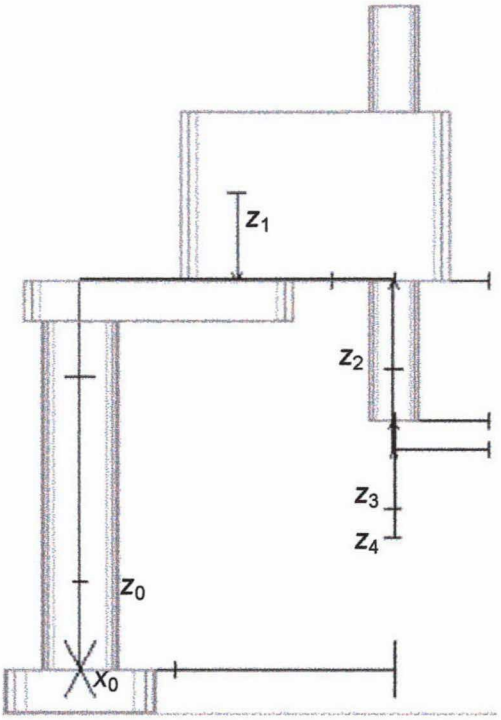


Figura 5.7 Estrutura do manipulador Inter (modelo SCARA)

Após a obtenção das configurações das juntas correspondentes aos pontos do caminho no espaço operacional pela cinemática inversa, gera-se a trajetória através de um algoritmo de splines cúbicos, com passo 0.02. A Figura 5.8, gerada através da câmera sintética, ilustra em verde a trajetória a ser seguida pelo efetuador final.

Para controlar o manipulador, foi implementado um algoritmo PD com compensação de gravidade, com ganhos $\mathbf{K}_p$= diag{750,750,500,750} e $\mathbf{K}_d$=10**I**. A resolução do sistema foi feita pelo método de Runge-Kutta de 4ª ordem, com passo de integração 0.001 e passo de amostragem 0.02.

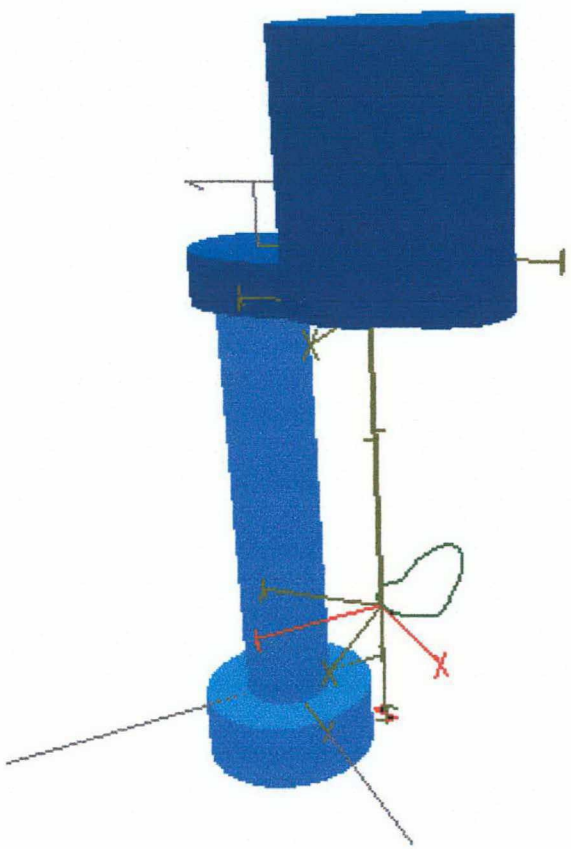


Figura 5.8 Trajetória definida no espaço operacional, correspondendo a uma circunferência no plano xy, com raio igual 0.10m, e com mudança na altura da ordem de 0.05m[54]

Os resultados, no espaço das juntas, são apresentados na Figura 5.9. A Figura 5.10 mostra o histórico da posição e orientação do efetuador final. Os gráficos das variáveis mostrados nas Figuras são traçados pelos componentes desenvolvidos para uso no simulador, demonstrando sua funcionalidade.

[54] O elo 3 não foi desenhado para possibilitar a visualização da trajetória.

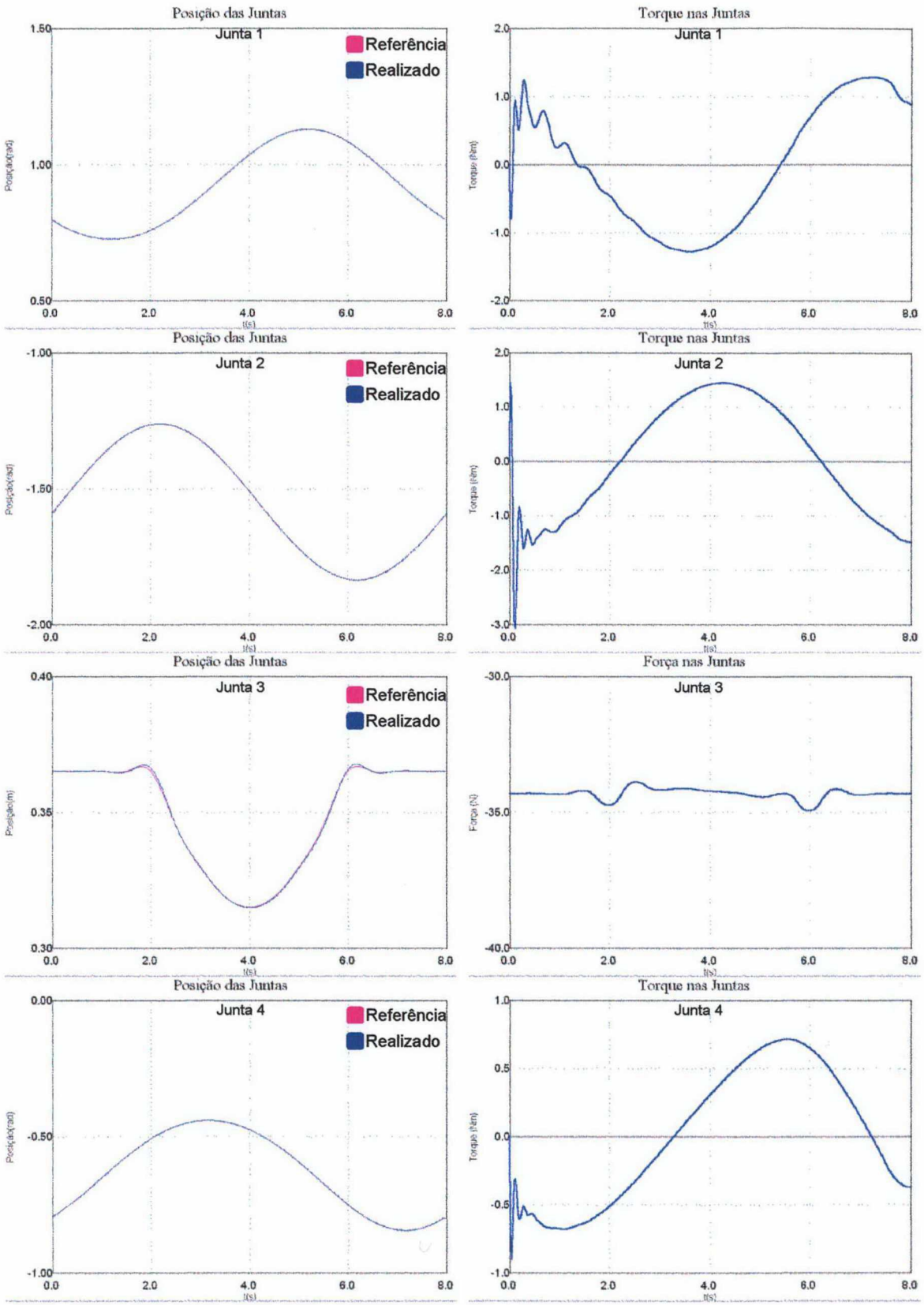


Figura 5.9 Histórico de posições e torques das juntas no seguimento de uma trajetória circular no espaço operacional, utilizando um controlador PD com compensação de gravidade

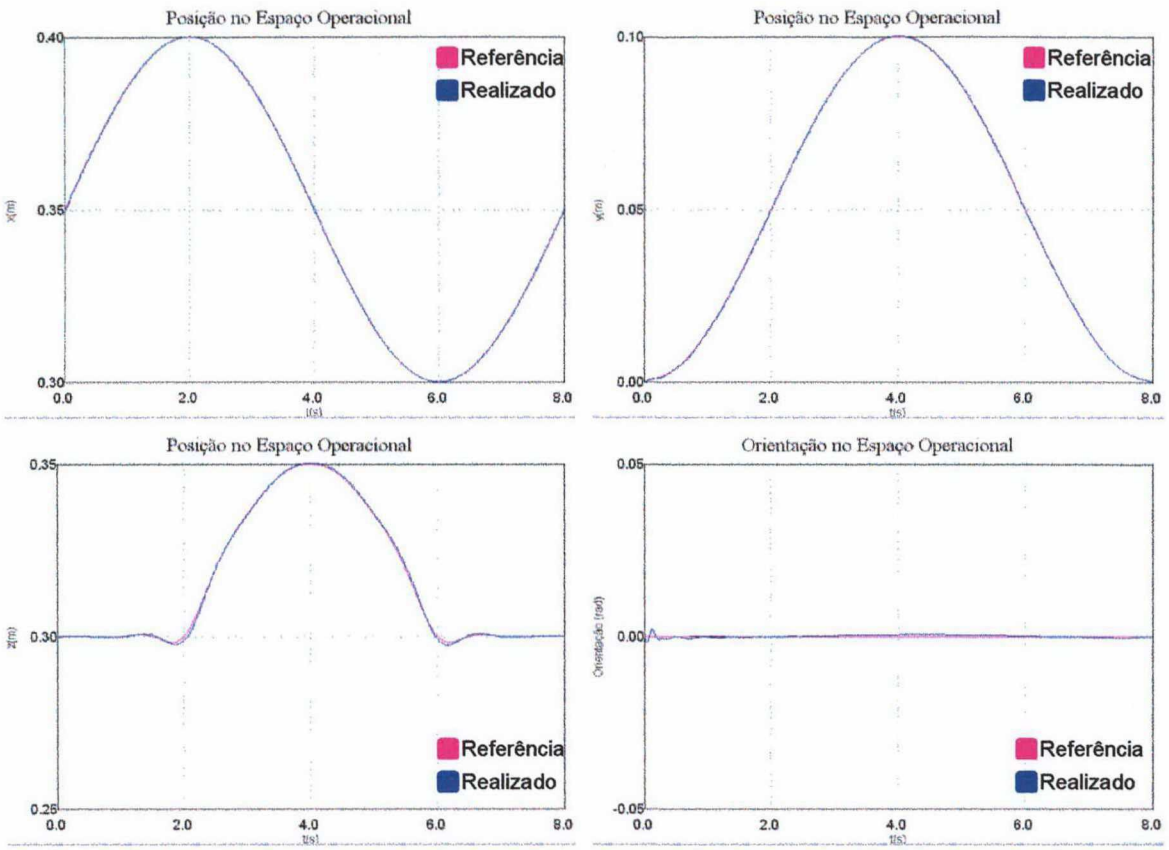


Figura 5.10 Histórico de posição e orientação do efetuador final no seguimento de uma trajetória circular, utilizando um controlador PD com compensação de gravidade

5.3 CONCLUSÃO

Este capítulo analisou os resultados obtidos com o simulador na modelagem de manipuladores e execução de tarefas em malha aberta e malha fechada. Inicialmente, foi feita a validação do modelo cinemático e do modelo dinâmico do simulador, através da comparação dos resultados de simulações feitas com um manipulador plano de dois graus de liberdade com os encontrados na literatura. Após, foi feita a modelagem de um manipulador tipo SCARA, utilizando parâmetros cinemáticos e dinâmicos de um manipulador real. Além de validar os modelos cinemático e dinâmico, as simulações comprovaram o funcionamento do mecanismo de acoplamento de controladores implementados externamente ao simulador, além dos elementos de visualização, como traçadores de gráficos e câmera sintética.

6. CONCLUSÕES GERAIS

Este trabalho apresentou o desenvolvimento de um *software* para a simulação de robôs manipuladores. Para esse fim, fez-se um estudo da modelagem cinemática e da modelagem dinâmica dos manipuladores. Foi analisada uma solução para a especificação de tarefas, e verificou-se a necessidade do uso de geradores de trajetória para o detalhamento do caminho a ser seguido pelo manipulador na execução de tarefas. Para possibilitar ao usuário uma maneira de testar seus próprios algoritmos de controle, foi criada uma forma de implementação destes externamente ao *software*, com posterior acesso deste aos controladores a serem testados.

Para análise dos resultados, foi criado um componente para traçado de gráficos das variáveis do sistema, como posições, velocidades, acelerações e esforços nas juntas e no efetuador final. A fim de dar noção de como as tarefas são executadas, realizam-se animações com modelos dos manipuladores em um ambiente virtual, visualizadas através de uma câmera sintética, implementada com base na biblioteca OpenGL.

Na modelagem cinemática do manipulador, foram observados problemas para a inversão cinemática, particularmente em configurações próximas a singularidades, tanto da cinemática quanto da representação mínima. Para a modelagem dinâmica, foi utilizada a formulação de Newton-Euler, que provou ser vantajosa do ponto de vista computacional, não só pela performance, mas também pela generalidade inerente ao método. Para a especificação dos manipuladores, foi criado um formato de arquivo texto que relaciona seus parâmetros cinemáticos e dinâmicos, além do modelo do corpo destes.

Em relação aos geradores de trajetória, foram implementados algoritmos de trajetórias ponto a ponto e trajetórias contínuas através de uma hierarquia de classes.

Para dar suporte à simulação, foram criadas classes para representação de matrizes e vetores, além de implementados métodos de soluções de sistemas de equações diferenciais ordinárias através de uma hierarquia de classes.

O paradigma da orientação a objetos provou ser adequado ao projeto em questão, possibilitando o desenvolvimento modular dos elementos da simulação, com uma integração quase direta destes ao *software*. A independência dos componentes agilizou os testes e modificações de código, além de garantir a reusabilidade dos componentes.

O uso do simulador provou ser eficaz para a representação de vários tipos de manipuladores, sendo utilizadas as especificações dos parâmetros cinemáticos e dinâmicos encontrados nos exemplos da literatura e do manual do manipulador SCARA do Laboratório de Robótica da UFSC[1,2,6,33]. Salienta-se a importância da identificação dos parâmetros dos manipuladores para a correspondência da simulação com a realidade.

Trabalhos futuros nesta direção incluem a implantação de novos recursos ao *software*, como facilidades para o usuário modelar as formas geométricas dos robôs e dos objetos do meio, seja através da interação entre o simulador e alguma ferramenta como o AutoCad ou 3D Studio, seja através da inclusão de um editor gráfico 3D. Para facilitar o estudo dos robôs manipuladores para iniciantes através do simulador, é desejável que este contenha uma base maior de modelos de manipuladores (cinemáticos, dinâmicos e geométricos) e controladores, preferivelmente baseados em robôs reais, além de um tutorial que explique passo a passo como especificar tarefas, como criar novos modelos e alterar os existentes, complementando com uma fundamentação teórica.

Considerando a revisão dos componentes, outro trabalho consiste na extensão do recurso empregado para a implementação de controladores por parte do usuário aos geradores de trajetória e aos manipuladores. Em relação a estes, seria interessante considerar os efeitos das flexibilidades nas juntas e nos elos, além de novos modelos para o atrito, pois estas características afetam sensivelmente o desempenho dos robôs[1,7,34].

Uma perspectiva de trabalho importante consiste na adição da interação do manipulador com o meio, permitindo simular tarefas que envolvam o controle de força, pois esta é uma área que vem se desenvolvendo em função das necessidades crescentes em automação de processos[2,7,21,35,36]. O uso de controle no espaço operacional deve ser

incluído como recurso do simulador. Além disso, devem ser criados novos componentes para a modelagem de elementos do meio e sensores de força, além da modelagem do contato. Uma série de trabalhos vêm sendo realizados neste sentido, cujos resultados podem ser aproveitados para o acréscimo desses recursos ao simulador[37,38,39,40].

Por fim, cabe salientar que os componentes criados para o simulador constituem uma contribuição deste trabalho, pois estes podem e já estão sendo reutilizados em outras aplicações.

ANEXO 1 - BIBLIOTECAS DE APOIO

As bibliotecas apresentadas a seguir contém classes, funções e constantes que foram desenvolvidas para dar suporte ao *software* simulador de robôs manipuladores. Esses componentes foram, na medida do possível, desenvolvidos de forma genérica, a fim de possibilitar seu uso em outros *software*.

A descrição das classes e seus métodos, das constantes, macros e funções abaixo tem como objetivo principal esclarecer como esses elementos podem ser empregados no desenvolvimento de *software*. Quando necessário, são listados programas exemplos para facilitar a compreensão da utilização de algum objeto mais complexo.

Deve-se observar que essas bibliotecas são a base do simulador de robôs manipuladores, especialmente por implementar matrizes, vetores e métodos numéricos de solução de equações diferenciais.

A1.1 CrExtMth

Nessa biblioteca estão definidas as classes relacionadas a entes matemáticos como matrizes, vetores e métodos numéricos para solução de equações diferenciais ordinárias. Seus métodos estão implementados na biblioteca `CrExtMth.lib`, que deve ser adicionada ao projeto do *software* em desenvolvimento. Não há um arquivo de cabeçalho único, pois a biblioteca foi montada a partir de diferentes arquivos fonte. Para cada caso, serão citados os arquivos de cabeçalho correspondentes para a adição no código do *software*.

Além dos elementos acima citados, estão implementadas funções auxiliares à manipulação de matrizes de transformação homogênea e representações mínimas de orientação por ângulos de Euler e RPY, bem como constantes e macros para tratar de ângulos e precisão numérica.

A1.1.1 Matrix

A classe `Matrix` modela matrizes reais bidimensionais e implementa os métodos para manipulá-las. Embora existam matrizes multidimensionais em C++, tanto estáticas quanto dinâmicas, sua manipulação matemática deixa a desejar. Além disso, a falta de

flexibilidade de definição do tamanho da matriz e suas operações motivaram a criação desta classe.

Uma matriz é uma coleção de elementos dispostos na forma de Tabela. Seus elementos individuais são acessados através de seu identificador e de dois subscritos, um para a linha e outro para a coluna em que o elemento está localizado na matriz, como mostrado na Figura A1.1.

$$M = \begin{bmatrix} m_{00} & m_{01} & \cdots & m_{0c} \\ m_{10} & m_{11} & \cdots & m_{1c} \\ \vdots & \vdots & \ddots & \vdots \\ m_{l0} & m_{l1} & \cdots & m_{lc} \end{bmatrix}$$

Figura A1.1 Representação de uma matriz bidimensional

Os métodos implementados em `Matrix` correspondem às operações aritméticas básicas, atribuição e acesso à matriz inteira ou em partes, verificação de matrizes particulares e operações específicas como normas e determinantes. Sua definição está no cabeçalho `CrMath.h`.

Matrix, ~Matrix Construtores e destrutor da classe

```
Matrix (void);
Matrix (unsigned r, unsigned c, double *V=NULL);
Matrix (unsigned r, unsigned c, unsigned t,...);
Matrix (Matrix &M);
```

Existem quatro construtores distintos para `Matrix`. O primeiro é o construtor vazio, que define um objeto `Matrix` com dimensão zero. O segundo construtor permite definir as dimensões da matriz, e opcionalmente inicializá-la a partir de um vetor de elementos `double`. O terceiro tem como parâmetros as dimensões desejadas e uma lista de elementos para inicialização, cujo tamanho é dado pelo parâmetro `t`. Por último, há o construtor de cópia, que define uma matriz copiando outra já existente. Estes construtores são válidos tanto para declaração de variáveis quanto para alocação dinâmica.

O destrutor de `Matrix` é automaticamente invocado quando uma instância dessa classe é destruída. Sua função é desalocar a memória utilizada no armazenamento dos elementos da matriz.

Clear() Limpa a matriz

```
void Clear (void);
```

O método `Clear()` define uma matriz com dimensão nula. Eqüivale a `Dim(0,0)`, mas mais eficiente, sendo indicado para *zerar* matrizes.

Dim() Define dimensões da matriz

```
void Dim (unsigned r, unsigned c, char Cp=mtNoCopy);
```

O método `Dim()` permite definir as dimensões da matriz. Ao redefinir as dimensões (`r` linhas e `c` colunas) de uma matriz já inicializada, o parâmetro `Cp` pode ser utilizado para forçar (`mtCopy`) ou não (`mtNoCopy`) a cópia dos elementos já existentes, de acordo com as novas dimensões. Por default, este parâmetro é `mtNoCopy`.

```
M.Dim(3,6);          // Define matriz com 3 linhas e 6 colunas
M.Dim(3,3,mtCopy); // Define matriz 3x3 copiando elementos
```

Fill() Preenche a matriz

```
void Fill (unsigned r, unsigned c, double V=0.0);
```

Este método define as dimensões da matriz, e preenche seus elementos com o valor `V` passado como parâmetro. Por default, `V=0.0`.

