INSTITUTO TECNOLÓGICO DE AERONÁUTICA

Henrique Silva Simplicio

IMPLEMENTAÇÃO DO MÉTODO DE COLOCAÇÃO DIRETA PARA OTIMIZAÇÃO DE TRAJETÓRIA USANDO O SOFTWARE MATLAB

Trabalho de Graduação
2024

Curso de Engenharia Aeroespacial

CDU ¿¿¿.¿¿

Henrique Silva Simplicio

IMPLEMENTAÇÃO DO MÉTODO DE COLOCAÇÃO DIRETA PARA OTIMIZAÇÃO DE TRAJETÓRIA USANDO O SOFTWARE MATLAB

Orientador

Prof. Dr. Mauricio Andrés Varela Morales (ITA)

Coorientador

Prof. Dr. Flávio Luiz Cardoso Ribeiro (ITA)

ENGENHARIA AEROESPACIAL

São José dos Campos
Instituto Tecnológico de Aeronáutica

2024

Dados Internacionais de Catalogação-na-Publicação (CIP)
Divisão de Informação e Documentação

Silva Simplicio, Henrique
Implementação do método de colocação direta para otimização de trajetória usando o software MATLAB / Henrique Silva Simplicio.
São José dos Campos, 2024.
78f.

Trabalho de Graduação – Curso de Engenharia Aeroespacial– Instituto Tecnológico de Aeronáutica, 2024. Orientador: Prof. Dr. Mauricio Andrés Varela Morales. Coorientador: Prof. Dr. Flávio Luiz Cardoso Ribeiro.

I. Instituto Tecnológico de Aeronáutica. II. Título.

REFERÊNCIA BIBLIOGRÁFICA

SILVA SIMPLICIO, Henrique. **Implementação do método de colocação direta para otimização de trajetória usando o software MATLAB**. 2024. 78f. Trabalho de Conclusão de Curso (Graduação) – Instituto Tecnológico de Aeronáutica, São José dos Campos.

CESSÃO DE DIREITOS

NOME DO AUTOR: Henrique Silva Simplicio
TITULO DO TRABALHO: Implementação do método de colocação direta para otimização de trajetória usando o software MATLAB.
TIPO DO TRABALHO/ANO: Trabalho de Conclusão de Curso (Graduação) / 2024

Henrique Silva Simplicio
Rua H8B, 234
12.228-461 – São José dos Campos–SP

IMPLEMENTAÇÃO DO MÉTODO DE COLOCAÇÃO DIRETA PARA OTIMIZAÇÃO DE TRAJETÓRIA USANDO O SOFTWARE MATLAB

Essa publicação foi aceita como Relatório Final de Trabalho de Graduação

Henrique Silva Simplicio

Autor

Mauricio Andrés Varela Morales (ITA)

Orientador

Flávio Luiz Cardoso Ribeiro (ITA)

Coorientador

Profa. Dra. Maisa de Oliveira Terra

Coordenadora do Curso de Engenharia Aeroespacial

São José dos Campos, ¿¿ de novembro de 2024.

Àqueles que sempre me apoiaram e acreditaram em mim, principalmente minha mãe e minha parceira de vida.

Agradecimentos

Agradeço primeiramente ao meu orientador, Prof. Dr. Mauricio A. V. Morales, pela compreensão, paciência e valiosa orientação ao longo deste trabalho. Seus ensinamentos e direcionamentos foram fundamentais para o desenvolvimento deste projeto.

Ao Instituto Tecnológico de Aeronáutica (ITA), seus professores e funcionários, por proporcionarem um ambiente de excelência acadêmica e pela formação de qualidade que recebi durante minha graduação.

À minha família, especialmente minha mãe, pelo apoio incondicional, incentivo e compreensão durante toda minha trajetória acadêmica. Sua presença e suporte foram essenciais para que eu pudesse alcançar meus objetivos.

À minha parceira de vida, pelo companheirismo, paciência e apoio emocional durante os momentos mais desafiadores desta jornada. Sua presença tornou este caminho mais leve e gratificante.

Aos meus colegas de curso, pela amizade, troca de conhecimentos e momentos compartilhados ao longo destes anos. Em especial, agradeço aos amigos do 316 pela excelente convivência ao longo desta jornada.