InsRow() Insere linhas

```
void InsRow (unsigned r=0, unsigned n=1, double V=0.0);
```

Insere `n` linhas na matriz a partir da linha `r`, deslocando as linhas que existem após esta. As novas linhas serão preenchidas com o valor `V`. Caso `r` seja maior que o número de linhas da matriz, as novas linhas serão adicionadas ao final desta. A matriz não pode estar originalmente vazia.

```
M.InsRow();          // Insere 1 linha no início da matriz
M.InsRow(2,3,1.0); // Insere 3 linhas a partir da linha 2
```

DelRow() Remove linhas

```
void DelRow (unsigned r=0, unsigned n=1);
```

Elimina `n` linhas da matriz, a partir da linha `r` (incluindo esta). As linhas existentes após a faixa a ser eliminada são relocadas. Caso o número de linhas a serem removidas exceda o número de linhas da matriz, esta será esvaziada como se fosse executado o método `Clear()`.

```
M.DelRow();     // Elimina a primeira linha da matriz
M.DelRow(2,1); // Elimina a linha 2 da matriz
```

InsCol() Insere colunas

```
void InsCol (unsigned c=0, unsigned n=1, double V=0.0);
```

Insere `n` colunas na matriz a partir da coluna `c`, deslocando as colunas existentes após esta. As novas colunas serão preenchidas com o valor `V`. Caso `c` seja maior que o número de colunas da matriz, as novas colunas serão adicionadas ao final desta. A matriz não pode estar originalmente vazia.

```
M.InsCol();          // Insere 1 coluna no início da matriz
M.InsCol(3,2,0.0); // Insere 2 colunas na coluna 3
```

DelCol() Remove colunas

```
void DelCol (unsigned c=0, unsigned n=1);
```

Elimina `n` colunas da matriz, a partir da coluna `c` (incluindo esta). As colunas existentes após a faixa a ser eliminada são relocadas. Caso o número de colunas a serem removidas exceda o número de colunas da matriz, esta será esvaziada como se fosse executado o método `Clear()`.

```
M.DelCol();     // Elimina a primeira coluna da matriz
M.DelCol(2,2); // Elimina as colunas 2 e 3 da matriz
```

Merge() Combina matrizes

```
void Merge (Matrix &M1, Matrix &M2, int kind=mtRow);
```

Este método define a matriz através da combinação das matrizes passadas como parâmetro. O parâmetro `kind` especifica se será feita junção das linhas (`mtRow`, default) ou junção das colunas (`mtCol`). O número de colunas, no primeiro caso, será igual ao da matriz com mais colunas. No segundo caso, o número de linhas será igual ao da matriz com mais linhas. Os elementos excedentes são preenchidos com zeros.

Operadores | , |= Combinam matrizes horizontalmente

```
Matrix operator|  (Matrix M);
void   operator|= (Matrix M);
```

Estes operadores fazem o mesmo que o método `Merge()` para combinação de matrizes através da junção de colunas. No primeiro caso, o operador retornará uma matriz formada pela combinação da matriz corrente com a matriz `M`. O operador |= não retorna, fazendo a combinação na própria matriz corrente. O número de linhas da matriz resultante é igual ao da matriz com mais linhas. Os elementos excedentes são preenchidos com zeros.

Operadores ^ , ^= Combinam matrizes verticalmente

```
Matrix operator^  (Matrix M);
void   operator^= (Matrix M);
```

Estes operadores fazem o mesmo que o método `Merge()` para combinação de matrizes através da junção de linhas. No primeiro caso, o operador retornará uma matriz formada pela combinação da matriz corrente com a matriz `M`. O operador ^= não retorna, fazendo a combinação na matriz corrente. O número de colunas da matriz resultante é igual ao da matriz com mais colunas. Os elementos excedentes serão preenchidos com zeros.

Eye() Matriz identidade

```
void Eye (unsigned r);
```

Define uma matriz identidade de dimensão `r x r`. Nesta, a diagonal principal é formada por 1.0s, enquanto os demais elementos são nulos.

Diag() Matriz diagonal

```
void Diag (Matrix M);
void Diag (int N, double *V);
```

Define uma matriz diagonal cujos elementos são passados como parâmetro. A dimensão da matriz quadrada é igual ao número de elementos de `M` no primeiro formato, ou igual a `N`, para o segundo formato.

Operador = Atribuição

```
Matrix& operator= (Matrix M);
```

Este operador implementa a atribuição de matrizes. Assim, a matriz corrente terá suas dimensões definidas pelas dimensões da matriz `M` e seus elementos serão copiados de `M`. Sua utilização é a mesma de um operador de atribuição convencional.

```
M1 = M2 // Copia a matriz M2 para a matriz M1
```

Operadores () , [] Acesso aos elementos da matriz

```
Matrix  operator() (unsigned i, unsigned j, unsigned r, unsigned c);
double& operator() (unsigned i, unsigned j);
double& operator[] (unsigned n);
```

Existem três possibilidades para o acesso aos elementos de uma matriz. Para o acesso a um elemento individual, pode-se utilizar a segunda forma do operador (), onde especifica-se a posição `(i,j)` do elemento. Além disso, o operador [] fornece uma maneira rápida de acesso aos elementos da matriz considerando-a um vetor. O parâmetro n especifica a posição no vetor formado pelas linhas consecutivas da matriz.

Obtém-se uma submatriz através da primeira forma do operador (). Nesta, especifica-se a posição `(i,j)` do primeiro elemento da submatriz e o número `r` de linhas e `c` de colunas a serem copiados. Se a especificação estiver fora das dimensões da matriz, será retornada uma matriz nula.

Deve-se observar que a submatriz resultante é uma nova matriz, cujas operações subsequentes não afetarão a matriz original, ao contrário dos operadores que dão acesso aos próprios elementos da matriz.

```
Matrix M(4,4);      // Define uma matriz 4x4
M(2,2) = 10;        // Atribui-se 10 ao elemento (2,2) de M
M[5] = 1.0;         // Atribui-se 1.0 ao elemento (1,1) de M
double X = M(1,0); // Copia o elemento (1,0) de M para X
double Y=M[7];      // Copia o elemento (1,3) de M para Y
Matriz Q=M(1,1,3,2); // A submatriz 3x2 de M é copiada para Q
```

Set() Preenche parcialmente uma matriz

```
void Set (Matrix M, unsigned i=0, unsigned j=0);
```

Como um complemento ao operador () para submatriz, o método `Set()` preenche parcialmente a matriz a partir de uma matriz `M` passada como parâmetro, a partir da posição inicial `(i,j)`.

GetLineAddress() Ponteiro para linhas da matriz

```
double* GetLineAddress (unsigned i=0);
```

Retorna um ponteiro `double` para o início da linha `i`. Os elementos desta linha e as posteriores serão vistos como um vetor `double`.

Size() Dimensões da matriz

```
int Size (char V=mtTotal);
```

`Size()` retorna o número de linhas da matriz (`V=mtRow`), o número de colunas (`V=mtCol`), ou o número total de elementos (`V=mtTotal`, default).

IsSquare() Verifica matriz quadrada

```
int IsSquare (void);
```

Retorna 1 se a matriz for quadrada ou 0, caso contrário.

IsRow() Verifica matriz linha

```
int IsRow (void);
```

Retorna 1 se a matriz for linha (número de linhas = 1).

IsColumn() Verifica matriz coluna

```
int IsColumn (void);
```

Retorna 1 se a matriz for coluna (número de colunas = 1).

IsDiagonal() Verifica matriz diagonal

```
int IsDiagonal (void);
```

Retorna 1 se a matriz for diagonal.

IsIdentity() Verifica matriz identidade

```
int IsIdentity (void);
```

Retorna 1 se a matriz for identidade.

IsSymmetric() Verifica matriz simétrica

```
int IsSymmetric (void);
```

Retorna 1 se a matriz for simétrica.

IsSkewSymmetric() Verifica matriz anti-simétrica

```
int IsSkewSymmetric (void);
```

Retorna 1 se a matriz for anti-simétrica.

IsOrthogonal() Verifica matriz ortogonal

```
int IsOrthogonal (void);
```

Retorna 1 se a matriz for ortogonal.

IsSingular() Verifica matriz singular

```
int IsSingular (void);
```

Retorna 1 se a matriz for singular (`Det()` = 0).

Det() Determinante

```
double Det (void);
```

Retorna o determinante da matriz. Se a matriz não for quadrada, `Det()` causa erro de divisão por zero.

Inv(), PInv() Inversa, pseudoinversa

```
Matrix Inv (void);
Matrix PInv (void);
```

`Inv()` retorna a inversa da matriz corrente (M^{-1}), se existir. Se a matriz não for quadrada ou não existir uma inversa (matriz singular), retorna uma matriz vazia. Se a matriz não for quadrada, pode-se utilizar a pseudo-inversa à direita, `PInv()`.

Trc() Traço

```
double Trc (void);
```

Retorna o traço da matriz, que consiste na soma dos elementos da sua diagonal principal. Para matriz não quadrada, retorna um valor desconhecido.

Norm() Norma

double Norm (void);

Retorna a norma 2 da matriz, definida como $\|M\| = \sqrt{\sum_{i=0}^{r}\sum_{j=0}^{c} m_{ij}^{\,2}}$

Trn(), Operador ~ Transposição

```
void   Trn (Matrix &M);
Matrix operator~ (void);
```

Determina a matriz transposta, onde as linhas são trocadas pelas colunas.

Operadores == , != Igualdade, desigualdade de matrizes

```
int operator== (Matrix M);
int operator!= (Matrix M);
```

Estes operadores permitem definir se duas matrizes são iguais ou diferentes. Para serem iguais, as matrizes precisam ter as mesmas dimensões e elementos. A igualdade dos elementos é testada para precisão `EPD` (v. Miscelânea de Funções). Estas funções retornam 1 se o teste for verdadeiro ou 0, caso contrário.

Ops() Operações aritméticas

```
void Ops (Matrix &M1, Matrix &M2, char kind='+');
```

O método `Ops()` realiza operações aritméticas entre duas matrizes, de forma que a matriz corrente armazene o resultado. Embora existam operadores que realizem as mesmas operações, `Ops()` pode ser mais eficiente por utilizar passagem de parâmetros por referência e não ser necessário copiar a matriz resultante. Isso, porém, limita o uso de `Ops()` a casos em que os operandos são variáveis, não valendo expressões. O parâmetro `kind` especifica a operação a ser efetuada, podendo ser '+', '-', '*' ou '/'.

```
A.Ops(B,C);        // A=B+C
A.Ops(B,C,'-');    // A=B-C
A.Ops(B,C,'*');    // A=B*C
```

Operadores +, += Soma

```
Matrix operator+  (void);
Matrix operator+  (Matrix M);
void   operator+= (Matrix &M);
Matrix operator+  (double E);
Matrix operator+  (double E, Matrix &M);
void   operator+= (double E);
```

O operador + tem várias funções, de acordo com a sua utilização. Na forma unária, não modifica a matriz. Na forma binária, permite somar um valor escalar a todos os elementos da matriz ou duas matrizes entre si, desde que tenham a mesma dimensão. O operador + retorna uma matriz de resultado, podendo esta ser vazia caso as dimensões das matrizes não sejam as mesmas. O += monta o resultado sobre a matriz corrente.

```
A=B+C;     // B+C resulta em uma matriz copiada para A
A=B+5.3;   // B+5.3 resulta em uma matriz copiada para A
A+=B;      // À matriz A é adicionada a matriz B
A+=5.3;    // Aos elementos de A soma-se o valor 5.3
```

Operadores - , -= Subtração, Negação

```
Matrix operator-  (void);
Matrix operator-  (Matrix M);
void   operator-= (Matrix &M);
Matrix operator-  (double E);
Matrix operator-  (double E, Matrix &M);
void   operator-= (double E);
```

O operador - tem várias funções, de acordo com a sua utilização. Na forma unária, troca o sinal dos elementos da matriz. Na forma binária, permite subtrair um valor escalar de todos os elementos da matriz ou uma matriz da matriz corrente, desde que as dimensões das matrizes sejam as mesmas. O operador - retorna uma matriz de resultado, podendo ser vazia se as dimensões das matrizes não forem as mesmas. O -= modifica a matriz corrente.

```
A=-B;      // Aij é igual a -Bij
A=B-C;     // B-C resulta em uma matriz copiada para A
A=B-5.3;   // B-5.3 resulta em uma matriz copiada para A
A=5.3-B;   // Aij = 5.3-Bij
A-=B;      // A matriz A é subtraída da matriz B
A-=5.3;    // Subtrai-se 5.3 dos elementos de A
```

Operadores * , *= Multiplicação

```
Matrix operator*  (Matrix M);
void   operator*= (Matrix &M);
Matrix operator*  (double E);
Matrix operator*  (double E, Matrix &M);
void   operator*= (double E);
```

Para o operador *, a multiplicação de matrizes é possível se o número de linhas da matriz a multiplicar for igual ao número de colunas da matriz corrente. Caso contrário, o resultado será uma matriz vazia. Também é possível multiplicar um valor escalar a todos os elementos da matriz. A diferença entre * e *= é que o primeiro retorna o resultado, enquanto o segundo modifica a matriz corrente (se a operação for válida).

```
A=B*C;     // B*C resulta em uma matriz copiada para A
A=B*1.5;   // B*1.5 resulta em uma matriz copiada para A
A=1.5*B;   // 5.3*B resulta em uma matriz copiada para A
A*=B;      // A = A*B
A*=1.5;    // Os elementos de A são multiplicados por 1.5
```

Operadores /, /= Divisão

```
Matrix  operator/  (Matrix M);
void    operator/= (Matrix &M);
Matrix  operator/  (double E);
void    operator/= (double E);
```

O operador / permite dividir os elementos da matriz por um valor escalar. Também é possível dividir a matriz corrente por uma matriz, desde que esta possua inversa. Assim, `A/B` é efetuada como AB^{-1}. Se não houver inversa, o resultado é uma matriz vazia. A diferença entre / e /= é que o primeiro retorna o resultado, enquanto o segundo modifica a matriz corrente (se a operação for válida).

```
A=B/C;     // B*C^-1 resulta em uma matriz copiada para A
A=B/1.5;   // B/1.5 resulta em uma matriz copiada para A
A/=B;      // A = A*B^-1
A/=1.5;    // Os elementos de A são divididos por 1.5
```

A1.1.2 VECTOR

A classe `Vector` modela a representação de vetores e, por extensão, de pontos no espaço cartesiano. Em se tratando do espaço 3D, o vetor é formado por três componentes, segundo cada eixo coordenado. Eventualmente, um vetor pode ser expresso em coordenadas esféricas (v. Figura A1.2), compostas pelo tamanho do vetor (magnitude) e seus ângulos de azimute e elevação (em torno dos eixos z e x, respectivamente). Pode-se representar um ponto no espaço cartesiano por um vetor, considerando que este expressa um deslocamento a partir da origem do sistema coordenado. A definição de `Vector` encontra-se no cabeçalho `CrMath.h`.

Os métodos implementados na classe correspondem às operações aritméticas básicas, outras específicas a vetores, testes e projeções, além de transformações homogêneas. Devido à sua natureza simples, os componentes de `Vector` são públicos, e podem ser acessados diretamente através do *array* `v` (v[0] = x; v[1] = y; v[2] = z).

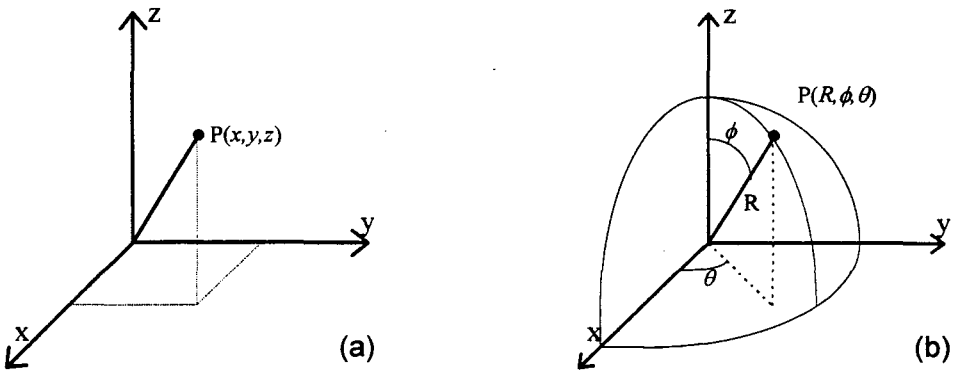


Figura A1.2 Representação de um ponto em coordenadas cartesianas(a) e coordenadas esféricas(b)

Vector Construtores da classe

```
Vector (void);
Vector (double a, double b, double c);
Vector (double *V);
Vector (Vector &V);
```

Existem quatro construtores distintos para `Vector`. O primeiro é o construtor vazio, que inicializa os componentes de `Vector` com 0.0. O segundo e o terceiro permitem definir valores para `Vector` através de três valores `double` ou de um vetor `double`. Por último, existe o construtor de cópia, que define os valores iniciais de um `Vector` com base em outro já existente. Este último é bastante utilizado de forma indireta, no retorno de funções ou passagem de parâmetros por valor. `Vector` não possui destrutor.

Operador () Acesso aos componentes do vetor

```
double& operator() (unsigned n);
```

Permite acesso aos componentes do vetor, de forma análoga ao acesso através do vetor v. O componente x corresponde a 0, o componente y a 1, e o componente z a 2.

```
V(1)    = 10.0;  // Define o componente Y de V como 10.0
X       = V(2);  // Copia o componente Z de V para X
W.v[0] = V(0);  // Copia o componente X de V para o X de W
```

IsUnitary() Verifica vetor unitário

```
int IsUnitary (void);
```

Retorna 1 se o vetor corrente for unitário ou 0 do contrário.

IsPerpendicular() Verifica perpendicularidade

```
int IsPerpendicular (Vector V);
```

Retorna 1 se o vetor corrente for perpendicular ao vetor `V`, ou 0 do contrário.

IsOrthogonal() Verifica ortogonalidade

```
int IsOrthogonal (Vector A, Vector B);
```

Retorna 1 se o vetor corrente for ortogonal a `A` e `B`, ou 0 do contrário.

Cartesian() Representação cartesiana

```
Vector Cartesian (void);
```

Considerando que o vetor corrente tem representação esférica, determina o equivalente em representação cartesiana, retornando o resultado em um vetor. Deve-se ter cuidado com este método e seu complementar (`Spherical()`), pois não há como distinguir um vetor cartesiano de um esférico. Além disso, os demais métodos funcionam somente em representação cartesiana (v. Figura A1.2).

Spherical() Representação esférica

```
Vector Spherical (void);
```

Considerando que o vetor corrente tem representação cartesiana, determina o equivalente em representação esférica, retornando o resultado em um vetor. Deve-se ter cuidado com este método e seu complementar (`Cartesian()`), pois não há como distinguir um vetor cartesiano de um esférico. Além disso, os demais métodos funcionam somente em representação cartesiana (v. Figura A1.2).

Project(), PerpProject() Projeções ortogonais

```
Vector Project     (Vector P);
Vector PerpProject (Vector P);
```

Estes métodos decompõem o vetor corrente em componentes segundo a direção do vetor `P` (`Project()`) e perpendicular a esta direção (`PerpProject()`). As projeções são mostradas na Figura A1.3.

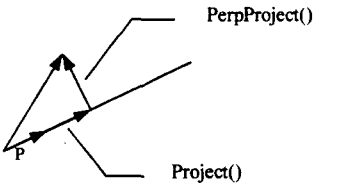


Figura A1.3 Projeções de um vetor em relação a outro vetor

Norm() Norma

```
double Norm (void);
```

Retorna a norma 2 do vetor, definida como $\|V\| = \sqrt{\sum_{i=0}^{2} v_i^2}$

Normalize() , Unitary() Normalização

```
void   Normalize (void);
Vector Unitary   (void);
```

Determina o vetor unitário do vetor corrente. A diferença entre os dois métodos é que o primeiro modifica o próprio vetor corrente, enquanto o segundo retorna o vetor unitário de resultado. Caso o vetor seja (0,0,0) ocorrerá erro de divisão por zero.

Operador = Atribuição

```
Vector& operator= (Vector V);
```

Este operador implementa a atribuição de vetores, copiando os componentes de V para o vetor corrente. Sua utilização é a mesma de um operador de atribuição convencional.

```
V1 = V2; // Copia os componentes de V2 para V1
```

Operadores +, += Soma

```
Vector operator+  (void);
Vector operator+  (double E);
Vector operator+  (Vector V);
void   operator+= (double E);
void   operator+= (Vector V);
```

O operador + tem várias funções, de acordo com a sua utilização. Na forma unária, não modifica o vetor. Na forma binária, permite somar um valor escalar a todos os elementos do vetor ou outro vetor. O operador + retorna um vetor de resultado. O += monta o resultado sobre o vetor corrente.

```
A=+B;     // Copia B para A
A=B+C;    // B+C resulta em um vetor copiado para A
A=B+5.3;  // B+5.3 resulta em um vetor copiado para A
A+=B;     // Ao vetor A adicionada-se o vetor B
A+=5.3;   // Aos elementos de A soma-se o valor 5.3
```

Operadores - , -= Subtração, Negação

```
Vector operator-  (void);
Vector operator-  (double E);
Vector operator-  (Vector V);
void   operator-= (double E);
void   operator-= (Vector V);
```

O operador - tem várias funções, de acordo com a sua utilização. Na forma unária, não modifica o vetor. Na forma binária, permite subtrair um valor escalar ou um vetor de todos os elementos do vetor corrente. O operador - retorna um vetor de resultado. O -= monta o resultado sobre o vetor corrente.

```
A=-B;     // Copia o inverso de B para A
A=B-C;    // B-C resulta em um vetor copiado para A
A=B-5.3;  // B-5.3 resulta em um vetor copiado para A
A-=B;     // Subtrai-se B do vetor A
A-=5.3;   // Subtrai-se 5.3 dos elementos de A
```

Operadores * , *= Multiplicação

```
Vector operator*  (double E);
Vector operator*  (Vector V);
Vector operator*  (double E, Vector V);
void   operator*= (double E);
```

O operador * multiplica um valor escalar a um vetor, servindo como escalonador, ou realiza produto vetorial entre dois vetores (v. `Cross()`). A diferença entre * e *= é que o primeiro retorna o resultado, enquanto o segundo modifica o vetor corrente.

```
A=B*C;     // B*C resulta em um vetor copiado para A
A=B*1.5;   // B*1.5 resulta em um vetor copiado para A
A=1.5*B;   // 1.5*B resulta em um vetor copiado para A
A*=B;      // A = A*B
A*=1.5;    // Os elementos de A são multiplicados por 1.5
```

Operadores /, /= Divisão

```
Vector operator/  (double E);
void   operator/= (double E);
```

O operador / divide os elementos do vetor por um valor escalar. A diferença entre / e /= é que o primeiro retorna o resultado, enquanto o segundo modifica o vetor corrente (se a operação for válida).

```
A=B/1.5;   // B/1.5 resulta em um vetor copiado para A
A/=1.5;    // Os elementos de A são divididos por 1.5
```

Operadores == , != Igualdade, desigualdade de vetores

```
int operator== (Vector V);
int operator!= (Vector V);
```

Estes operadores testam se dois vetores são iguais ou diferentes. Para igualdade, todos os componentes do vetor corrente devem ser iguais a `V`, com precisão `EPD` (v. Miscelânea de Funções). Caso contrário, os vetores serão considerados diferentes. Para respostas positivas, os métodos retornam 1; do contrário, retornam 0.

Dot() Produto escalar entre dois vetores

```
double Dot (Vector A, Vector B);
```

Implementa o produto escalar entre dois vetores. Retorna um valor `double`.

Cross() Produto vetorial entre dois vetores

```
Vector Cross (Vector A, Vector B);
```

Implementa o produto vetorial entre dois vetores, análogo ao operador *.

Angle() Ângulo entre dois vetores

```
double Angle (Vector A, Vector B);
```

Retorna o ângulo em radianos entre dois vetores.

Transform(), TransTrn() Transformação homogênea

```
Vector Transform (Matrix &M);
Vector TransfTrn (Matrix &M);
```

Realiza transformação no vetor, retornando o resultado como vetor. A transformação homogênea consiste em uma matriz 4x4, que contém a composição das transformações primárias possíveis. Para maiores detalhes, ver Transformações Homogêneas. `TransTrn()` é similar a `Transform()`, exceto que a matriz de transformação homogênea é fornecida transposta.

Translate() Translação de ponto

```
Vector Translate (Matrix &M);
Vector Translate (double a, double b, double c=0.0);
Vector Translate (Vector V);
```

Translada o ponto representado pelo vetor corrente, fornecido um outro vetor, os componentes nas direções cartesianas, ou uma matriz de transformação homogênea.

Rotate() Rotaciona um ponto

```
Vector Rotate (Matrix &M);
Vector Rotate (Vector R, Vector V);
Vector Rotate (double Rx,Ry,Rz, double Tx=0.0,Ty=0.0,Tz=0.0);
```

Rotaciona o ponto representado pelo vetor a partir de uma matriz de transformação homogênea (assumindo que sua submatriz superior esquerda 3x3 seja somente de rotação), ou da rotação consecutiva em torno dos eixos x, y, z em torno de um ponto, fornecidos por vetores ou componentes individuais.

Scale() Escalona um ponto

```
Vector Scale (double a, double b, double c=1.0);
Vector Scale (Vector S);
```

Escalona as componentes do vetor segundo os fatores passados individualmente ou por um vetor.

Shear() Distorce um ponto

```
Vector Shear (double hxy,double hxz,double hyx,double hyz,double hzx,double hzy);
```

Distorce o ponto de acordo com os fatores passados como parâmetro.

A1.1.3 TRANSFORMAÇÕES HOMOGÊNEAS

Transformações homogêneas são uma forma simples de realizar mapeamento entre dois sistemas de coordenadas cartesianos ou de alterar posições de acordo com uma determinada regra. Consistem em matrizes [4x4] que, multiplicando a representação vetorial de um ponto, transforma-o. Para tanto, seja um ponto **P** de dimensão aumentada, cuja representação é $\mathbf{P}(P_x,P_y,P_z,1)'$. Usando uma transformação **M**, obtém-se o ponto $\mathbf{Q}(Q_x,Q_y,Q_z,1)'$ da seguinte forma:

$$\mathbf{Q}=\mathbf{MP} \tag{A1.1}$$

$$(Q_x,Q_y,Q_z,1)'=\begin{bmatrix} & & t_x \\ \mathbf{H} & & t_y \\ & & t_z \\ 0 & & 1 \end{bmatrix}(P_x,P_y,P_z,1)' \tag{A1.2}$$

onde **H**[3x3] é uma transformação de similaridade, e $[t_x,t_y,t_z]$ corresponde a uma translação.

Uma transformação homogênea pode ser dividida em um conjunto de transformações elementares, que compreendem *translações*, *escalonamentos*, *distorções* e *rotações* em torno dos eixos coordenados. A composição de transformações homogêneas é feita através da multiplicação de suas matrizes.

Transformações homogêneas ainda têm as seguintes propriedades:

- Mantém linhas retas;
- Mantém o paralelismo entre linhas;
- Mantém a proporcionalidade;
- Volumes são escalonados por det **H**.

As funções abaixo geram matrizes de transformação homogênea, de acordo com as operações elementares acima citadas, cujas formas são mostradas a seguir. Para compô-las, basta multiplicá-las. Para transformar um vetor, utilizar o método `Transform()`.

InvHomog() Inversa de transformação homogênea

```
Matrix InvHomog (Matrix &M);
```

Retorna uma matriz de transformação homogênea definida como a inversa da matriz de transformação homogênea passada como parâmetro. É mais eficiente que o método `Inv()` para matrizes, porque aproveita as propriedades específicas de matrizes homogêneas.

SetTranslateMatrix() Cria matriz de translação

```
Matrix SetTranslateMatrix (double X, double Y, double Z=0.0);
Matrix SetTranslateMatrix (Vector T);
```

Define uma matriz de transformação homogênea representando uma translação, passada como um vetor ou pelos componentes individuais. A matriz resultante tem a forma

$$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

SetScaleMatrix() Cria matriz de escala

```
Matrix SetScaleMatrix (double X, double Y, double Z=1.0);
Matrix SetScaleMatrix (Vector S);
```

Define uma matriz de transformação homogênea representando escalonamento, passado como um vetor ou pelos componentes individuais. A matriz resultante tem a forma

$$\begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

SetShearMatrix() Cria matriz de distorção

```
Matrix SetShearMatrix (double hxy,double hxz,double hyx,double hyz,double hzx,double hzy)
```

Define uma matriz de transformação homogênea representando uma distorção especificada pelos parâmetros. A matriz resultante tem a forma

$$\begin{bmatrix} 1 & h_{xy} & h_{xz} & 0 \\ h_{yx} & 1 & h_{yz} & 0 \\ h_{zx} & h_{zy} & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Euler2Matrix(), Matrix2Euler() Ângulos de Euler para matriz

```
Matrix Euler2Matrix (double X, double Y, double Z);
Matrix Euler2Matrix (Vector R);
void   Matrix2Euler (Matrix M, double &X,double &Y,double &Z);
Vector Matrix2Euler (Matrix M);
```

Retorna uma matriz de transformação homogênea definida a partir dos ângulos de Euler ZYZ. A função inversa, `Matrix2Euler()`, retorna um conjunto possível de ângulos de Euler ZYZ que geraria uma matriz de transformação homogênea correspondente a rotação.

RPY2Matrix() Ângulos RPY para matriz

```
Matrix RPY2Matrix (double X, double Y, double Z);
Matrix RPY2Matrix (Vector R);
```

Retorna uma matriz de transformação homogênea definida a partir dos ângulos Roll-Pitch-Yaw.

SetRotateMatrix() Cria matriz de rotação

```
Matrix SetRotateMatrix (double Rx,Ry,Rz, double Tx=0.0,Ty=0.0,Tz=0.0);
Matrix SetRotateMatrix (Vector R, Vector T);
```

Define uma matriz de transformação homogênea representando rotações consecutivas em torno dos eixos x,y,z, respectivamente, a partir de um ponto passado como parâmetro. Esta matriz é a composição das rotações elementares descritas abaixo e de translações, na forma

$$M=T(-V)*R(x)*R(y)*R(z)*T(V) \qquad \text{(A1.3)}$$

$$R(x)=\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & c & s & 0\\ 0 & -s & c & 0\\ 0 & 0 & 0 & 1\end{bmatrix},\quad R(y)=\begin{bmatrix}c & 0 & -s & 0\\ 0 & 1 & 0 & 0\\ s & 0 & c & 0\\ 0 & 0 & 0 & 1\end{bmatrix},\quad R(z)=\begin{bmatrix}c & s & 0 & 0\\ -s & c & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\end{bmatrix},$$

onde s é o seno e c é o cosseno do ângulo de rotação (em radianos).

A1.1.4 MISCELÂNEA

Consistem em funções, constantes e #defines genéricos sem classificação específica.

EPS, EPD Constantes de precisão

As constantes `EPS` e `EPD` especificam duas precisões pré-definidas utilizadas em outras funções. Estas são iguais a

```
#define EPS     1e-6
#define EPD     1e-12
```

Pi e derivados Constantes trigonométricas

As constantes abaixo são necessárias em operações que envolvem ângulos.

```
#define Pi      3.14159265358979323846
#define Pid2    1.57079632679489661923      // Pi/2
#define Pid4    0.78539816339744830961 6    // Pi/4
#define Pix2    6.28318530717958648         // Pi*2
#define Pid180  0.0174532925199432958       // Pi/180
```

g2r(), r2g() Conversores de ângulos

Estes `#defines` convertem valores em graus para radianos (`g2r`) e vice-versa (`r2g`).

sign() Sinal

Este #define retorna +1 se o valor do parâmetro for positivo, -1 se este for negativo ou 0, de este for 0.

sqr(), cbc() Quadrado, cubo

`sqr` retorna um valor elevado ao quadrado, enquanto `cbc` retorna um valor elevado ao cubo. Ambos são #defines.

NearZero() Aproxima para zero

```
double NearZero (double X, double P=EPS);
```

Esta função verifica se o valor absoluto do parâmetro `X` é menor que a precisão especificada `P` e, se for, retorna 0.0.

RoundOff() Arredondamento

```
double RoundOff (double X, double P=EPS);
```

Esta função arredonda `X` para uma determinada precisão.

Intg(), Frac() Inteiro e fracionário

```
double Intg (double X);
double Frac (double X);
```

Estas funções retornam a parte inteira (`Intg()`) e a parte menor que 1.0 (`Frac()`) de um valor numérico `double`.

gt() , ge() , lt() , le() , eq() , ne() Comparação de reais

```
int gt (double X, double Y, double P=EPS); // maior que
int ge (double X, double Y, double P=EPS); // maior ou igual a
int lt (double X, double Y, double P=EPS); // menor que
int le (double X, double Y, double P=EPS); // menor ou igual a
int eq (double X, double Y, double P=EPS); // igual a
int ne (double X, double Y, double P=EPS); // diferente de
```

Estas funções são melhorias das operações de comparação normalmente empregadas para valores `double`. Isso se deve porque a comparação é feita com uma determinada precisão, definida pelo parâmetro `P`. Estas funções retornam 1, se a comparação for verdadeira, ou 0 do contrário.

AdjAng() , AdjAngDeg() Arredondamento

```
double AdjAng    (double X);
double AdjAngDeg (double X);
```

Ajustam `X` para a faixa $-\pi$ a π (`AdjAng()`) ou −180.0 a 180.0 (`AdjAngDeg()`).

A1.1.5 TODE SOLVER

Equações diferenciais são formadas por uma função desconhecida e suas derivadas [19]. Esta classe de equações desempenha um papel fundamental em engenharia, pois vários fenômenos físicos são modelados em função de suas taxas de variação.

Equações diferenciais ordinárias são equações diferenciais em que existe apenas uma variável independente, sobre a qual as variáveis dependentes são derivadas. Elas são classificadas de acordo com a mais alta derivada presente, designando a *ordem* da equação.

Em nível computacional, existem diversas técnicas de solução numérica para essa classe de equações, classificadas entre métodos de passo fixo e de passo variável.

A classe `TODESolver` implementa um método de passo fixo conhecido como método de Euler. Este método se baseia no fato de que uma equação diferencial ordinária de primeiro grau tem a forma genérica

$$\frac{dy}{dx} = f(x,y) \tag{A1.4}$$

e sua solução pode ser discretizada da forma

$$y_{i+1} = y_i + \phi h \tag{A1.5}$$

onde ϕ é interpretada como uma declividade e h é o passo de discretização. Para o caso do método de Euler, $\phi = f(x,y)$. Obviamente, há erro em relação à solução analítica, dependendo da natureza da equação e do passo de discretização.

Para resolver equações de ordem mais alta, deve-se realizar uma transformação para um sistema de equações de primeira ordem. A definição de `TODESolver` está no cabeçalho `CrOde.h`.

O sistema a ser resolvido deve estar implementado em uma classe derivada da classe base abstrata `TODE`, apresentada no item A1.1.7.

TODESolver, ~TODESolver Construtores e destrutor da classe

```
TODE (void);
TODE (unsigned pEqNo, double pStep, double pSampling=0.0);
```

Existem dois construtores distintos para `TODESolver`. O primeiro é o construtor vazio, que não configura a instância da classe para operação, sendo necessário o uso do método `Define()` para sua configuração. O segundo construtor recebe como parâmetros o número de equações do sistema a resolver, o passo de discretização e o passo de amostragem. Este é opcional, sendo indicado nos casos em que o passo é pequeno e não se deseja armazenar todos os valores da solução. Se não definido, o passo de amostragem assume o mesmo valor do passo de integração.

O destrutor de `TODESolver` é automaticamente invocado quando uma instância dessa classe é destruída, com a função de desalocar a memória utilizada para o uso do mecanismo de solução. Os resultados obtidos pelo uso da instância sendo destruída não são perdidos, pois são armazenados em matrizes fornecidas pelo usuário.

Define() Define parâmetros do solver

```
int  Define (double V, char Kind=odeEqNo);
```

`Define()` é utilizada para a configuração do *solver*, recebendo como parâmetros o valor e o tipo de elemento a configurar. Este pode ser `odeEqNo` (default, representa o número de equações do sistema a resolver), `odeStep` (passo de discretização) e `odeSampling` (passo de amostragem).

Get() Retorna parâmetros do *solver*

```
double Get (char Kind=odeEqNo);
```

Complementar a `Define()`, o método `Get()` retorna valores da configuração do *solver*. O valor será retornado de acordo com o parâmetro passado, cujos valores possíveis são os mesmos utilizados em `Define()`.

Solve() Resolve o sistema de equações diferenciais ordinárias

```
int Solve (TODE *ODE, Matrix &X, double I,double F,
        int xi=0,int xf=-1, int Ind=odeNoTime, int dxi=-1,int dxf=-1);
Matrix Solve (TODE *ODE, double F);
```

Este método é o cerne da utilização de uma instância de `TODESolver`. `Solve()` retorna o número de valores gerados pela solução do sistema de equações diferenciais ordinárias. Como parâmetros, são necessários o endereço do objeto que representa o sistema a ser resolvido, uma matriz onde os resultados serão armazenados e o intervalo

para o qual se deseja a solução (`I` e `F`). A matriz de resultados será de dimensão tal que o número de linhas seja igual ao número de amostras (definidas em função do intervalo e do passo de amostragem) e o número de colunas sendo igual ao número de variáveis a armazenar. Por default, este valor é igual ao número de equações do sistema. Se o parâmetro `Ind` for igual a `odeTime`, a última coluna da matriz conterá os valores da variável independente. As variáveis de interesse a armazenar podem ser definidas pelo intervalo (`xi-xf`). Pode-se, ainda, armazenar os valores das derivadas ao longo da solução, especificando nos parâmetros (`dxi-dxf`) quais serão de interesse.

A segunda forma de `Solve()` é utilizada nos casos em que não se deseja resolver o sistema para um intervalo de tempo, mas sim obter um resultado final, onde uma condição determina a parada do método. Esta é implementada na classe descendente de `TODE` que modela o sistema de equações a resolver. A variável independente, neste caso, é inicializada com 0, sendo necessário definir o limite final desta, para que o método não fique executando indefinidamente. O resultado é retornado em uma matriz coluna.

Para uma maior compreensão da utilização destes métodos, ver o exemplo de utilização de objetos dessa classe no item A1.1.9.

A1.1.6 TRKUTTA

A classe `TRKutta` é derivada de `TODESolver`, tendo utilização similar. A principal diferença está no método empregado, que é o Runge-Kutta de passo fixo, cuja ordem pode ser especificada como 2ª, 3ª, 4ª, 5ª ou 8ª. Deve-se escolher a ordem de acordo com a complexidade e a ordem da equação diferencial ordinária a ser resolvida, a necessidade de precisão e a carga computacional máxima permitida.

Em relação a Euler, o método Runge-Kutta é mais preciso, graças ao emprego de expansão das equações diferenciais ordinárias em séries de Taylor, sem a necessidade de cálculos excessivos[19]. Embora existam diversas variações, a solução geral tem a forma

$$y_{i+1} = y_i + \phi(x,y,h)h \tag{A1.6}$$

onde $\phi(x,y,h)$ é uma função incremental que pode ser interpretada como a *declividade* no intervalo considerado. Esta função tem a forma

$$\phi = a_0k_0 + a_1k_1 + \ldots + a_{n-1}k_{n-1} \tag{A1.7}$$

$$k_0 = f(x,y) \tag{A1.7a}$$

$$k_1 = f(x+p_0h,\ y_0+q_{00}k_0h) \tag{A1.7b}$$

$$k_2 = f(x+p_1h,\ y_1+q_{10}k_0h+q_{11}k_1h) \tag{A1.7c}$$

...

$$k_{n-1} = f(x+p_{n-2}h,\ y_{n-2}+q_{n-2,0}k_0h+q_{n-2,1}k_1h+...+q_{n-2,n-2}k_{n-2}h) \qquad \text{(A1.7d)}$$

onde a_i, p_i e q_{ij} são constantes determinadas igualando a solução com a expansão em série de Taylor e comparando seus coeficientes, de acordo com a ordem do método Runge-Kutta utilizada.

Em termos de utilização, a única diferença em relação a `TODESolver` está nos construtores e no método `Define()`, que incluem a ordem do método Runge-Kutta. Recomenda-se a leitura da referência de `TODESolver`.

TRKutta, ~TRKutta Construtores e destrutor da classe

```
TRKutta (void);
TRKutta (unsigned pEqNo, double pStep, double pSampling=0.0, unsigned pOrder=RKO4);
```

Em relação a `TODESolver`, apenas o construtor com parâmetros muda, tendo um parâmetro a mais, que é a ordem do método. Se não especificado, o Runge-Kutta será de 4ª ordem.