Por fim, agradeço a todos que, direta ou indiretamente, contribuíram para a realização deste trabalho e para minha formação acadêmica.

*"Agir conforme aquilo que se fala, alinhar discurso e prática,
além de ser uma postura ética, é um sinal de autenticidade."*
— Mario Sergio Cortella

Resumo

Este trabalho apresenta o desenvolvimento de uma biblioteca em MATLAB para solução de problemas de otimização de trajetória utilizando o método de colocação direta trapezoidal. A biblioteca visa simplificar a resolução deste tipo de problema, permitindo que o usuário concentre-se apenas na modelagem, sem necessidade de implementar a lógica matemática da solução numérica. A implementação é validada através de três casos de teste: um problema de movimento unidimensional simples, o problema clássico da braquistócrona e um problema de otimização de trajetória de subida de uma aeronave eVTOL. Os resultados obtidos são comparados com soluções analíticas, quando disponíveis, e com resultados da literatura utilizando outros softwares de otimização. O trabalho contribui para a área de controle ótimo ao disponibilizar uma ferramenta que facilita a implementação de soluções numéricas para problemas de otimização de trajetória, sendo particularmente útil para aplicações em engenharia aeroespacial e áreas correlatas.

Abstract

This work presents the development of a MATLAB library for solving trajectory optimization problems using the trapezoidal direct collocation method. The library aims to simplify the resolution of such problems, allowing users to focus solely on modeling without the need to implement the mathematical logic of the numerical solution. The implementation is validated through three test cases: a simple one-dimensional motion problem, the classic brachistochrone problem, and an eVTOL aircraft climb trajectory optimization problem. The obtained results are compared with analytical solutions, when available, and with results from the literature using other optimization software, demonstrating the effectiveness of the developed library. The work contributes to the field of optimal control by providing a tool that facilitates the implementation of numerical solutions for trajectory optimization problems, being particularly useful for aerospace engineering applications and related fields.

Lista de Figuras

Lista de Códigos Fonte

Lista de Abreviaturas e Siglas

EDO	Equação Diferencial Ordinária
eVTOL	*electric Vertical Take-Off and Landing*
PCO	Problema de Controle Ótimo
PNL	Programação Não-Linear
POO	Programação Orientada a Objetos

Sumário

1 Introdução

1.1 Motivação

A otimização de trajetórias é um campo fundamental na engenharia de controle, com aplicações que vão desde o controle de veículos autônomos até a gestão de recursos em sistemas complexos. Embora existam diversos métodos analíticos para resolver problemas simples, a maioria dos problemas práticos requer soluções numéricas devido à sua complexidade.

Com o advento dos computadores digitais, tornou-se viável a solução de sistemas dinâmicos mais complexos e realistas através de métodos numéricos. Entre estes métodos, a colocação direta destaca-se por sua robustez e eficiência computacional. No entanto, a implementação destes métodos pode ser desafiadora, exigindo um profundo conhecimento tanto da teoria matemática subjacente quanto das técnicas de programação necessárias.

Atualmente, existem algumas ferramentas disponíveis para a solução de problemas de otimização de trajetória, como o PSOPT (BECERRA, 2022) e o OptimTraj (KELLY, 2022). Contudo, há ainda espaço para o desenvolvimento de soluções que combinem a simplicidade de uso com a flexibilidade necessária para abordar diferentes tipos de problemas. Em particular, uma biblioteca em MATLAB - ambiente amplamente utilizado na comunidade acadêmica e de engenharia - que implemente métodos de colocação direta de forma modular e extensível pode contribuir significativamente para a área.

Neste contexto, a motivação deste trabalho é desenvolver uma ferramenta que simplifique o processo de solução de problemas de otimização de trajetória, permitindo que usuários possam focar na modelagem do problema em si, sem a necessidade de implementar a complexa lógica matemática subjacente aos métodos numéricos de solução.

1.2 Objetivo

Este trabalho tem como objetivo a implementação de uma biblioteca no MATLAB, que permita a solução de problemas de otimização de trajetória utilizando um método de colocação direta, mais especificamente a colocação trapezoidal. Inspirando-se nos softwares PSOPT (BECERRA, 2022) e OptimTraj (KELLY, 2022), tal biblioteca deverá simplificar a resolução desse tipo de problema, de modo que o usuário deverá apenas modelizar o problema e entrar com os parâmetros necessários por meio de uma interface de dados. Assim, não haverá necessidade de implementar a lógica matemática por trás da solução numérica para obtenção dos resultados pertinentes.