Define() Define parâmetros do solver

```
int Define (double V, char Kind=odeEqNo);
```

Em relação a `TODESolver`, o método `Define()` tem como diferença apenas a possibilidade de se definir a ordem do *solver* empregado, se `Kind` for igual a `odeOrder`. As ordens possíveis são `RKO2` (segunda ordem, equivalente ao método de Euler), `RKO3` (terceira ordem), `RKO4` (quarta ordem), `RKO5` (quinta ordem), e `RKO8` (oitava ordem). A ordem do método deve ser escolhida de acordo com os requisitos de precisão da solução e carga computacional desejável.

Get() Retorna parâmetros do *solver*

```
double Get (char Kind=odeEqNo);
```

A única diferença deste método em relação a sua implementação na classe base é a possibilidade de retornar a ordem do método de Runge-Kutta empregado em `Solve()`.

Solve() Resolve a função diferencial

```
int Solve (TODE *ODE, Matrix &X, double I,double F,
        int xi=0,int xf=-1, int Ind=odeNoTime, int dxi=-1,int dxf=-1);
Matrix Solve (TODE *ODE, double F);
```

As duas variações deste método são empregadas da mesma maneira que no uso de instâncias da classe base, diferindo apenas em sua implementação.

A1.1.7 TODE

A classe `TODE` é uma classe base abstrata, que deve ser derivada para implementar um sistema de equações diferenciais ordinárias. Esta classe define os métodos e comportamento básico que devem ser seguidos para que se possa utilizar as classes `TODESolver` e `TRKutta` para a solução de equações diferenciais ordinárias. Como toda classe abstrata, não se pode instanciar objetos `TODE`, apenas utilizá-la como base de outras classes.

TODE, ~TODE — Construtores e destrutor da classe

```
TODE (void);
```

O construtor de `TODE` é vazio. As classes derivadas devem, nos seus contrutores, definir as dimensões das matrizes `X0` e `dX` (privadas), além de definir as condições iniciais do sistema na matriz `X0`. As duas matrizes devem ter dimensão `[EqNox1]`, onde `EqNo` é o número de equações diferenciais do sistema. Além destas matrizes, devem ser inicializadas também quaisquer atributos adicionais definidos na classe derivada.

GetEqNo() — Retorna o número de equações diferenciais

```
int GetEqNo (void);
```

Esta função tem como objetivo único retornar o número de equações do sistema.

Set() — Define as condições iniciais do sistema

```
void Set (Matrix &V);
```

Esta função deve inicializar a matriz `X0`, que contém as condições iniciais do sistema.

Init() — Retorna as condições iniciais do sistema

```
Matrix Init (void);
```

Esta função é chamada pela instância de `TODESolver` ou `TRKutta` quando do início da execução do método `Solve()`, para que as condições iniciais sejam definidas.

Evaluate() — Calcula as derivadas do sistema

```
Matrix Evaluate (double t, Matrix &X);
```

Esta função é a que realiza o cálculo das equações do sistema, retornando uma matriz coluna com as derivadas obtidas. Este é o método onde o sistema de equações é modelado.

SampleAction() Ação a executar em um passo de amostragem

```
void SampleAction (double t, int i);
```

O método `Solve()` da instância `TODESolver` ou `TRKutta` chama esta função cada vez que um conjunto de valores é colocado na matriz de resultados, para que alguma ação extra opcional seja executada, como a atualização de um contador de tempo, por exemplo. Os parâmetros `t` e `i` são, respectivamente, o valor da variável independente e o número da amostra gerada.

Continue() Condição de continuidade da resolução do sistema

```
int Continue (double t, Matrix &X);
```

`Continue()` é utilizada na segunda forma de `Solve()`, para verificar se o método deve continuar a resolução do sistema de equações. A função retorna 1, se o sistema deve continuar a ser resolvido, ou 0, do contrário.

A1.1.8 Roteiro Para a Solução de Sistemas de Equações Diferenciais Ordinárias

Para a solução de um sistema de equações diferenciais ordinárias modelados em uma classe derivada de `TODE` por uma instância da classe `TODESolver` ou `TRKutta`, alguns passos simples devem ser seguidos:

1. Modelar o sistema a resolver: Deve-se criar a classe derivada de `TODE`, implementando os seus métodos, particularmente `Evaluate()` e `Init()`.
2. Criar uma instância de `TODESolver`: Se declarada sem parâmetros, torna-se necessária a utilização do método `Define()` para configuração do *solver*.
3. Definir a ordem da equação, passo e amostragem: É feito através do método `Define()`. Devem ser definidos a ordem da equação (número de equações do sistema), o passo de discretização e o passo de amostragem. No caso de `TRKutta`, deve-se especificar também a ordem do método.
4. Resolver o sistema de equações: Através do método `Solve()`, que requer uma matriz para armazenamento da solução. Neste método, devem ser especificados o intervalo da solução e as variáveis que devem ser armazenadas.

A1.1.9 EXEMPLOS DE UTILIZAÇÃO DE TODE, TODESOLVER E TRKUTTA

O código abaixo ilustra a solução de uma equação diferencial ordinária. A classe `TODETeste` representa a equação diferencial a ser resolvida,

$$\dot{x} = -2t^3 + 12t^2 - 20t + 8.5 \tag{A1.8}$$

Faz-se uma comparação entre a solução por TODE e a solução analítica,

$$x = -0.5t^4 + 4t^3 - 10t^2 + 8.5t + 1 \tag{A1.9}$$

para a condição inicial x_0=1, sendo ambas armazenadas em um arquivo texto.

```
/*********************************/
/* Exemplo de utilização de TODE */
/* (c) Carlos R Rocha, 2000      */
/*********************************/
#define inicio 0.0
#define fim    4.0
#define passo  0.1

class TODETeste : public TODE
{public:
       TODETeste (void) {X0.Dim(1,1); X0[0]=1.0; dX.Dim(1,1);};
      ~TODETeste (void) {};
int    GetEqNo(void)    {return 1;};
void   Set (Matrix &V)  {if (V.Size(mtTotal)==1) X0 = V;};
Matrix Evaluate (double t, Matrix &X)
{dX[0] = -2.0*pow(t,3.0)+12.0*pow(t,2.0)-20.0*t+8.5;
 return dX;}
void  SampleAction (double t, int i){};
int   Continue     (double t, Matrix &X) {return (t<=7.5);};
};

double Fx (double t) //Solucao analitica
{return -0.5*pow(t,4.0) + 4*pow(t,3) - 10*t*t + 8.5*t + 1.0;}


int main()
{FILE     *A;
 register i;
 double   t;
 Matrix   X;
ODE = ::new TODETeste;

ODESolver = ::new TRKutta (1,passo,passo,RKO4);
ODESolver->Solve  (ODE,X,inicio,fim,0,0,odeNoTime,0,0);

if ((A = fopen("Resultados.Dat","w")) != NULL)
   {for (i=0, t=inicio; i<X.Size(mtTotal); i+=2, t+=passo)
        fprintf(A,"%6.3lf  %11.5lf  %11.5lf  %11.5lf  %13.7lf\n",
                  t, X[i], Fx(t), fabs(Fx(t)-X[i]), X[i+1]);}
fclose(A);

delete ODE; delete ODESOlver;
return 0;}
```

O código a seguir utiliza a segunda variação do método `Solve()` para a resolução da cinemática inversa em um manipulador plano de dois graus de liberdade, definindo uma condição de continuidade que será falsa quando o erro de posição estiver abaixo de um determinado limite.

```
/*********************************/
/* Exemplo de utilização de TODE */
/* (c) Carlos R Rocha, 2000      */
/*********************************/
#include <CrMath.h>
#include <CrODE.h>

class TCInv : public TODE
{private:
   Matrix K, R, J, C;
   Matrix Pd;

   Matrix JInv (Matrix &X)
      {J.Dim (2,2);
       J[0] = -sin(X[0])-sin(X[0]+X[1]); J[1] = -sin(X[0]+X[1]);
       J[2] = cos(X[0])+cos(X[0]+X[1]);  J[3] = cos(X[0]+X[1]);
       return J.Inv();}

   Matrix Cin (Matrix &X)
      {C.Dim(2,1);
       C[0] = cos(X[0])+cos(X[0]+X[1]); C[1] = sin(X[0])+sin(X[0]+X[1]);
       return C;}

 public:
       TCInv (void) {X0.Dim(2,1); X0[0]=1.5; X0[1] = 1.5; dX.Dim(2,1);
                     K.Dim(2,2); K[0] = K[3] = 100.0;
                     Pd.Dim(2,1); Pd[0] = -0.44969; Pd[1] = 0.98259;};
      ~TCInv (void) {};
int    GetEqNo(void)    {return 2;};
void   Set (Matrix &V)  {if (V.Size(mtTotal)==2) X0 = V;};

Matrix Evaluate (double t, Matrix &X)
{R = Pd-Cin(X);
 dX = JInv(X)*K*R;
 return dX;}

void  SampleAction (double t, int i){};
int   Continue     (double t, Matrix &X)
     {R = Pd-Cin(X); return (ne(R[0],0.0) || ne(R[1],0.0));};
};


int main()
{TODE       *ODE    = ::new TCInv;
 TODESolver *Solver = ::new TODESolver (2, 0.001, 0.1);
 Matrix X;

X = Solver->Solve (ODE, 5.0); // Limite máximo para achar a solução : 5.0

::delete ODE;
::delete Solver;
return 0;}
```

A1.2 CRMISC

Esta biblioteca contém funções diversas, mais voltadas para a manipulação de strings. No C++ Builder, as strings são preferencialmente tratadas como objetos da classe AnsiString, o que reduz a portabilidade das funções dessa biblioteca. Para utilizar os componentes dessa biblioteca, deve-se adicionar `CrMisc.lib` ao projeto do *software*. As definições estão no cabeçalho `CrMisc.h`.

min(), max(), lim() Funções de limites

Estes `#define` determinam o menor entre dois valores (`min`), o maior entre dois valores (`max`), ou estabelecem limites para o valor (`lim`). Neste caso, se o valor testado for menor que o limite inferior ou maior que o limite superior, o retorno é o limite violado.

GetBit(), SetBit(), ResetBit() Manipulação de bits

```
unsigned GetBit   (unsigned n, unsigned b);
unsigned SetBit   (unsigned n, unsigned b);
void     SetBit   (unsigned &n, unsigned b);
unsigned ResetBit (unsigned n, unsigned b);
void     ResetBit (unsigned &n, unsigned b);
```

Estas funções manipulam bits de um valor unsigned. `GetBit()` retorna o valor do bit `b` de `n`. Para definir o bit como 1, utiliza-se `SetBit()`. Para definir como 0, utiliza-se `ResetBit()`. `SetBit()` e `ResetBit()` tem dois formatos possíveis. O primeiro retorna o resultado, enquanto o segundo modifica o parâmetro n passado.

AnsiString2Angle(), Angle2AnsiString() Converte ângulo para string

```
AnsiString Angle2AnsiString (double V);
double     AnsiString2Angle (AnsiString V);
```

Estas funções lidam com a representação em `AnsiString` de ângulos em radianos. `Angle2AnsiString()` verifica se o parâmetro é um dos valores especiais de ângulos e, se positivo, retorna um nome; senão, apenas converte o valor para `AnsiString`. O contrário é feito por `AnsiString2Angle()` : se o parâmetro for um nome conhecido, retorna o ângulo correspondente; senão, apenas realiza a conversão para `double`. As constantes são: `pi` (π), `pix2` (2π), `pid2` ($\pi/2$), `pid3` ($\pi/3$), `pid4`($\pi/4$), `pid6` ($\pi/6$), positivas ou negativas.

CCount() Contagem de caracteres

```
int CCount (AnsiString S, char c);
```

Retorna o número de ocorrências do caractere `c` na string `S`.

CPos() Retira comentários

```
int CPos (AnsiString S, char c, int p=1);
```

Retorna a posição da primeira ocorrência do caractere `c` na string `S`, a partir de `p`.

IsNumeric() Verifica formato numérico

```
int IsNumeric (AnsiString S);
```

`IsNumeric()` retorna 1 se a string estiver em um formato numérico real, considerando sinal, ponto decimal e algarismos. Se não for numérico, retorna 0.

AnsiString2Double() Converte string para real

```
double AnsiString2Double (AnsiString S, double D=0.0);
```

Esta função difere da normalmente utilizada por remover caracteres que não fazem parte de um formato numérico para depois realizar a conversão para `double`. Se não puder ser identificado um formato numérico, retorna `D`.

CutComment() Retira comentários

```
AnsiString CutComment (AnsiString Orig);
```

Considerando comentários de linha como o caractere #, esta função retira esta parte da linha (incluindo o caractere #).

Cut() Destaca campos

```
AnsiString Cut (AnsiString S, char fd, unsigned n);
int Cut (AnsiString &S, char fd, unsigned n, AnsiString &D);
```

Considerando a string como sendo formada por campos separados por caracteres específicos, denominados separadores de campos, esta função retira o campo `n` desta string. O caractere separador é definido por `fd`. A primeira forma retorna o resultado, enquanto a segunda retorna o resultado no parâmetro `D`. Este último formato exige que as strings envolvidas estejam armazenadas em variáveis, não podendo utilizar expressões.

FullName() Adiciona diretório ao nome do arquivo

```
AnsiString FullName (AnsiString F, AnsiString D);
```

Se o nome de arquivo em `F` for especificado de forma relativa, será adicionado a ele o caminho absoluto contido em `D`. Retorna-se o resultado dessa operação, que consistirá apenas de `F` se este for absoluto.

ANEXO 2 - COMPONENTES DE VISUALIZAÇÃO

As bibliotecas apresentadas a seguir contém classes, funções e constantes que foram desenvolvidas para dar suporte ao *software* simulador de robôs manipuladores. Esses componentes foram, na medida do possível, desenvolvidos de forma genérica, a fim de possibilitar seu uso em outros *software*.

A2.1 MiscGL

Nessa biblioteca estão definidas as classes e funções relacionadas a representação de cores e materiais em OpenGL, bem como funções de conversão de e para o formato de cores utilizado no Windows e em C++ Builder. Além destas, ainda são definidas funções de construção de matrizes de rotações elementares.

Para utilizar `MiscGL`, deve-se incluir também a biblioteca `CrExtMth.lib` no projeto do *software* em desenvolvimento. No código do *software* deve ser adicionado o cabeçalho `MiscGL.h`.

A2.1.1 Representações de Cores

Em OpenGL, as cores podem ser representadas de forma indexada (considerando que o dispositivo gráfico trabalhe com paleta de cores) ou por composição das cores básicas vermelho, verde e azul (RGB), somando-se a estes o parâmetro alfa, que indica transparência (RGBA). Esta última representação é mais empregada, pois com ela podem ser usados recursos de incidência de luz e outros efeitos durante a renderização de primitivas OpenGL. Em `MiscGL` são definidas duas classes para representação de cores, `TCor` e `TCorf`. Enquanto a primeira trata da representação de uma cor através de quatro componentes básicas (RGBA) de tipo inteiro sem sinal de um byte (`GLubyte`), `TCorf` representa uma cor através dos mesmo quatro componentes, mas de tipo real (`GLfloat`), que podem variar entre 0.0 e 1.0. As operações básicas são as mesmas. Assim, existem construtores que permitem definir um novo objeto a partir de um valor do tipo `TColor` (definido nas bibliotecas do C++Builder) ou das componentes da cor. O acesso às componentes é público, o que permite modificar uma cor sem necessitar de métodos específicos. É feita a sobrecarga do operador de atribuição e de um operador de conversão

para `TColor`. Por fim, existe a possibilidade de acessar cor através de strings (`SetString()`, `GetString()`) além de se interagir diretamente com o contexto de renderização OpenGL (`SetGL()`, `GetGL()`).

A2.1.1.1 Classe TCor

TCor() — Construtores da classe

```
TCor(void);
TCor(TColor C);
TCor(GLubyte r, GLubyte g, GLubyte b, GLubyte a);
```

Existem três construtores para `TCor()`. O construtor vazio não faz inicializações. O segundo tipo de construtor recebe um valor `TColor` (definido nas biliotecas do C++ Builder) que consiste em um inteiro de 32 bits, em que as componentes RGB são definidas, respectivamente, dos bits menos significativos para os mais significativos. O terceiro formato permite definir cada componente da cor através de inteiros sem sinal que podem variar entre 0 e 255.

Operador TColor() — Converte a cor para TColor

```
operator TColor();
```

Este operador permite realizar a conversão de um objeto `TCor` para a representação segundo o tipo `TColor`, definido nas bibliotecas do C++Builder.

```
TColor C = TColor(Cor); // Cor é uma instância de TCor
```

SetString() — Define a cor através de uma string

```
void SetString(AnsiString A);
```

Recebe um item `AnsiString` (definido nas bibiotecas do C++Builder), e utiliza-o para definir as componentes de cor. Para tanto, essas componentes, na string, devem estar separadas por vírgulas e ter formato numérico inteiro.

```
Cor.SetString ("255,255,255,255"); // Cor é uma instância de TCor
```

GetString() — Retorna a cor em uma string

```
AnsiString GetString(void);
```

Este método gera uma string a partir dos valores dos componentes da cor, no formato `"rrr,ggg,bbb,aaa"`, retornando-o ao chamador do método.

```
AnsiString S = Cor.GetString(); // Cor é uma instância de TCor
```

SetGL() Define a cor do contexto de renderização

```
void SetGL(void);
```

Este método usa as componentes de cor para definir a cor corrente no contexto de renderização OpenGL ativo, através de `glColor*()`.

GetGL() Pega a cor do contexto de renderização

```
void GetGL(void);
```

Define o item de cor a partir da cor corrente no contexto de renderização OpenGL.

Componentes de Cor Acesso aos componentes individuais da cor

Os componentes da cor no formato RGBA podem ser acessados individualmente através do vetor RGBA[]. Na ordem, os elementos correspondem aos componentes vermelho, verde, azul e alfa.

```
Cor.RGBA[0] = 255;                    // Define em cor o vermelho máximo
Cor.RGBA[1]=Cor.RGBA[2] = Cor.RGBA[0]; // Faz os 3 componentes iguais
```

A2.1.1.2 Classe TCorf

TCorf() Construtores da classe

```
TCorf(void);
TCorf(TColor C);
Tcorf(TCorf &C);
TCorf(GLfloat r, GLfloat g, GLfloat b, GLfloat a);
```

Existem três construtores para `TCorf()`. O construtor vazio não faz inicializações. O segundo tipo de construtor recebe um valor `TColor`, que consiste em um inteiro de 32 bits, em que as componentes RGB são definidas, respectivamente, dos bits menos significativos para os mais significativos. O terceiro formato é o tradicional construtor de cópia, e o último formato permite definir cada componente da cor através de valores reais que podem variar entre 0 e 1.0.

Operador TColor() Converte a cor para TColor

```
operator TColor();
```

Este operador permite realizar a conversão de um objeto `TCorf` para uma representação do tipo `TColor`, definido nas bibliotecas do C++Builder.

```
TColor C = TColor(Corf); // Corf é uma instância de TCorf
```

SetString() Define a cor através de uma string

```
void SetString(AnsiString A);
```

Recebe um item `AnsiString` (definido nas bibiotecas do C++Builder), e utiliza-o para definir as componentes de cor. Para tanto, essas componentes, na string, devem estar separadas por vírgulas e ter formato numérico real, entre 0.0 e 1.0.

```
Corf.SetString ("1.0,1.0,1.0,1.0"); // Cor branca
```

GetString() Retorna a cor em uma string

```
AnsiString GetString(void);
```

Este método gera uma string a partir dos valores dos componentes da cor, no formato `"r,g,b,a"`, retornando-o ao chamador do método.

```
AnsiString S = Corf.GetString(); // Corf é uma instância de TCorf
```

SetGL() Define a cor do contexto de renderização

```
void SetGL(void);
```

Este método usa as componentes de cor para definir a cor corrente no contexto de renderização OpenGL ativo, através de `glColor*()`.

GetGL() Pega a cor do contexto de renderização

```
void GetGL(void);
```

Define o item de cor a partir da cor corrente no contexto de renderização OpenGL.

Operador = Atribuição

```
TCorf& operator= (TCorf C);
```

Este operador implementa a atribuição entre instâncias de TCorf.

Componentes de Cor Acesso aos componentes individuais da cor

Os componentes da cor no formato RGBA podem ser acessados individualmente através do vetor `RGBA[]`. O primeiro elemento corresponde ao componente vermelho, o segundo corresponde ao componente verde, o terceiro, ao componente azul e o quarto ao componente alfa.

A2.1.2 MATERIAIS

Para acrescentar realismo à renderização de objetos, não basta a descrição da cor das superfícies, mas também indicação do material que as compõem. Uma superfície

metálica de uma determinada cor causa uma impressão visual diferente de uma superfície de mesma cor, só que de material cerâmico, por exemplo.

No contexto OpenGL, materiais descrevem como superfícies devem ser renderizadas, através da definição de como superfícies refletem luz. Um material é composto por cinco propriedades, tipicamente descritas como cores no formato RGBA.

A propriedade *ambiente*(ambient) descreve como uma superfície reflete a luz ambiente existente[55], enquanto a propriedade *difusa*(diffuse) define como as luzes que incidem sobre a superfície devem ser refletidas[56]. As propriedades *especular*(specular) e *brilho*(shininess) referem-se aos aspectos reflexivos afetados pela posição do observador, e *emissividade*(emissive) corresponde à luz própria emitida pela superfície (útil no caso de modelagem de objetos como lâmpadas, abajures ou fogo).

A classe `TMaterial` modela materiais segundo a definição do OpenGL, composto pelas cinco propriedades acima comentadas. Estas podem ser definidas através dos construtores da classe ou através dos métodos `Set()` ou `SetString()`. Os atributos podem ser obtidos através dos métodos `Get()` ou `GetString()`. Os métodos `SetGL()` e `GetGL()` interagem com o contexto de renderização OpenGL corrente, definindo o material corrente ou atribuindo a um objeto `TMaterial` os atributos de material do contexto de renderização.

TMaterial Construtores da classe

```
TMaterial (void);
TMaterial (TCorf C0, TCorf C1, TCorf C2, TCorf C3, GLfloat S);
TMaterial (TMaterial &C);
```

Existem três construtores distintos para `TMaterial`. O primeiro é o construtor vazio. O segundo permite definir os atributos através de elementos `TCorf`, correspondendo os parâmetros aos atributos ambiente, difuso, especular, emissividade e brilho. Este é definido por um valor real entre 0 e 128. Por fim, tem-se o construtor de cópia, que inicializa uma instância de `TMaterial` com base em outra já existente.

[55] Luz ambiente corresponde à quantidade de luz que vem de todas as direções, iluminando por igual todos os objetos de uma cena.
[56] Nesse caso, as luzes são emitidas por fontes luminosas presentes à cena e direcionadas.

Operador = Atribuição

```
TMaterial& operator= (TMaterial C);
```

Este operador implementa a atribuição entre instâncias de `TMaterial`. Sua utilização é a mesma de um operador de atribuição convencional.

```
M1 = M2; // Copia os componentes de M2 para M1
```

Set() Define atributos de material

```
void Set (TCorf C, TMaterialKind K=mkAmbient);
```

Este método define atributos do material. O valor é passado como um item `TCorf`, e o tipo de atributo é definido pelo segundo parâmetro, sendo default o tipo ambiente (`mkAmbient`). Os demais atributos são definidos pelos nomes `mkDiffuse` (difuso), `mkSpecular` (especular), `mkEmission` (emissividade) e `mkShininess` (brilho). Para este, somente o primeiro elemento de `TCorf` é levado em conta, visto o brilho, que indica quão concentrado é o reflexo, ser definido por um único valor real, variando entre 0 e 128.

Get() Retorna atributos do material

```
TCorf Get (TMaterialKind K=mkAmbient);
```

Complementar a `Set()`, este método retorna um objeto `TCorf`, relativo a um dos atributos do material, definidos como em `Set()`.

SetString() Define atributos através de strings

```
void SetString (AnsiString A, TMaterialKind K=mkAmbient);
```

`SetString()` permite definir um atributo de material (como em `Set()`), ou todos os atributos de uma só vez (para `mkAll`). Cada atributo deve ser escrito em uma string como quatro componentes reais valendo entre 0 e 1, separados por vírgulas. Para brilho, somente uma componente é necessária, valendo entre 0 e 128. Para definir todos os atirbutos de uma vez, a string deverá conter todos os atributos separados por pontos-e-vírgula, na ordem `mkAmbient;mkDiffuse;mkSpecular;mkEmission;mkShininess`.

```
M.SetString("1.0,1.0,0.0,1.0"); // Define o atributo ambiente
M.SetString("0.0,0.0,0.0,1.0",mkEmission); // Atributo e missividade
M.SetString("1.0,1.0,0.0,1.0;1.0,1.0,0.0,1.0;1.0,1.0,0.0,1.0;\
            0.0,0.0,0.0,1.0;50.0",mkAll);// Define todos os atributos
```

GetString() Retorna atributos em strings

```
AnsiString GetString (TMaterialKind K=mkAmbient);
```

Complementar a `SetString()`, este método retorna em uma string um dos atributos do material ou todos de uma vez(`mkAll`). Neste caso, os atributos estarão separados por pontos-e-vírgula. Cada atributo é composto por quatro componentes entre 0 e 1 separados por vírgulas, exceto brilho, que é um valor real entre 0 e 128.

```
S=GetString(mkAll);      // Retorna todos os atributos do Material
```

SetGL() Define o material do contexto de renderização

```
void SetGL(void);
```

Este método usa as funções OpenGL `SetMaterial*()` para definir um novo material no contexto de renderização corrente.

GetGL() Retorna o material do contexto de renderização

```
void GetGL(void);
```

`GetGL()` retorna para o objeto `TMaterial` os atributos do material em uso no contexto de renderização corrente.

A2.1.3 FUNÇÕES DIVERSAS

São um conjunto de funções para conversão de representações de cores, traçados de eixos coordenados e rotações elementares.

Color2RGBA(), RGBA2Color() Conversão de unsigned long para RGBA

```
void          Color2RGBA (unsigned long Color, GLubyte *RGBA);
unsigned long RGBA2Color (GLubyte *RGBA);
```

Cores RGBA podem ser representadas pelo tipo `unsigned long`, na forma `0xaabbggrr`. As funções em questão permitem separar as componentes de cores em um vetor de quatro elementos `GLubyte` (inteiros de um byte sem sinal) ou agrupar as componentes definidas desta forma em um `unsigned long`.

TColor2RGBA(), RGBA2TColor() Conversão de TColor para RGBA

```
void TColor2RGBA (TColor Color, GLubyte *RGBA);
void TColor2RGBA (TColor Color, GLfloat *RGBA);
TColor RGBA2TColor (GLfloat *RGBA);
TColor RGBA2TColor (GLubyte *RGBA);
```

O tipo `TColor` é definido nas bibliotecas C++Builder, consistindo em um `unsigned long` onde o componente mais significativo tem um significado diferente do parâmetro alfa usado em OpenGL. Assim, essas funções separam as componentes RGB de um valor `TColor` e definem o valor alfa como 1.0(255). De forma contrária, as funções que agregam as componentes em um valor `TColor` definem esse componente como 0. As variações desses métodos são devidas ao tipo dos componentes, inteiro (`GLubyte`) ou real (`GLfloat`).

TCor2TCorf(),TCorf2TCor() Conversão entre representações de cor

```
void TCor2TCorf (TCor C1, TCorf &C2);
void TCorf2TCor (TCorf C1, TCor &C2);
```

Estes métodos realizam a conversão entre as duas representações de cores, cuja diferença esta no tipo das componentes, que em `TCor` são inteiros de um byte sem sinal e em `TCorf` são reais.

GLAxis() Traça eixos coordenados

```
void GLAxis(void);
void GLAxis(double X);
```

Estas funções desenham eixos coordenados em um contexto de renderização OpenGL. Se passado um parâmetro real, a função traça o eixo escalonado do valor passado. Do contrário, os eixos são traçados com tamanho igual a 1.0. São desenhados também as letras X, Y e Z, indicando os eixos coordenados.

RotX(),RotY(),RotZ() Gera matrizes de rotação

```
void RotX (double *e, double A);
void RotY (double *e, double A);
void RotZ (double *e, double A);
```

As três funções têm por finalidade a composição de uma matriz de transformação homogênea (v. Anexo 1), representando uma rotação de `A` graus segundo o eixo `X` (`RotX()`), o eixo `Y` (`RotY()`) ou o eixo `Z` (`RotZ()`). A matriz será armazenada no endereço referenciado por `e`, no formato utilizado em OpenGL, em que as matrizes são formadas em ordem das colunas.

A2.2 TCrGLCam - Câmera Sintética Usando OpenGL

A classe `TCrGlCam` encapsula, em um componente VCL (Visual Component Library) do C++ Builder, os mecanismos necessários para criar e manter um contexto de renderização OpenGL em uma área gráfica no Windows. Instâncias de `TCrGlCam` funcionam como *câmeras sintéticas*, cuja finalidade é mostrar um pedaço de um cenário virtual, segundo uma posição e orientação relativas ao sistema de coordenadas desse cenário. Além de determinar o ponto de visão e a orientação, é possível, neste componente, definir as características de iluminação da cena, tornando a renderização mais realista.

A posição e a orientação são propriedades fundamentais de `TCrGLCam`. A posição é especificada através de `Position`, cujas componentes reais definem a posição através de

coordenadas no espaço cartesiano. Assim como `Position`, a propriedade `Orientation` é formada por três componentes, que correspondem aos ângulos de azimute, inclinação e giro, todos em graus. A Figura A2.1 ilustra como essas propriedades são definidas.

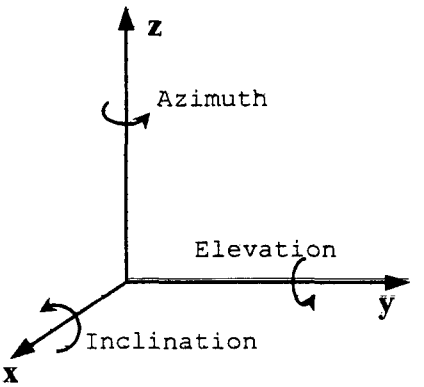


Figura A2.1 Valores `Azimuth`, `Elevation` e `Inclination` da orientação da câmera sintética

O tipo de projeção influencia diretamente como a cena é renderizada na câmera sintética. A propriedade `Projection` permite definir dois tipos de projeções através da sub-propriedade `Kind`. Se esta for `pkOrtographic`, a cena é desenhada segundo uma projeção ortográfica, também chamada de paralela, onde a escala e os ângulos dos objetos são mantidos, a despeito de sua distância aparente (interessante para desenho técnico). O tipo `pkPerspective` acrescenta perspectiva ao desenho da cena, fazendo com que objetos mais distantes do observador pareçam menores, podendo inclusive modificar as proporções dos objetos. O volume de visão, em ambos os casos, é a parte da cena que será desenhada, e as sub-propriedades `Width`, `Height`, `Near`, `Far` e `FieldOfView` definem este volume, de acordo com o tipo de perspectiva (v. Figura A2.2).

O uso de iluminação na renderização de cenas acrescenta realismo pela modificação das cores dos objetos causada pela influência das fontes luminosas, embora com um custo maior de processamento. A classe `TCrGlCam` encapsulou a definição de iluminação da cena, permitindo que uma mesma cena possa ser vista em várias câmeras de formas diferentes, de acordo com a definição de iluminação de cada uma. Para tanto, deve-se ativar este recursos através da propriedade lógica `Lighting`. Após, se foram utilizadas fontes individuais de luz, estas devem ser ativadas pela propriedade lógica `Light[i]`, onde o índice define a fonte de luz desejada. O número máximo de fontes de luz disponíveis é

informado pela propriedade `MaxLights`. As características das fontes de luz são acessadas por `SetLightPosition()`, `GetLightPosition()`, `SetLightDirection()`, `GetLightDirection()`, `SetSpotLight()`, `GetSpotLight()`, `SetLight()` e `GetLight()`. Estes dois últimos métodos também ajudam a definir as características da iluminação ambiente, que está presente mesmo sem nenhuma outra fonte de luz.

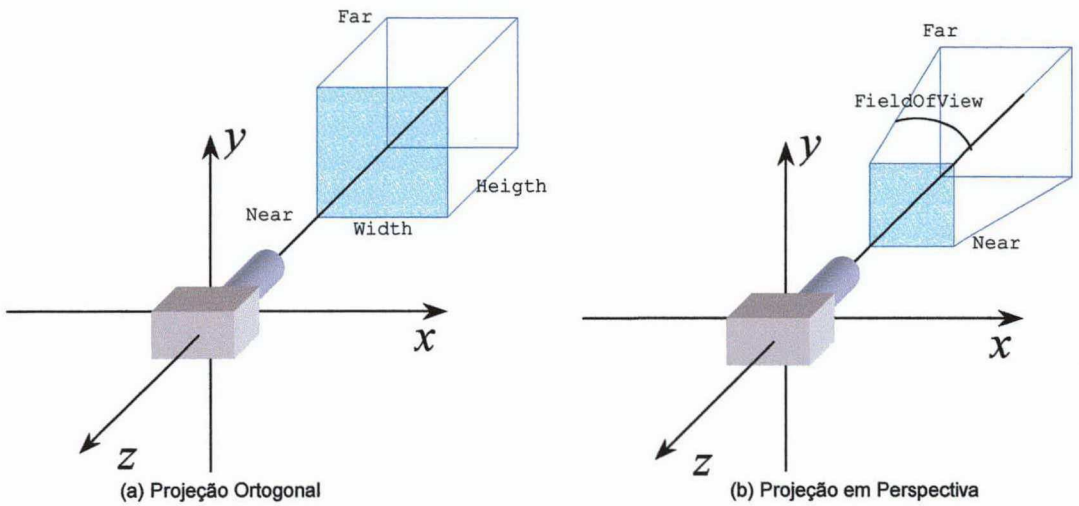


Figura A2.2 Tipos de projeção e seus volumes de visão

O desenho é feito por procedimentos escritos pelo usuário, utilizando primitivas OpenGL ou de bibliotecas derivadas desta. Pode-se vincular o código de desenho ao evento `OnPaint` do componente. Assim, o desenho será feito automaticamente, a cada ativação da janela. Se desejado, pode-se também assumir o controle manual do desenho, colocando o bloco de código que realiza o desenho entre os métodos `BeginPaint()` e `EndPaint()` da câmera. O primeiro ativa o contexto de renderização da câmera, enquanto o último força o redesenho da cena mostrada pela câmera. Se desejado, o eixo do sistema de referência da cena é automaticamente desenhado, se a propriedade lógica `AxisVisible` for verdadeira.

Position Posição da câmera

A propriedade `Position` é formada por três componentes reais, correspondentes às coordenadas cartesianas da posição da câmera sintética em relação à origem do sistema de coordenadas da cena. Eles podem ser acessados tanto individualmente (por `X`, `Y` e `Z`) como através dos métodos `Set()` e `AsVector()`, que usam instâncias de `Vector` (v. `CrExtMth`). Esta propriedade está disponível em tempo de projeto através do Object Inspector.

```
C=>Position=>X = 0.0;                    // Define a coordenada X
C->Position->Set(Vector(2.0,2.0,2.0)); // Define a posição da câmera
Vector V = C->Position->AsVector();      // Retorna a posição da câmera
```

Orientation Orientação da câmera

A propriedade `Orientation` é formada por três componentes reais. `Azimuth` é a inclinação da câmera segundo o eixo vertical desta (0° corresponde à câmera na direção do eixo `-Z` do sistema de coordenadas da cena). `Elevation` refere-se à inclinação da câmera segundo seu eixo horizontal `Y` (0° corresponde à câmera paralela ao plano `XY` do sistema de coordenadas da cena). `Inclination` define o movimento da câmera segundo sua linha de visão, dando a impressão de um giro desta enquanto foca fixamente uma direção. Todos esse valores são expressos em graus. Como Position, os componentes podem ser acessados tanto individualmente (por `Azimuth`, `Elevation` e `Inclination`) como através dos métodos `Set()` e `AsVector()`, que usam instâncias de `Vector`. Esta propriedade está disponível em tempo de projeto através do Object Inspector.

```
C->Orientation->Inclination = 45.0; //Inclinação da "cabeça"
C->Orientation->Set(Vector(A,E,I)); //Orientação definida por um vetor
Vector V = C->Orientation->AsVector(); // Retorna a orientação
```

Move() Movimenta a câmera

```
void __fastcall Move (Vector P);
void __fastcall Move (double dx, double dy, double dz);
void __fastcall Move (TOrientAngle O, double A);
```

O método `Move()`, em sua primeira e segunda formas, translada a câmera de forma relativa à posição corrente desta. A terceira forma deste método permite rotacionar a câmera ao redor de um de seus eixos (`oaAzimuth`, `oaElevation`, `oaInclination`), definido pelo parâmetro `O`, de um ângulo `A` graus.

Projection Características do volume de visão

Esta propriedade define o tipo de projeção a ser utilizada na renderização, bem como o volume de visão utilizado para *recortar* o que vai ser mostrado pela câmera. `Kind` é a sub-propriedade que especifica se a projeção será ortográfica (`pkOrthographic`) ou em perspectiva (`pkPerspective`). Enquanto o primeiro tipo de projeção mantém o tamanho e as proporções dos objetos a serem desenhados, o segundo tipo causa a distorção dos objetos em função da distância aparente entre estes e o observador, a fim de proporcionar a sensação de profundidade da cena. O volume de visão para projeções ortográficas é definido pelas sub-propriedades `Width` (largura), `Height` (altura), `Near` (distância da face mais próxima) e `Far` (distância da face mais afastada). O volume de visão está sempre

centralizado segundo a câmera. Para projeções em perspectiva, o volume de visão corresponde a um tronco de pirâmide, sendo definido pelas sub-propriedades `Near`, `Far` e `FieldOfView` (campo de visão). Este é expresso em graus, e afeta diretamente o efeito de profundidade (v. Figura A2.2). Esta propriedade é acessível em tempo de projeto através do Object Inspector.

```
C->Projection->Kind = pkPerspective; // Projeção em perspectiva
C->Projection->Near = 1.0;           // Volume de visão
C->Projection->Far  = 10.0;          // Volume de visão
C->Projection->FieldOfView = 45.0;   // Volume de visão
C->Projection->Width = 2.5;          // Volume de visão
C->Projection->Heigth = 2.0;         // Volume de visão
```

Lighting, Light[] Ativação de iluminação e fontes de luz

A propriedade lógica `Lighting` é que define o uso de iluminação na renderização de cenas. Por default, é falsa, indicando que nenhuma iluminação será considerada. Ao ser ativada, leva-se em conta na renderização a intensidade de luz ambiente (que não tem uma fonte, como se estivesse igualmente presente em toda a cena) e as intensidades e direções das fontes de luz ativas. Estas são ativadas através da propriedade `Light[i]`, onde `i` pode variar entre 0 e `MaxLights`, definindo a fonte de luz a ser ativada. Por default, apenas a fonte de luz 0 está ativa, estando as outras desativadas (false).

MaxLights Número máximo de fontes de luz

Esta propriedade somente leitura informa o número de fontes disponíveis para a câmera sintética. Esta depende da versão da biblioteca OpenGL utilizada no sistema.

SetLight(), GetLight() Acesso às características de luz

```
void   __fastcall SetLight (TCorf C, int L=-1, int K=GL_AMBIENT);
TCorf  __fastcall GetLight (int L=-1, int K=GL_AMBIENT);
```

A função `SetLight()` define as características da luz ambiente ou da luz emitida por uma fonte de luz. A característica é indicada pelo parâmetro `K`, podendo ser `GL_AMBIENT` (luz ambiente), `GL_DIFFUSE` (valor da luz difusa) ou `GL_SPECULAR` (define a cor do reflexo que ajudará no reflexo emitido em um objeto). O valor da característica é passado no parâmetro `C`, que é do tipo `TCorf`. A fonte de luz acessada é `L`. Se não informada (default=-1), corresponde à luz ambiente. Neste caso, a única característica é a ambiente. A contraparte de `SetLight()` é `GetLight()`, que retorna os valores de luz.

SetLightPosition(), GetLightPosition() Acesso à posição das fontes de luz

```
void __fastcall SetLightPosition (Vector P, bool Positional=false, int L=0);
void __fastcall SetLightPosition (GLfloat X,GLfloat Y,GLfloat Z,
                                  bool Positional,int L);
void __fastcall GetLightPosition (Vector &P, bool &Positional, int L=0);
void __fastcall GetLightPosition (GLfloat &X, GLfloat &Y, GLfloat &Z,
                                  bool &Positional, int L);
```

`SetLightPosition()` define a posição da fonte de luz. Esta é muito importante, pois as fontes emitem a luz a partir de sua posição, e o ângulo de incidência dos raios de

luz em uma superfície influencia a renderização desta. São duas as formas de especificar a posição. Na primeira, utiliza-se um parâmetro `Vector`, enquanto na segunda as componentes `X`, `Y` e `Z` são informadas separadamente. O parâmetro `Positional`, se verdadeiro, indica que a luz terá uma direção a ser definida pelo usuário através de `SetLightDirection`. Do contrário, a direção será (0,0,-1). A fonte de luz é especificada pelo parâmetro `L`. Da mesma forma que `SetLightPosition()`, a função `GetLightPosition()` tem duas formas equivalentes.