1.3 Revisão Bibliográfica

Esta seção apresenta uma revisão histórica da otimização de trajetórias, com foco especial na modelagem do problema como um Problema de Controle Ótimo (PCO) e nas técnicas de solução, tanto analíticas quanto numéricas, que culminam na proposta de implementação de métodos de colocação direta usando o software MATLAB.

Iniciamos com as origens da otimização de trajetórias e a modelagem inicial do problema como um PCO. O estudo da otimização de trajetórias tem início com o problema da braquistócrona: encontrar a curva sob a qual uma partícula, sujeita a um campo gravitacional constante, sem atrito e com velocidade inicial nula, leva o menor tempo para se deslocar entre dois pontos (SUSSMANN; WILLEMS, 1997). Dentre as soluções propostas, a mais famosa usava o Cálculo Variacional.

Por sua vez, o Controle Ótimo tem sua origem ligada ao desenvolvimento do Cálculo Variacional, uma vez que o objetivo do primeiro é minimizar (ou maximizar) uma função objetivo, o que, por sua vez, é o tema estudado pelo segundo. Isaac Newton, Johann Bernoulli, Leonhard Euler e Ludovico Lagrange são alguns importantes nomes que inicialmente contribuíram para o desenvolvimento do Controle Ótimo (BECERRA, 2008). Como resultados importantes obtidos no desenvolvimento da teoria, podem-se citar a programação dinâmica (BELLMAN, 2010), o princípio mínimo de Pontryagin (PONTRYAGIN, 1987) e a formulação do regulador quadrático linear (*Linear-quadratic Regulator*, LQR) e do filtro de Kalman (KáLMáN, 1960a; KáLMáN, 1960b).

Existem diversas excelentes fontes que desenvolvem a teoria do Controle Ótimo, formulando diferentes tipos de problemas e suas soluções (BETTS, 2010; KIRK, 2004; BRYSON, 2018; ATHANS; FALB, 2007). Dentre os PCOs, há alguns casos específicos para os quais pode-se obter uma solução totalmente analítica. Como os mais conhecidos, têm-se os sistemas escalares lineares e o regulador quadrático linear, com possíveis variações nas condições de contorno (LEWIS *et al.*, 2012).

Com o advento do computador digital, tornou-se viável a solução de sistemas dinâmicos mais complexos e, consequentemente, de tratar problemas mais realistas. Os métodos numéricos desenvolvidos para solução de PCOs são, normalmente, classificados em duas categorias: os métodos indiretos e os métodos diretos (BETTS, 2010). Dentre aqueles classificados como indiretos, podem-se citar o *shooting* indireto e a colocação indireta. Por sua vez, dentre os classificados como diretos há maior variedade, como os métodos de colocação direta, o *shooting* direto, os métodos pseudoespectrais (BETTS, 1998) e a quasilinearização (PAINE, 1967).

Finalmente, nos concentramos na colocação direta como uma abordagem numérica para resolver PCOs. Nos métodos diretos, o problema de otimização de trajetória é discretizado, transformando-o em um problema de Programação Não-Linear (PNL). Os métodos de colocação direta não são diferentes. Neles, as funções contínuas que definem o problema são discretizadas usando vários segmentos polinomiais, construindo a função conhecida como *spline*. Dentre os métodos de discretização que se classificam como colocação direta, dois têm maior notoriedade: a colocação trapezoidal e a colocação de Hermite-Simpson (KELLY, 2017; BETTS, 2010).

Quanto a rotinas que implementam soluções numéricas e problemas exemplos disponibilizados, (LEWIS *et al.*, 2012) traz diversas rotinas em MATLAB, tanto para simulação de problemas usando resultados obtidos, quanto para implementação de métodos de solução numérica. (BECERRA, 2022) implementa o PSOPT, um software completo escrito em C++, apresentando uma interface de dados que permite a construção facilitada de problemas distintos e disponibilizando diferentes métodos de solução, como o pseudoespectral e as colocações trapezoidal e de Hermit-Simpson. (KELLY, 2022) é também um software completo escrito em MATLAB, que apresenta exemplos variados e permite a escolha dentre diversos métodos de solução. Além disso, em (KELLY, 2017), o autor objetiva ensinar sobre a implementação de métodos de colocação direta para solução de problemas de otimização de trajetória, de modo que apresenta explicações claras e focadas no assunto, além de vários exemplos de problemas e rotinas em MATLAB.

2 Controle Ótimo

O controle ótimo é uma disciplina fundamental na teoria de controle, com amplas aplicações em diversas áreas da engenharia e ciências aplicadas. Esta teoria lida com o problema de determinar a trajetória de controle que otimiza uma determinada medida de desempenho, sujeita às dinâmicas do sistema e às restrições impostas. A formulação geral do controle ótimo fornece uma base teórica robusta para resolver uma ampla gama de problemas de otimização, desde o controle de veículos autônomos até a gestão de recursos em sistemas complexos.

Neste capítulo, será abordada, inicialmente, a formulação geral do controle ótimo, destacando os princípios e as equações que fundamentam a teoria. Em seguida, explora-se a formulação por colocação direta, uma técnica numérica eficiente para resolver problemas de controle ótimo, com foco especial no método de colocação trapezoidal. Esta abordagem não apenas simplifica a implementação computacional, mas também melhora a precisão e a estabilidade das soluções, tornando-a uma ferramenta valiosa para pesquisadores e profissionais na área de otimização de trajetórias.

2.1 Formulação Geral

Segundo (BETTS, 2010), o Problema de Controle Ótimo pode ser formulado como um conjunto de N fases. Para a fase i, o sistema dinâmico pode ser descrito como um conjunto de variáveis dinâmicas

$$\mathbf{z} = \begin{bmatrix} \mathbf{x}^{(i)}(t) \\ \mathbf{u}^{(i)}(t) \end{bmatrix},$$

composto por $n_x^{(i)}$ variáveis de estado e por $n_u^{(i)}$ variáveis de controle, respectivamente. Além disso, o sistema pode conter $n_p^{(i)}$ parâmetros $\mathbf{p}^{(i)}$, os quais são independentes de t.

Normalmente, as dinâmicas do sistema são definidas por um conjunto de EDOs, chamadas de *equações de estado*:

$$\dot{\mathbf{x}}^{(i)} = \mathbf{f}^{(i)}\left[\mathbf{x}^{(i)}(t), \mathbf{u}^{(i)}(t), \mathbf{p}^{(i)}, t\right].$$

A formulação geral do Problema de Controle Ótimo com N fases utilizada neste trabalho é descrita em (BECERRA, 2022) e apresentada a seguir.

Problema $\mathcal{P}_1$

Encontrar as trajetórias de controle, $\mathbf{u}^{(i)}(t)$, $t \in \left[t_0^{(i)}, t_f^{(i)}\right]$, trajetórias de estado $\mathbf{x}^{(i)}(t)$, $t \in \left[t_0^{(i)}, t_f^{(i)}\right]$, parâmetros estáticos $\mathbf{p}^{(i)}$ e instantes $t_0^{(i)}, t_f^{(i)}$, $i = 1, \cdots, N$, que minimizem o índice de desempenho J, definido pela equação 2.1.

$$J = \sum_{i=1}^{N} \left[\overbrace{\varphi^{(i)}\left[\mathbf{x}^{(i)}\left(t_0^{(i)}\right), \mathbf{x}^{(i)}\left(t_f^{(i)}\right), \mathbf{p}^{(i)}, t_0^{(i)}, t_f^{(i)}\right]}^{\text{termo de Mayer}} + \underbrace{\int_{t_0^{(i)}}^{t_f^{(i)}} L^{(i)}\left[\mathbf{x}^{(i)}(t), \mathbf{u}^{(i)}(t), \mathbf{p}^{(i)}, t\right] dt}_{\text{termo de Lagrange}} \right] \tag{2.1}$$

onde $\varphi(\cdot)$ representa a função objetivo de fronteira e $L(\cdot)$ o integrando da integral de caminho ao longo da trajetória (KELLY, 2017).