SetLightDirection(), GetLightDirection() Direção da luz emitida pelas fontes

```
void __fastcall SetLightDirection (Vector P, int L=0);
void __fastcall SetLightDirection (GLfloat X, GLfloat Y, GLfloat Z, int L=0);
Vector __fastcall GetLightDirection (int L=0);
void __fastcall GetLightDirection (GLfloat &X, GLfloat &Y,GLfloat &Z,int L=0);
```

A função `SetLightDirection()` é utilizada para definir a direção de emissão da luz, através de um vetor unitário. As duas formas deste método são equivalentes, mudando apenas na especificação do vetor (um item `Vector` na primeira ou três componentes separados na segunda). A função complementar, `GetLightDirection()`, retorna o valor do vetor de posição. A direção da luz emitida por uma fonte só tem sentido se esta for posicional (definida por `SetLightPosition()`).

SetSpotLight() , GetSpotLight() Características de focalização da luz

```
void __fastcall SetSpotLight (GLfloat X,int K=GL_SPOT_CUTOFF,int L=0);
GLfloat __fastcall GetSpotLight (int K=GL_SPOT_CUTOFF, int L=0);
```

`SetSpotLight()` define as características de foco da iluminação. A fonte é indicada pelo parâmetro `L`, e o tipo de característica é definido por `K`, podendo ser `GL_SPOT_CUTOFF` (ângulo de abertura da fonte de luz, entre 0 ou 90 graus, ou 180 graus), `GL_SPOT_EXPONENT` (distribuição no cone de luz, variando entre 0 e 128), `GL_CONSTANT_ATTENUATION`, `GL_LINEAR_ATTENUATION` ou `GL_QUADRATIC_ATTENUATION`. Estes últimos definem características de atenuação da luz em função da distância percorrida. O valor é passado em `X`. A função que retorna esses valores é `GetSpotLight()`.

BeginPaint(), EndPaint() Delimitam o desenho

```
void __fastcall BeginPaint(void);
void __fastcall EndPaint(void);
```

Estes métodos devem ser usados para delimitar um bloco de código de primitivas OpenGL de desenho a serem renderizadas na câmera sintética. `BeginPaint()` ativa o contexto de renderização da câmera, se for o caso. `EndPaint()` força a atualização da cena renderizada pela câmera. Sem estas primitivas, o desenho pode não ser renderizado.

OnPaint Método de resposta ao evento de pintura

O evento OnPaint associa um método para o desenho automático de uma cena, cada vez que for necessária a atualização do componente `TCrGlCam` ou da janela onde ele está

situado. Se utilizado, o método escrito pelo usuário associado ao evento `OnPaint` não precisa empregar `BeginPaint()` e `EndPaint()`. Este evento, porém, pode causar efeito de *piscada*, não sendo recomendado para fazer animação.

Clear(), Color Limpa o contexto de renderização

```
void __fastcall Clear(TCorf C);
```

A função `Clear()` limpa o contexto de renderização da câmera sintética, preenchendo-o com uma cor definida no parâmetro C. A cada atualização, o método `Clear()` é automaticamente chamado, com o valor definido na propriedade `Color`.

AxisVisible, AxisColor, AxisScale Características do eixo da cena

A câmera pode desenhar automaticamente os eixos de referência do ambiente virtual onde está situada, a fim de auxiliar a localização do usuário na cena. A propriedade `AxisVisible`, se falsa, inibe esse desenho automático. A cor é, por default, branca, mas pode ser alterada por `AxisColor`. A propriedade `AxisScale` define o fator de escalonamento do eixo, sendo 1.0 por default.

Culling, CullFace, FrontFaceIs Corte de faces

Na renderização de cenas, pode-se melhorar a performance pela exclusão de superfícies que não serão mostradas nas cenas. Por exemplo, em poliedros convexos, as faces internas não serão vistas por quem não está no interior. O corte de superfícies não visíveis na renderização é feito pela propriedade `Culling`, que é verdadeira por default. A face a ser cortada pode ser a frontal (`glFront`) ou a traseira(`glBack`, default), definida na propriedade `CullFace`. Para definir qual é a face frontal de uma superfície, utiliza-se a propriedade `FrontFaceIs`, que, se tiver o valor `glCCW`, é considerada como aquela cuja normal é definida pelo seguimento dos pontos que a definem no sentido anti-horário, ou no sentido horário, se o valor da propriedade for `glCW`.

PolygonMode, ShadeModel Modo de desenho de polígonos

Os polígonos são definidos por conjuntos de pontos, que são desenhados de acordo com a propriedade `PolygonMode`. Se este for `glFill` (default), os polígonos serão preenchidos. Se for `glLine`, será traçado apenas o contorno dos polígonos. Se for `glPoint`, apenas os pontos serão marcados. No caso de `PolygonMode` ser `glFill`, o preenchimento se dará de acordo com a definição da propriedade `ShadeModel`. Se esta for `glSmooth` (default), o preenchimento do polígono será feito levando em conta as cores de todos os pontos que o definem, através da interpolação das cores. Se for `glFlat`, apenas a cor do primeiro ponto será utilizada no preenchimento. Quanto mais elaborado o preenchimento do polígono, maior a carga computacional envolvida.

DepthTest Teste de profundidade dos polígonos

Esta propriedade, ativada por default, define a ordem de desenho dos objetos de uma cena, em função de sua distância do ponto de observação. Se desativada, pode haver sobreposição dos elementos da cena.

A2.3 TCrLineGraph - Traçador de Gráficos

A classe `TCrLineGraph` define um componente VCL (Visual Component Library) do C++ Builder que pode ser utilizado para traçado de gráficos de funções ou conjuntos de valores armazenados em matrizes. O componente desenha um gráfico com todos os elementos necessários, como legendas, títulos e escalas, além do próprio traçado dos valores, em uma escala fixa ou escalonado automaticamente. A Figura A2.3 mostra esses elementos.

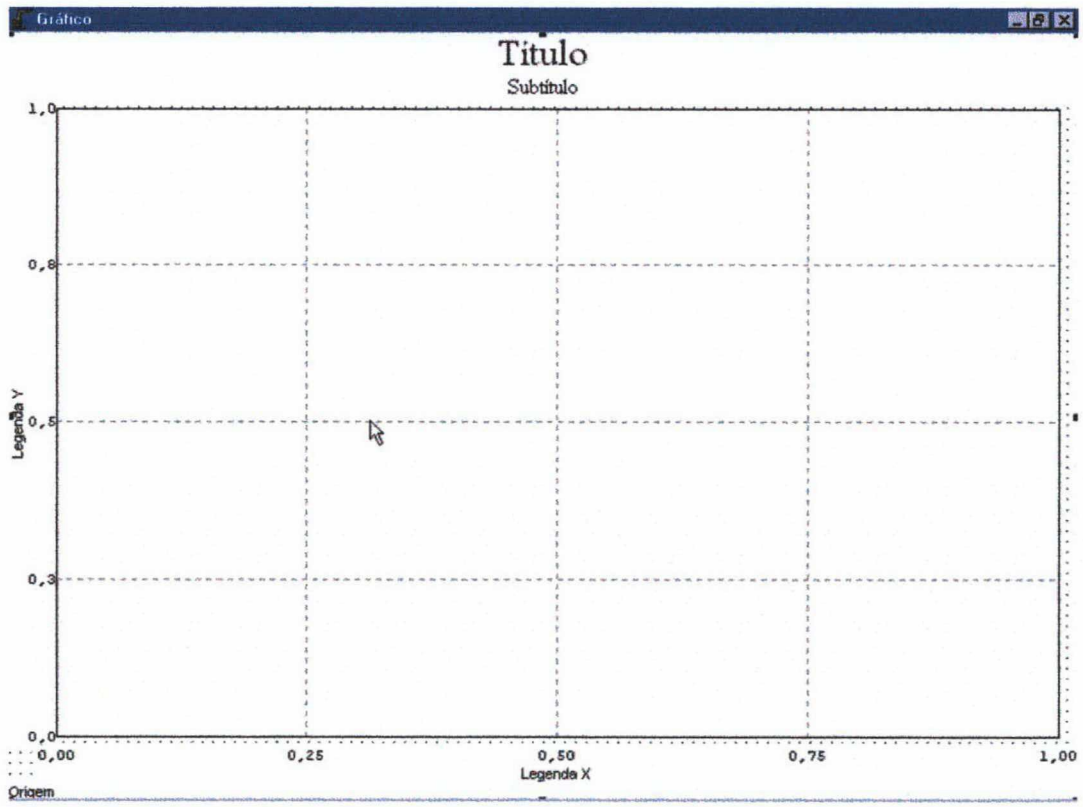


Figura A2.3 Componente `CrLineGraph`

Os elementos de identificação dos gráficos são definidos por propriedades como Title, SubTitle, Source, CaptionX, CaptionY, ScaleX e ScaleY. O grid e as bordas também são definidos por propriedades específicas.

As fontes de dados utilizadas para os traçados dos gráficos são definidas por métodos como `Set()`, `Add()` e `Del()`, além das propriedades `Initial`, `Final` e `Step`. Os atributos das linhas dos gráficos são definidos por propriedades como `GraphPenColor`, `GraphPenWidth` e `GraphPenstyle`.

Initial, Final, Step — Valores inicial, final e passo de contagem

Estas propriedades são utilizadas para definir o intervalo de variação da variável independente e seu passo. Normalmente, esta é a componente horizontal do gráfico, e se a componente vertical for uma função, esta será calculada tendo os valores como parâmetros. No caso da componente horizontal ser definida por uma função, será desenhado um gráfico paramétrico, utilizando esse intervalo para calcular os valores dos componentes horizontais.

GraphCount — Número de gráficos a traçar

Esta propriedade somente leitura informa quantas fontes distintas de dados estão associadas ao componente gráfico.

GraphPenColor, GraphPenWidth, GraphPenStyle — Atributos do gráfico

Estas três propriedades definem como a linha de um gráfico em particular será traçado, sendo definido pelo índice utilizado nestas propriedades do tipo vetor. `GraphPenColor` define a cor a ser utilizado no traçado do gráfico. `GraphPenWidth` define a largura do traço, e `GraphPenStyle` o estilo de traçado (contínuo, tracejado, traço ponto) se a largura for igual a 1.

GraphVisible — Visibilidade de um gráfico

Esta propriedade lógica vetor especifica se uma determinada fonte de dados deve ser traçada.

ShowAxis, Axis, AxisColor — Atributos dos eixos

O componente gráfico, por default, traça os eixos vertical e horizontal, se os dados passam pelo valor 0 horizontal ou o vertical. `ShowAxis` determina se os eixos deverão ser traçados ou não. `Axis` pode assumir os valores `lsVertical` (somente o eixo vertical será traçado), `lsHorizontal` (somente o eixo horizontal será traçado) ou `lsBoth` (ambos os eixos serão traçados). `AxisColor` define a cor com a qual os eixos serão traçados.

ShowGrid, Grid, GridPen, Color Atributos da grade

Ë possível desenhar uma grade sob o gráfico, a fim de auxiliar a sua análise. A propriedade `ShowGrid`, verdadeira por default, define se a grade deverá ser traçada ou não. `Grid` pode assumir os valores `lsHorizontal` (somente linhas horizontais serão traçadas), `lsVertical` (somente linhas verticais serão traçadas) ou `lsBoth` (ambas as linhas serão traçadas). `GridPen` é uma propriedade do tipo `TPen`, que pode ser expandida para definir os atributos da caneta de traçado da grade. A propriedade Color define a cor do fundo sobre o qual a grade e os gráficos serão traçados.

ShowBorder, BorderColor Atributos da borda

O gráfico é circundado por uma borda, que será desenhada se `ShowBorder` for verdadeiro (default). `BorderColor` estabelece a cor das linhas da borda.

GraphVisible Visibilidade de um gráfico

Esta propriedade vetor especifica se uma determinada fonte de dados deve ser traçada.

Set(), Get() Fontes dos dados dos gráficos

```
void __fastcall Set (double I, double F, double S=-1.0);
void __fastcall Set (void *P, int C=-1);
void __fastcall Set (int N, void *P, int C=-1);
void __fastcall Set (int N, void *P, int C, TColor Cor,
                     int W, TPenStyle S, AnsiString A, bool V);
void* __fastcall Get (int N, int &C);
```

O método `Set()` tem diversas variações, que permitem definir a origem das componentes verticais dos gráficos e da componente horizontal (comum a todos os gráficos. A primeira variação eqüivale a definir os valores de `Initial`, `Final` e `Step` simultaneamente. A segunda variação define a origem dos componentes horizontais dos pontos dos gráficos, recebendo o endereço de uma função ou de uma matriz. Neste caso, o valor de `C` identifica a coluna de onde os dados serão retirados.

O componente `TCrLineGraph` contém uma lista das fontes de dados que formam as componentes verticais dos pontos dos gráficos. As duas outras variações de `Set()` definem os atributos dos elementos desta lista. O parâmetro `N` identifica o elemento da lista que será definido. `P` é o endereço de uma função ou uma matriz que será a fonte dos dados. Se for uma matriz, o parâmetro `C` é obrigatório, identificando a coluna de onde os dados serão retirados. Se utilizada a terceira variação de `Set()`, as características da linha do gráfico correspondente serão determinadas automaticamente. Se usada a última variação, os atributos podem ser definidos juntamente com a origem dos dados.

O método `Get()` é complementar a `Set()`, retornando o endereço da fonte dos dados correspondente ao gráfico definido pelo parâmetro `N`. Se for uma matriz, o parâmetro `C` será diferente de -1, indicando a coluna utilizada.

Add(), Del(), MoveBy() Lista das fontes dos dados dos gráficos

```
void __fastcall Add (void *P, int C=-1, TColor Cor=clBtnFace, int W=1,
     TPenStyle S=psSolid, AnsiString A=AnsiString(""), bool V=true, int N=-1);
void __fastcall Del (int N=-1);
void __fastcall MoveBy (int N, int C=1);
```

Estas funções controlam a lista de fontes dos dados. `Add()` adiciona uma fonte à lista de gráficos a serem traçados. O parâmetro `P` é o endereço da fonte dos dados, e se esta for uma matriz, o parâmetro `C` deverá conter a coluna de onde os dados serão extraídos. Os demais parâmetros definem as características do gráfico traçado, como cor, largura e estilo do traçado. `Del()` retira um elemento da lista de gráficos.

Os gráficos são traçados na ordem com na qual as fontes de dados aparecem na lista. O método `MoveBy()` modifica essa ordem, movendo o elemento `N` da lista `C` posições acima(se negativo) ou abaixo na lista.

Title, SubTitle, Source Elementos textuais do gráfico

Estes três atributos definem o título, o subtítulo, mostrados acima do gráfico, e o comentário mostrado abaixo do gráfico, normalmente usado para indicar a origem dos dados. Estas são do tipo `TLabelAttr`, que é subdividido em várias proriedades.

A subpropriedade `Color` define a cor do fundo da área do elemento considerado. Em `Text` escreve-se o texto que será escrito. `Alignment` pode assumir os valores `asCenter` (texto centralizado), `asLeft` (texto alinhado a esquerda) ou `asRight` (texto alinhado a direita). `Font` define as características das fontes utilizadas para escrever o texto. `Visible`, se verdadeiro (default), indica que o elemento será escrito no componente.

CaptionX, CaptionY Legendas dos eixos

As duas propriedades definem as legendas do eixo horizontal e do eixo vertical, respectivamente. Sendo do tipo `TLabelAttr`, aplicam-se as mesmas explicações das propriedades `Title`, `SubTitle` e `Source`.

ScaleX, ScaleY Escalas do gráfico

As duas propriedades definem os fatores de escala dos eixos horizontal e vertical, respectivamente, além dos valores que serão mostrados. Ambas são do tipo `TScaleAttr`, que pode ser expandida em subpropriedades.

A subpropriedade `Font` define os atributos dos valores escritos nas legendas. `Color` define a cor do fundo das escalas. `AutoSize` especifica se a dimensão da legenda será automaticamente calculada em função dos valores mostrados ou se obedecerá ao valor especificado em `Size` (pixels). Os valores são mostrados com no máximo `Places` algarismos antes do ponto decimal, e `Decimals` algarismos após o ponto decimal. `ShowSignal` define se o sinal deve ser mostrado mesmo quando os valores são positivos.

A escala é definida em função dos dados, se `AutoLimits` for verdadeiro, ou em função das propriedades `Max` e `Min`, do contrário. `Frequency` define o intervalo com que os valores são escritos na escala. `Visible` define se a escala será mostrada ou não no gráfico.

ANEXO 3 - MODELOS DE MANIPULADORES E CONTROLADORES

A3.1 MANIPULADOR PLANO COM DOIS GRAUS DE LIBERDADE

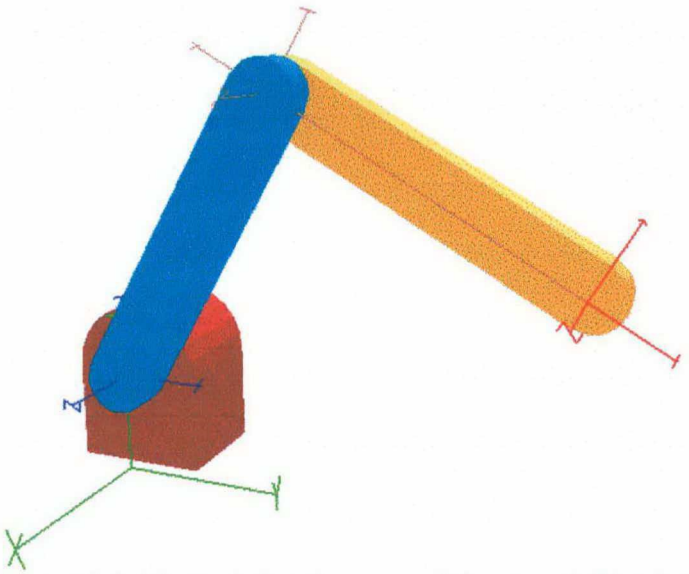


Figura A3.1. Manipulador plano com dois graus de liberdade

```
[CONFIG]
Name=Manipulador Plano 2 DOF
DOF=2
Flags=Euler|NoLimits|Dynamics|Rotor|Friction
Position=0.0,0.0,0.3
Orientation=0.0,pid2,pid2
[DESCRIPTION]
Manipulador plano de 2 graus de liberdade
Parametros extraidos de Sciavicco e Siciliano (Exercicio 4.2)
[DH-DYNAMICS]
#VJ;a,alfa,d,teta,m,lcx,lcy,lcz,Ixx,Iyy,Izz,Ixy,IXz,Iyz,Fst,Fvs,Im,Kr,Zmx,Zmy,Zmz
3;1,0,0,pid3,50,-0.5,0,0,0.0,0.0,10.0,0.0,0.0,0.0,0,0,0.01,100,0,0,1
3;1,0,0,-pid2,50.0,-0.5,0,0,0.0,0.0,10.0,0.0,0.0,0.0,0,0,0.01,100,0,0,1
[MODELLIST]
File=barra.m3d
File=base.m3d
[LINKS]
Link=0
Visibility=Visible
Axis=Visible;0.0,0.0,1.0,1.0;0.25
Model=12;0.0,0.0,-0.051;0.0,0.0,0.0;1,1,1
Link=1
Visibility=Visible
Axis=Visible;0.8,0.6,0.5,1.0;0.25
Model=11;0.0,0.0,0.0;0.0,0.0,0.0;1,1,1
Link=2
Visibility=Visible
Axis=Visible;1.0,0.0,0.0,1.0;0.25
Model=11;0.0,0.0,-0.101;0.0,0.0,0.0;1,1,1
#A linha abaixo define o material do elo 2, anexa à linha Model= acima
#1.0,0.804,0.51,1.0;1.0,0.804,0.51,1.0;0.85,0.85,0.25,1.0;0,0,0,1.0;50.0
```

A3.2 MANIPULADOR SCARA

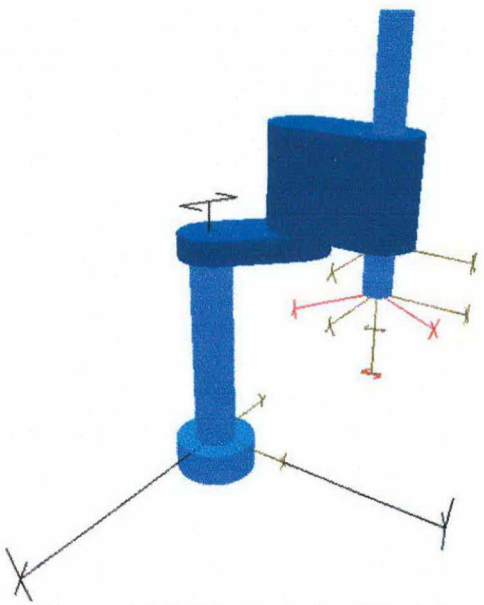


Figura A3.2. Manipulador SCARA

```
[CONFIG]
Name=Manipulador Inter do LAI/DAS/UFSC - Tipo SCARA
DOF=4
Flags=Euler|Limits|Dynamics|NoRotor|NoFriction
Position=0.0,0.0,0.0
Orientation=pid2,0.0,0.0
[DESCRIPTION]
Manipulador SCARA instalado no LAI/DAS/UFSC
Construido na Universidade Tecnica de Zurique(ETH)
Parametros extraidos de Bier, Dissertacao de Mestrado

[DH-DYNAMICS]
#VJ;a,alfa,d,teta,m,lcx,lcy,lcz,Ixx,Iyy,Izz,Ixy,IXz,Iyz,Fst,Fvs,Im,Kr,Zmx,Zmy,Zmz
3;0.25,0.0,0.665, 0.0,11.4,-0.118,0,0,0,0,0.23,0,0,0,0,0,0,1,0,0,1
3;0.25, pi,  0.0, 0.0,19.5,-0.116,0,0,0,0,0.16,0,0,0,0,0,0,1,0,0,1
2;0.0, 0.0,0.240, 0.0, 2.0,     0,0,0,0,0, 0.1,0,0,0,0,0,0,1,0,0,1
3;0.0, 0.0,  0.05, 0.0, 1.5,    0,0,0,0,0, 0.1,0,0,0,0,0,0,1,0,0,1

[LIMITS]
#qmin,qmax,vmin,vmax,amin,amax,tmin,tmax
-2.25,2.25,  -3.0,  3.0, -80.0, 80.0,-333.0,333.0
-1.90,1.90,  -3.0,  3.0,-100.0,100.0,-157.0,157.0
 0.14, 0.4,-0.888,0.888,  -3.0,  3.0,-3.493,3.493
 -2.5, 2.5, -20.0, 20.0,-500.0,500.0,-33.53,33.53

[MODELLIST]
File=Cilindro.m3d

#Barra que compoe o elo de um manipulador 2 DOF
BeginModel
NAME=Barra
ID=12
```

```
VERTEXNUM=40
FACENUM=22
AMBIENT=0.0,0.353,0.588,1.0
DIFFUSE=0.0,0.353,0.588,1.0
SPECULAR=0.0,0.196,0.5,1.0
SHININESS=50.0
EMISSION=0,0,0,1.0
V= 0.000, 0.090,0.07;-0.250, 0.090,0.07;-0.295, 0.078,0.07;-0.314, 0.064,0.07
V=-0.328, 0.045,0.07;-0.340, 0.016,0.07;-0.340,-0.016,0.07;-0.328,-0.045,0.07
V=-0.314,-0.064,0.07;-0.295,-0.078,0.07;-0.250,-0.090,0.07; 0.000,-0.090,0.07
V= 0.045,-0.078,0.07; 0.064,-0.064,0.07; 0.078,-0.045,0.07; 0.090,-0.016,0.07
V= 0.090, 0.016,0.07; 0.078, 0.045,0.07; 0.064, 0.064,0.07; 0.045, 0.078,0.07
V= 0.000, 0.090,0.00; 0.045, 0.078,0.00; 0.064, 0.064,0.00; 0.078, 0.045,0.00
V= 0.090, 0.016,0.00; 0.090,-0.016,0.00; 0.078,-0.045,0.00; 0.064,-0.064,0.00
V= 0.045,-0.078,0.00; 0.000,-0.090,0.00;-0.250,-0.090,0.00;-0.295,-0.078,0.00
V=-0.314,-0.064,0.00;-0.328,-0.045,0.00;-0.340,-0.016,0.00;-0.340, 0.016,0.00
V=-0.328, 0.045,0.00;-0.314, 0.064,0.00;-0.295, 0.078,0.00;-0.250, 0.090,0.00
F=20:0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19
F=20:20,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39
F=4:0,20,39,1
F=4:10,30,29,11
F=4:0,19,21,20
F=4:19,18,22,21
F=4:18,17,23,22
F=4:17,16,24,23
F=4:16,15,25,24
F=4:15,14,26,25
F=4:14,13,27,26
F=4:13,12,28,27
F=4:12,11,29,28
F=4:10,9,31,30
F=4:9,8,32,31
F=4:8,7,33,32
F=4:7,6,34,33
F=4:6,5,35,34
F=4:5,4,36,35
F=4:4,3,37,36
F=4:3,2,38,37
F=4:2,1,39,38
EndModel

[LINKS]
Link=0
Visibility=Visible
Axis=Visible;1.0,1.0,0.0,1.0;0.15
Model=3;0.0,0.0,0.0;0.0,0.0,0.0;0.06,0.06,0.595
Model=3;0.0,0.0,-0.1001;0.0,0.0,0.0;0.12,0.12,0.1
Link=1
Visibility=Visible
Axis=Visible;1.0,1.0,0.0,1.0;0.15
Model=12;0.0,0.0,-0.07;0.0,0.0,0.0;1,1,1
Link=2
Visibility=Visible
Axis=Visible;1.0,1.0,0.0,1.0;0.15
Model=12;0.0,0.0,-0.29;0.0,0.0,0.0;1,1,4.14
Link=3
Visibility=Visible
Axis=Visible;1.0,1.0,0.0,1.0;0.15
Model=3;0.0,0.0,-0.710;0.0,0.0,0.0;0.039,0.039,0.710
Link=4
Visibility=Visible
Axis=Visible;1.0,0.25,0.25,1.0;0.15
```

A3.3 MANIPULADOR ROBOTURB

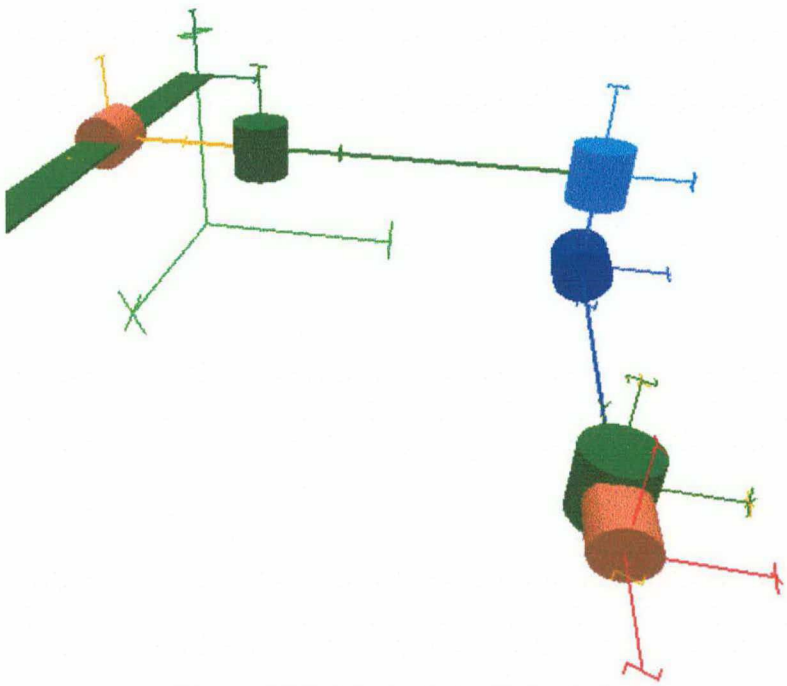


Figura A3.3. Manipulador RoboTurb

```
[CONFIG]
Name=RoboTurb
DOF=7
Flags=Euler|NoLimits|NoDynamics
Position=0.0,0.0,0.2
Orientation=-pi,-pid2,-pid2
[DESCRIPTION]
Manipulador redundante para operacoes de recuperacao de turbinas
Parametros extraidos do Relatorio do Estudo dos Parâmetros
Cinemáticos e Dinâmicos do Modelo Final do Manipulador
de Henrique Simas, Raul Guenther e Edson de Pieri (Set.2000)

[DH-DYNAMICS]
#VJ;a,alfa,d,teta,m,lcx,lcy,lcz,Ixx,Iyy,Izz,Ixy,IXz,Iyz
2;0.0 , 0.0 , 0.25 ,0.0
3;0.15,-pid2, 0.0  ,0.0
3;0.30, 0.0 , 0.0  ,0.0
3;0.0 , pid2,-0.087,0.0
3;0.0 ,-pid2, 0.3  ,0.0
3;0.0 , pid2, 0.0  ,0.0
3;0.0 , 0.0 , 0.057,0.0

[LIMITS]
#qmin,qmax,vmin,vmax,amin,amax,tmin,tmax
0,1.6,0,0,0,0,0,0
-pix2,pix2,0,0,0,0,0,0
-pix2,pix2,0,0,0,0,0,0
-pix2,pix2,0,0,0,0,0,0
-pix2,pix2,0,0,0,0,0,0
-pix2,pix2,0,0,0,0,0,0
-pix2,pix2,0,0,0,0,0,0
```

```
[MODELLIST]
File=Cilindro.m3d
File=Cubo.m3d
[LINKS]
Link=0
Visibility=Visible
Axis=Visible;0.0,0.6,0.0,1.0;0.075
Model=2;0.0,0.0,0.8;0.0,0.0,0.0;0.02,0.0015,0.8;0.3,0.5,0.0,1.0;0.3,0.5,0
.0,1.0;0.85,0.85,0.25,1.0;0,0,0,1.0;50.0
Link=1
Visibility=Visible
Axis=Visible;1.0,0.75,0.0,1.0;0.075
Model=3;0.0,0.0,-
0.025;0.0,0.0,0.0;0.025,0.025,0.05;0.7,0.5,0.4,1.0;0.7,0.5,0.4,1.0;0.85,0
.85,0.25,1.0;0,0,0,1.0;50.0
Link=2
Visibility=Visible
Axis=Visible;0.3,0.6,0.0,1.0;0.075
Model=3;0.0,0.0,-
0.025;0.0,0.0,0.0;0.025,0.025,0.05;0.3,0.5,0.0,1.0;0.3,0.5,0.0,1.0;0.85,0
.85,0.25,1.0;0,0,0,1.0;50.0
Link=3
Visibility=Visible
Axis=Visible;0.0,0.6,1.0,1.0;0.075
Model=3;0.0,0.0,-0.025;0.0,0.0,0.0;0.025,0.025,0.05
Link=4
Visibility=Visible
Axis=Visible;0.0,0.25,1.0,1.0;0.075
Model=3;0.0,0.0,-
0.025;0.0,0.0,0.0;0.025,0.025,0.05;0.0,0.0,0.75,1.0;0.0,0.0,0.75,1.0;0.85
,0.85,0.25,1.0;0,0,0,1.0;50.0
Link=5
Visibility=Visible
Axis=Visible;0.3,0.6,0.0,1.0;0.075
Model=3;0.0,0.0,-
0.025;0.0,0.0,0.0;0.025,0.025,0.05;0.3,0.5,0.0,1.0;0.3,0.5,0.0,1.0;0.85,0
.85,0.25,1.0;0,0,0,1.0;50.0
Link=6
Visibility=Visible
Axis=Visible;1.0,0.75,0.0,1.0;0.075
Model=3;0.0,0.0,0.02;0.0,0.0,0.0;0.02,0.02,0.04;0.7,0.5,0.4,1.0;0.7,0.5,0
.4,1.0;0.85,0.85,0.25,1.0;0,0,0,1.0;50.0
Link=7
Visibility=InVisible
Axis=Visible;1.0,0.0,0.0,1.0;0.075
```

A3.4 MANIPULADOR CILÍNDRICO

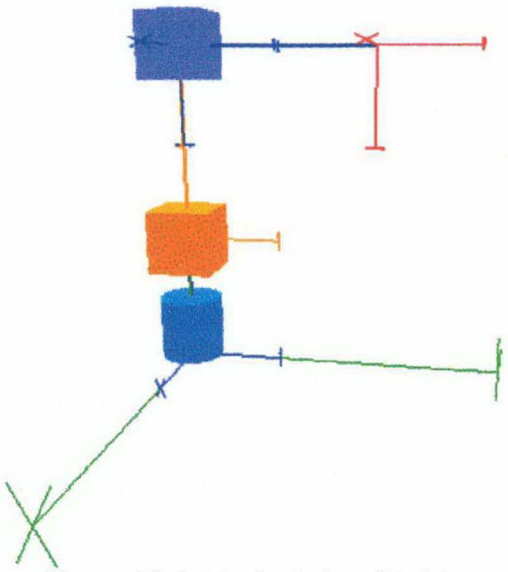


Figura A3.4. Manipulador cilíndrico

```
[CONFIG]
Name=Manipulador Cilindrico 3DOF
DOF=3
Flags=Euler|NoLimits|NoDynamics
Position=0.0,0.0,0.0
Orientation=0.0,0.0,0.0
[DESCRIPTION]
Manipulador cilindrico de 3 graus de liberdade
[DH-DYNAMICS]
#VJ;a,alfa,d,teta
3;0.0,  0.0,0.2 ,0.0
2;0.0,-pid2,0.3,0.0
2;0.0,  0.0,0.3,0.0
[MODELLIST]
File=cilindro.m3d
File=cubo.m3d
[LINKS]
Link=0
Visibility=Visible
Axis=Visible;0.0,0.0,1.0,1.0;0.15
Model=3;0.0,0.0,0.0;0.0,0.0,0.0;0.05,0.05,0.1
Link=1
Visibility=Visible
Axis=Visible;1.0,0.6,0.0,1.0;0.15
Model=2;0.0,0.0,0.0;0.0,0.0,0.0;0.05,0.05,0.05
Link=2
Visibility=Visible
Axis=Visible;0.0,0.0,1.0,1.0;0.15
Model=2;0.0,0.0,0.0;0.0,0.0,0.0;0.05,0.05,0.05
Link=3
Visibility=Invisible
Axis=Visible;1.0,0.0,0.0,1.0;0.15
```

A3.5 MANIPULADOR ESFÉRICO

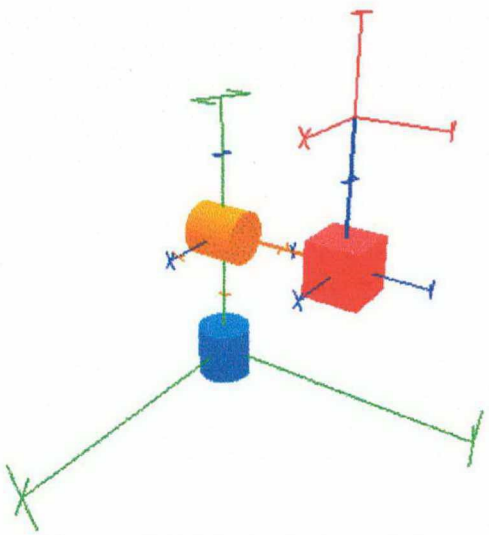


Figura A3.5. Manipulador esférico

```
[CONFIG]
Name=Manipulador Esferico 3DOF
DOF=3
Flags=Euler|NoLimits|NoDynamics
Position=0.0,0.0,0.25
Orientation=0.0,0.0,0.0
[DESCRIPTION]
Manipulador esférico com 3 graus de liberdade
Parâmetros cinemáticos apenas
[DH-DYNAMICS]
#VJ;a,alfa,d,teta
3;0.0,-pid2,0.0,0.0
3;0.0,pid2,0.25,0.0
2;0.0,0.0,0.25,0.0
[MODELLIST]
File=cilindro.m3d  # ID do modelo : 3
File=cubo.m3d      # ID do modelo : 2
[LINKS]
Link=0
Visibility=Visible
Axis=Visible;0.0,0.0,1.0,1.0;0.15
Model=3;0.0,0.0,-0.30;0.0,0.0,0.0;0.05,0.05,0.1
Link=1
Visibility=Visible
Axis=Visible;1.0,0.5,0.0,1.0;0.13
Model=3;0.0,0.0,-0.05;0.0,0.0,0.0;0.05,0.05,0.1;
#A linha abaixo é o material utilizado para visualizacao do modelo
#1.0,0.6,0.0,1.0;1.0,0.6,0.0,1.0;0.85,0.85,0.25,1.0;0,0,0,1.0;50.0
Link=2
Visibility=Visible
Axis=Visible;0.0,0.0,1.0,1.0;0.15
Model=2;0.0,0.0,0.0;0.0,0.0,0.0;0.05,0.05,0.05
Link=3
Visibility=InVisible
Axis=Visible;1.0,0.0,0.0,1.0;0.15
```

A3.6 MANIPULADOR ARTICULADO

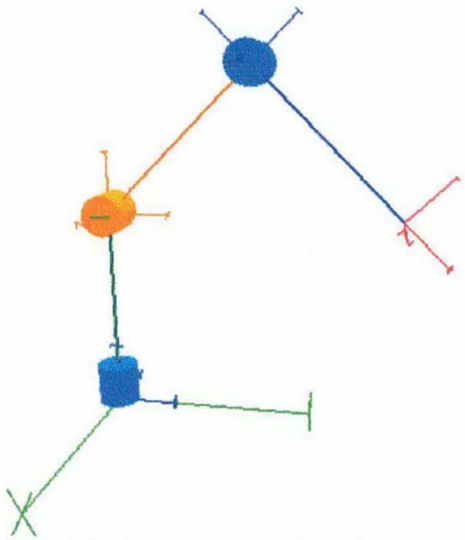


Figura A3.6. Manipulador articulado (antropomórfico)

```
[CONFIG]
Name=Manipulador Articulado 3DOF
DOF=3
Flags=Euler|NoLimits|NoDynamics
Position=0.0,0.0,0.0
Orientation=pid2,0.0,0.0
[DESCRIPTION]
Manipulador articulado (ou antropomórfico) com 3 graus de liberdade
Parâmetros cinemáticos apenas
[DH-DYNAMICS]
#VJ;a,alfa,d,teta
3;0.0,pid2,0.5,0.0
3;0.5,0.0,0.0,pid4
3;0.5,0.0,0.0,-pid2
[MODELLIST]
File=cilindro.m3d
[LINKS]
Link=0
Visibility=Visible
Axis=Visible;0.0,0.0,1.0,1.0;0.15
Model=3;0.0,0.0,0.0;0.0,0.0,0.0;0.05,0.05,0.1
Link=1
Visibility=Visible
Axis=Visible;1.0,0.5,0.0,1.0;0.15
Model=3;0.0,0.0,-0.05;0.0,0.0,0.0;0.05,0.05,0.1;
Link=2
Visibility=Visible
Axis=Visible;0.0,0.0,1.0,1.0;0.15
Model=3;0.0,0.0,-0.05;0.0,0.0,0.0;0.05,0.05,0.1
Link=3
Visibility=InVisible
Axis=Visible;1.0,0.0,0.0,1.0;0.15
```

A3.7 MANIPULADOR CARTESIANO

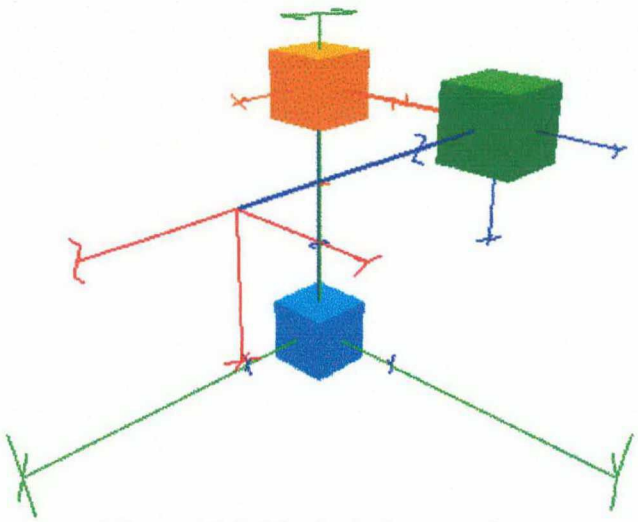


Figura A3.7. Manipulador cartesiano

```
[CONFIG]
Name=Manipulador Esferico 3DOF
DOF=3
Flags=Euler|NoLimits|NoDynamics
Position=0.0,0.0,0.0
Orientation=0.0,0.0,0.0
[DESCRIPTION]
Manipulador cartesiano com 3 graus de liberdade
Parâmetros cinemáticos apenas
[DH-DYNAMICS]
#VJ;a,alfa,d,teta
2;0.0,-pid2,0.4,0.0
2;0.0,pid2,0.3,pid2
2;0.0,0.0,0.4,0.0
[MODELLIST]
File=cubo.m3d
[LINKS]
Link=0
Visibility=Visible
Axis=Visible;0.0,0.0,1.0,1.0;0.15
Model=2;0.0,0.0,0.0;0.0,0.0,0.0;0.05,0.05,0.05;0.0,0.6,1.0,1.0;0.0,0.6,1.
0,1.0;0.85,0.85,0.25,1.0;0,0,0,1.0;50.0
Link=1
Visibility=Visible
Axis=Visible;0.9,0.4,0.0,1.0;0.15
Model=2;0.0,0.0,0.0;0.0,0.0,0.0;0.05,0.05,0.05;1.0,0.6,0.0,1.0;1.0,0.6,0.
0,1.0;0.85,0.85,0.25,1.0;0,0,0,1.0;50.0
Link=2
Visibility=Visible
Axis=Visible;0.0,0.0,1.0,1.0;0.15
Model=2;0.0,0.0,0.0;0.0,0.0,0.0;0.05,0.05,0.05;0,0.8,0.0,1.0;0,0.8,0.0,1.
0;0.85,0.85,0.25,1.0;0,0,0,1.0;50.0
Link=3
Visibility=InVisible
Axis=Visible;1.0,0.0,0.0,1.0;0.15
```

A3.8 CONTROLADOR PROPORCIONAL-DERIVATIVO

```
// Arquivo CtrlPD.cpp
#include "..\DllCtrl.h"

USELIB("E:\Borland\CBuilder\LIB\CrExtMth.LIB");

int WINAPI DllEntryPoint(HINSTANCE hinst, unsigned long reason, void*)
{return 1;}

extern "C" int __declspec(dllexport) CreateInstance (TDllController** P);

//---------------------------------------------------------------------
// Classe Controle PD
class TPDCtrl : public TDllController
{
protected:
  Matrix Kp, Kd;
  int    NV;

public:
  TPDCtrl (int n=0)     {NV = n; Kp.Eye(n); Kd.Eye(n);};
  TPDCtrl (TPDCtrl &C) {Kp = C.Kp; Kd = C.Kd; NV = C.NV;};
 ~TPDCtrl (void) {};

virtual int    Reset     (void) {};
virtual int    Calculate (Matrix &Qd, Matrix &Q, Matrix &U);
virtual int    Set       (double V, int Kind, int i=0, int j=0);
virtual double Get       (int Kind=ctDOF, int i=0, int j=0);

virtual int ParamNum  (void) {return 2;};
virtual int ParamSize (int P, int &L, int &C);
virtual int ParamName (int P, char *S);
virtual int ParamHelp (int P, char *S);
};

//---------------------------------------------------------------------
// Implementação

// Cria a instância do controlador
int CreateInstance (TDllController** P)
{if (((*P)=new TPDCtrl(0)) != NULL)   return 1;
 else                                 return 0;}

// Determina a ação de controle τ=K_p(q_d−q)−K_p q̇
int TPDCtrl::Calculate (Matrix &Qd, Matrix &Q, Matrix &U)
{U = Kp*(Qd(0,0,NV,1)-Q(0,0,NV,1)) - Kd*Q(NV,0,NV,1);
 return (U.Size(mtTotal)!=0);}

// Retorna informações do controlador
double TPDCtrl::Get (int Kind, int i, int j)
{switch(Kind)
   {case ctDOF : return NV;       // Numero de variaveis de estado (DOF)
    case 1     : return Kp(i,j); // Valor do elemento i,j da matriz Kp
    case 2     : return Kd(i,j); // Valor do elemento i,j da matriz Kd
    default    : return 0;    } }
```

```
// Define parâmetros do controlador
int TPDCtrl::Set (double V, int Kind, int i, int j)
{switch(Kind)
   {case ctDOF : if (V >= 0.0)          // Numero de graus de liberdade
                    {NV = (int)V;       // a controlar
                     Kp.Dim (NV,NV,mtCopy); Kd.Dim (NV,NV,mtCopy);}
                 else return 0;
                 break;
    case 1 :  Kp(i,j) = V;     // Valor de um dos elementos da matriz Kp
              break;
    case 2 :  Kd(i,j) = V;     // Valor de um dos elementos da matriz Kd
              break;
    default : return 0;
   }
return 1;}

// Retorna as dimensões das matrizes de ganhos em L e C
int TPDCtrl::ParamSize (int P, int &L, int &C)
{switch(P)
   {case 1  :
    case 2  : L = C = NV;       // Dimensões das matrizes de ganhos
              break;
    default : return 0;   }
 return 1;}

// Retorna os nomes das matrizes de ganhos em S
int TPDCtrl::ParamName (int P, char *S)
{switch(P)
   {case 1  : if (S != NULL) strcpy (S,"Kp");
              break;
    case 2  : if (S != NULL) strcpy (S,"Kd");
              break;
    default : return 0;   }
 return 1;}

// Retorna descrição das matrizes de ganhos em S
int TPDCtrl::ParamHelp (int P, char *S)
{switch(P)
   {case 1  : if (S != NULL) strcpy (S,"Matriz de ganhos da realimentação
do erro de posição");
              break;
    case 2   : if (S != NULL) strcpy (S,"Matriz de ganhos do termo de
realimentação de velocidade");
              break;
    default : return 0;   }
 return 1;}
```

A3.9 Controlador PD com Compensação da Gravidade Para o Manipulador Plano de Dois Graus de Liberdade

```
// Arquivo CtrlPDGrav.cpp
#include "..\DllCtrl.h"
#include <vcl\vcl.h>
#pragma hdrstop

USERES("CtrlPDGrav.res");
USELIB("E:\Borland\CBuilder\LIB\CrExtMth.LIB");

int WINAPI DllEntryPoint(HINSTANCE hinst, unsigned long reason, void*)
{return 1;}

extern "C" int __declspec(dllexport) CreateInstance (TDllController** P);

//---------------------------------------------------------------------
// Classe Controle PD com compensacao da gravidade para m2dof.rdf
class TPDGravCtrl : public TDllController
{
protected:
  Matrix Kp, Kd, G, m, a;
  int    NV;
  double g;

public:
  TPDGravCtrl (int n=0) {NV = 2; Kp.Eye(2); Kd.Eye(2); G.Dim(2,1);
                         m.Dim(2,1); m[0] = m[1] = 50.0;
                         a.Dim(2,1); a[0] = a[1] = 1.0; g = 9.81;};
  TPDGravCtrl (TPDGravCtrl &C) {Kp = C.Kp; Kd = C.Kd; NV = C.NV; G = C.G;
                                m = C.m; a = C.a; g = C.g;};
 ~TPDGravCtrl (void) {};

virtual int    Reset     (void) {};
virtual int    Calculate (Matrix &Qd, Matrix &Q, Matrix &U);
virtual int    Set       (double V, int Kind, int i=0, int j=0);
virtual double Get       (int Kind=ctDOF, int i=0, int j=0);

virtual int ParamNum  (void) {return 5;};
virtual int ParamSize (int P, int &L, int &C);
virtual int ParamName (int P, char *S);
virtual int ParamHelp (int P, char *S);
};


//---------------------------------------------------------------------
// Implementação

// Cria a instância do controlador
int CreateInstance (TDllController** P)
{if (((*P)=new TPDGravCtrl(0)) != NULL)  return 1;
 else                                    return 0;}
```

```
// Determina a ação de controle τ = K_p(q_d − q) − K_p q̇ + G
int TPDGravCtrl::Calculate (Matrix &Qd, Matrix &Q, Matrix &U)
{G[1] = m[1]*a[1]*cos(Q(0,0)+Q(1,0))*g;
 G[0] = (m[0]+m[1])*a[0]*cos(Q(0,0))*g + G[1];
 U = Kp*(Qd(0,0,NV,1)-Q(0,0,NV,1)) - Kd*Q(NV,0,NV,1) + G;
 return (U.Size(mtTotal)!=0);}

// Define parâmetros do controlador
int TPDGravCtrl::Set (double V, int Kind, int i, int j)
{switch(Kind)
   {case ctDOF : return 0;
    case 1 :  Kp(i,j) = V;      break;
    case 2 :  Kd(i,j) = V;      break;
    case 3 :  m(i,j) = V;       break;
    case 4 :  a(i,j) = V;       break;
    case 5 :  g = V;            break;
    default : return 0;
   }
return 1;}

// Retorna informações do controlador
double TPDGravCtrl::Get (int Kind, int i, int j)
{switch(Kind)
   {case ctDOF : return NV;
    case 1     : return Kp(i,j);
    case 2     : return Kd(i,j);
    case 3     : return m(i,j);
    case 4     : return a(i,j);
    case 5     : return g;
    default    : return 0;   } }

// Retorna as dimensões dos parametros em L e C
int TPDGravCtrl::ParamSize (int P, int &L, int &C)
{switch(P)
   {case 1  :
    case 2  : L = C = NV;          break;
    case 3  :
    case 4  : L = NV; C = 1;       break;
    case 5  : L = C = 1;           break;
    default : return 0;   }
 return 1;}

// Retorna os nomes dos parametros em S
int TPDGravCtrl::ParamName (int P, char *S)
{switch(P)
   {case 1  : if (S != NULL) strcpy (S,"Kp");
              break;
    case 2  : if (S != NULL) strcpy (S,"Kd");
              break;
    case 3  : if (S != NULL) strcpy (S,"m");
              break;
    case 4  : if (S != NULL) strcpy (S,"a");
              break;
    case 5  : if (S != NULL) strcpy (S,"g");
              break;
    default : return 0;   }
 return 1;}
```

```
// Retorna descrição dos parametros em S
int TPDGravCtrl::ParamHelp (int P, char *S)
{switch(P)
   {case 1  : if (S != NULL) strcpy (S,"Matriz de ganhos da realimentação
do erro de posição");
              break;
    case 2  : if (S != NULL) strcpy (S,"Matriz de ganhos do termo de
realimentação de velocidade");
              break;
    case 3  : if (S != NULL) strcpy (S,"Massas dos elos");
              break;
    case 4  : if (S != NULL) strcpy (S,"Comprimentos dos elos");
              break;
    case 5  : if (S != NULL) strcpy (S,"Aceleração da gravidade");
              break;
    default : return 0;   }
 return 1;}
```

A3.10 CONTROLADOR PD COM COMPENSAÇÃO DA GRAVIDADE PARA O MANIPULADOR SCARA

```
// Arquivo CtrlPDGravInter.cpp
#include "..\DllCtrl.h"
#include <vcl\vcl.h>
#pragma hdrstop

USERES("CtrlPDGravInter.res");
USELIB("E:\Borland\CBuilder\LIB\CrExtMth.LIB");

int WINAPI DllEntryPoint(HINSTANCE hinst, unsigned long reason, void*)
{return 1;}

extern "C" int __declspec(dllexport) CreateInstance (TDllController** P);


//---------------------------------------------------------------------------
// Classe Controle PD
class TPDGravCtrl : public TDllController
{
protected:
  Matrix Kp, Kd, G;
  int    NV;
  double g, m;

public:
  TPDGravCtrl (int n=0) {NV = 4; Kp.Eye(4); Kd.Eye(4); G.Fill(4,1);
                         m = 3.5; g = 9.81; G[2] = -m*g;};
  TPDGravCtrl (TPDGravCtrl &C) {Kp = C.Kp; Kd = C.Kd; NV = C.NV;
                                G = C.G; m = C.m; g = C.g;};
 ~TPDGravCtrl (void) {};

virtual int    Reset     (void) {};
virtual int    Calculate (Matrix &Qd, Matrix &Q, Matrix &U);
virtual int    Set       (double V, int Kind, int i=0, int j=0);
virtual double Get       (int Kind=ctDOF, int i=0, int j=0);

virtual int ParamNum  (void) {return 4;};
virtual int ParamSize (int P, int &L, int &C);
virtual int ParamName (int P, char *S);
virtual int ParamHelp (int P, char *S);
};


int CreateInstance (TDllController** P)
{if (((*P)=new TPDGravCtrl(0)) != NULL)  return 1;
 else                                    return 0;}

int TPDGravCtrl::Calculate (Matrix &Qd, Matrix &Q, Matrix &U)
{U = G + Kp*(Qd(0,0,NV,1)-Q(0,0,NV,1)) - Kd*Q(NV,0,NV,1);
 return (U.Size(mtTotal)!=0);}
```

```
int TPDGravCtrl::Set (double V, int Kind, int i, int j)
{switch(Kind)
   {case ctDOF : return 0;
    case 1 :  Kp(i,j) = V;              break;
    case 2 :  Kd(i,j) = V;              break;
    case 3 :  m = V; G[2] = -m*g;       break;
    case 4 :  g = V; G[2] = -m*g;       break;
    default : return 0;
   }
return 1;}

double TPDGravCtrl::Get (int Kind, int i, int j)
{switch(Kind)
   {case ctDOF : return NV;
    case 1     : return Kp(i,j);
    case 2     : return Kd(i,j);
    case 3     : return m;
    case 4     : return g;
    default    : return 0;   }
}

int TPDGravCtrl::ParamSize (int P, int &L, int &C)
{switch(P)
   {case 1  :
    case 2  : L = C = NV;     break;
    case 3  :
    case 4  : L = C = 1;      break;
    default : return 0;   }
 return 1;}

int TPDGravCtrl::ParamName (int P, char *S)
{switch(P)
   {case 1  : if (S != NULL) strcpy (S,"Kp");
                 break;
    case 2  : if (S != NULL) strcpy (S,"Kd");
                 break;
    case 3  : if (S != NULL) strcpy (S,"m");
                 break;
    case 4  : if (S != NULL) strcpy (S,"g");
                 break;
    default : return 0;   }
 return 1;}

int TPDGravCtrl::ParamHelp (int P, char *S)
{switch(P)
   {case 1  : if (S != NULL) strcpy (S,"Matriz de ganhos da realimentação
do erro de posição");
                 break;
    case 2  : if (S != NULL) strcpy (S,"Matriz de ganhos do termo de
realimentação de velocidade");
                 break;
    case 3  : if (S != NULL) strcpy (S,"Massas dos elos 3 e 4 somados");
                 break;
    case 4  : if (S != NULL) strcpy (S,"Aceleração da gravidade");
                 break;
    default : return 0;   }
 return 1;}
```

REFERÊNCIAS BIBLIOGRÁFICAS

[1] SPONG, M.; VIDYASAGAR, M. **Robot Dynamics and Control**. New York: John Wiley and Sons, 1989.

[2] SCIAVICCO, L.; SICILIANO, B. **Modeling and Control of robot Manipulators**. New York: McGraw-Hill, 1996.

[3] GROOVER, M. P.; WEISS, M.; NAGEL, R. N. et al. **Robótica - Tecnologia e Programação**. São Paulo: McGraw-Hill, 1989.

[4] GRAY, J. O. Recent Developments in Advanced Robotics & Intelligent Machines. In: **Advanced Robotics and Intelligent Machines**. Londres: The Institution of Electrical Engineers, 1996. p. 1-18.

[5] ASIMOV, I. **Visões de Robô**. Rio de Janeiro: Record, 1994.

[6] ASADA, H.; SLOTINE, J. J. E. **Robot Analysis and Control**. New York: John Wiley and Sons, 1986.

[7] DE WIT, C.C.; SICILIANO, B.; BASTIN, G. **Theory of Robot Control**. Londres: Springer-Verlag, 1996.

[8] SILVA, J. C. **Desenvolvimento de Um Sistema Para Análise e Modelamento Dinâmico de Robôs Industriais**. Florianópolis, 1990. Dissertação (Mestrado em Engenharia Mecânica) - Departamento de Engenharia Mecânica – Centro Tecnológico, Universidade Federal de Santa Catarina.

[9] ROCHA, C. R. **Um Simulador Com Animação Gráfica em Três Dimensões Para Robôs Manipuladores**. Rio Grande, 1996. Projeto de Graduação em Engenharia Mecânica. Fundação Universidade do Rio Grande.

[10] RUMBAUGH, J.; BLAHA, M.; PREMERLANI, W. et al. **Modelagem e Projetos Baseados Em Objetos**. Rio de Janeiro: Campus, 1997.

[11] LEWIS, F. L.; ABDALLAH, C. T.; DAWSON, D.M. **Control of Robot Manipulators**. New York: MacMillan, 1993.

[12] DENAVIT, J.; HARTENBERG, R. S. A Kinematic Notation for Lower-Pair Mechanisms Based on Matrices. **ASME Journal of Applied Mechanics**, [S.l.], v. 22, p. 215-221, 1955.

[13] HILL, F.S. **Computer Graphics**. New York: MacMillan, 1990.

[14] YOSHIKAWA, T. **Foundations of Robotics**. Cambridge: MIT Press, 1990.

[15] LUH, J.Y.S. An Anatomy of Industrial Robots and Their Controls. **IEEE Transactions on Automatic Control**, [S.l.], v.AC-28, n.2, p. 133-153, 1983.

[16] KRAUS Jr., W. **The Robotic Manipulation of Flexible Materials: A Hybrid Position/Force Approach**. Sidney, 1996. Tese (Ph.D em Engenharia Elétrica). Faculty of Engineering and Information Technology, The Australian National University.

[17] GOMES, S. C. P. Modelagem de Atritos Internos às Articulações de Robôs Manipuladores. In: **Congresso Brasileiro de Engenharia Mecânica** (1995: Belo Horizonte). Anais. Belo Horizonte, 1995.

[18] _____. **Métodos Numéricos: Teoria e Programação**. Rio Grande: Editora da FURG, 1999.

[19] CHAPRA, S. C.; CANALE, R. P. **Numerical Methods For Engineers**. 2. Ed. New York: McGraw-Hill, 1985.

[20] MILJANOVIC, D. M.; CROFT, E. A. A Taxonomy For Robot Control. In: **IEEE International Conference on Robotics and Automation** (Maio 1999 : Detroit). Proceedings. Detroit, 1999. p. 176-181.

[21] DANIEL, R. W.; FISCHER, P. J.; MCAREE, P. R. Force feedback control in robots and its application to decommissioning. In: **Advanced Robotics and Intelligent Machines**. Londres: The Institution of Electrical Engineers, 1996. p. 71-88.

[22] NYHOFF, L.; LEESTMA. S. **Data Structures and Program Design in Pascal**. New York: MacMillan, 1991.

[23] BOOCH, G.; RUMBAUGH, J.; JACOBSON, I. **UML - Guia do Usuário**. Rio de Janeiro: Campus, 2000.

[24] SHLAER, S.; MELLOR, S. J. **Análise de Sistemas Orientados a Objetos**. São Paulo: McGraw-Hill, 1991.

[25] **TechEncyclopedia**. http://www.techweb.com/encyclopedia, dez. 2000.

[26] WIENER, R. S.; PINSON, L.J. **C++: Programação Orientada Para Objeto - Manual Prático e Profissional**. São Paulo: Makron Books, 1991.

[27] COX, B. J. **Programação Orientada Para Objeto**. São Paulo: Makron Books, 1991.

[28] **Borland C++ Builder**. http://www.borland.com/bcppbuilder, dez. 2000.

[29] FOSNER, R. **OpenGL Programming for Windows 95 and Windows NT**. Reading: Addison-Wesley, 1997.

[30] **OpenGL - High Performance 2D/3D Graphics**. http://www.opengl.org, dez. 2000.

[31] BCBCWG.hlp. Borland C++Builder Component Writer's Guide.

[32] BCBPG.hlp. Borland C++Builder Programmer's Guide.

[33] GOLIN, J. GUENTHER, R.; WEIHMANN, L. **Manual do Usuário Robô INTER**. Florianópolis: Laboratório de Robótica - UFSC, 1998.

[34] RAMIREZ, A. R. G. **Controle de Posição de Robôs Manipuladores com Transmissões Flexíveis**. Florianópolis, 1998. Dissertação (Mestrado em Engenharia Elétrica) - Departamento de Automação e Sistemas – Centro Tecnológico, Universidade Federal de Santa Catarina.

[35] GORINEVSKY, D. M.; FORMALSKY, A. M.; SCHNEIDER, A. Y. **Force Control of Robotic Systems**. Boca Raton: CRC Press, 1997.

[36] GUENTHER, R.; DE PIERI, E. R. A Simulação de Robôs em Contato Com o Meio. In.: **I Workshop de Robótica Inteligente** (1997 : Brasília). Anais. Brasília: Ed. UnB, 1997.

[37] DE LUCA, A.; MANES, C. Modeling of Robots in Contact with a Dynamic Environment. **IEEE Transactions on Robotics and Automation**, [S.l.], v.10, n.4, p. 542-548, 1994.

[38] TENAGLIA, C. A.; ORIN, D. E.; LAFARGE, R.A.; LEWIS, C. Toward development of a Generalized Contact Algorithm for Polyhedral Objetcs. In.: **IEEE International Conference on Robotics and Automation** (Maio 1999 : Detroit). Proceedings. Detroit, 1999. p. 2887-2892.

[39] FEATHERSTONE, R.; THIEBAUT, S. S.; KHATIB, O. A General Contact Model for Dynamically-Decoupled Force/Motion Control. In.: **IEEE International Conference on Robotics and Automation** (Maio 1999 : Detroit). Proceedings. Detroit, 1999. p. 3281-3286.

[40] STAFFETTI, E.; ROS, L.; THOMAS, F. A Simple Characterization of The Infinitesimal Motions Separating General Polyhedra in Contact. In.: **IEEE International Conference on Robotics and Automation** (Maio 1999 : Detroit). Proceedings. Detroit, 1999. p. 571-577.