O problema está sujeito às restrições diferenciais

$$\dot{\mathbf{x}}^{(i)}(t) = \mathbf{f}^{(i)}\left[\mathbf{x}^{(i)}(t), \mathbf{u}^{(i)}(t), \mathbf{p}^{(i)}, t\right], \; t \in \left[t_0^{(i)}, t_f^{(i)}\right], \tag{2.2}$$

às restrições de trajetória

$$\mathbf{g}_L^{(i)} \leq \mathbf{g}^{(i)}\left[\mathbf{x}^{(i)}(t), \mathbf{u}^{(i)}(t), \mathbf{p}^{(i)}, t\right] \leq \mathbf{g}_U^{(i)}, \; t \in \left[t_0^{(i)}, t_f^{(i)}\right], \tag{2.3}$$

às restrições de evento

$$\mathbf{e}_L^{(i)} \leq \mathbf{e}^{(i)}\left[\mathbf{x}^{(i)}\left(t_0^{(i)}\right), \mathbf{u}^{(i)}\left(t_0^{(i)}\right), \mathbf{x}^{(i)}\left(t_f^{(i)}\right), \mathbf{u}^{(i)}\left(t_f^{(i)}\right), \mathbf{p}^{(i)}, t_0^{(i)}, t_f^{(i)}\right] \leq \mathbf{e}_U^{(i)}, \tag{2.4}$$

às restrições de ligação de fase

$$
\begin{aligned}
\boldsymbol{\psi}_L \leq \boldsymbol{\psi}[&\mathbf{x}^{(1)}\left(t_0^{(1)}\right), \mathbf{u}^{(1)}\left(t_0^{(1)}\right), \\
&\mathbf{x}^{(1)}\left(t_f^{(1)}\right), \mathbf{u}^{(1)}\left(t_f^{(1)}\right), \mathbf{p}^{(1)}, t_0^{(1)}, t_f^{(1)}, \\
&\mathbf{x}^{(2)}\left(t_0^{(2)}\right), \mathbf{u}^{(2)}\left(t_0^{(2)}\right), \\
&\mathbf{x}^{(2)}\left(t_f^{(2)}\right), \mathbf{u}^{(2)}\left(t_f^{(2)}\right), \mathbf{p}^{(2)}, t_0^{(2)}, t_f^{(2)}, \\
&\vdots \\
&\mathbf{x}^{(N)}\left(t_0^{(N)}\right), \mathbf{u}^{(N)}\left(t_0^{(N)}\right), \\
&\mathbf{x}^{(N)}\left(t_f^{(N)}\right), \mathbf{u}^{(N)}\left(t_f^{(N)}\right), \mathbf{p}^{(N)}, t_0^{(N)}, t_f^{(N)}] \leq \boldsymbol{\psi}_U,
\end{aligned}
\tag{2.5}
$$

às restrições de limite

$$
\begin{aligned}
\mathbf{u}_L^{(i)} \leq \mathbf{u}^{(i)}(t) \leq \mathbf{u}_U^{(i)},\ t \in \left[t_0^{(i)}, t_f^{(i)}\right], \\
\mathbf{x}_L^{(i)} \leq \mathbf{x}^{(i)}(t) \leq \mathbf{x}_U^{(i)},\ t \in \left[t_0^{(i)}, t_f^{(i)}\right], \\
\mathbf{p}_L^{(i)} \leq \mathbf{p}^{(i)} \leq \mathbf{p}_U^{(i)}, \\
t_{0_L}^{(i)} \leq t_0^{(i)} \leq t_{0_U}^{(i)}, \\
t_{f_L}^{(i)} \leq t_f^{(i)} \leq t_{f_U}^{(i)},
\end{aligned}
\tag{2.6}
$$

e às seguintes restrições

$$
\begin{aligned}
& t_f^{(i)} - t_0^{(i)} \geq 0,\ i = 1, \cdots, N \\
& \mathbf{u}^{(i)} : \left[t_0^{(i)}, t_f^{(i)}\right] \to \mathbb{R}^{n_u^{(i)}} \\
& \mathbf{x}^{(i)} : \left[t_0^{(i)}, t_f^{(i)}\right] \to \mathbb{R}^{n_x^{(i)}} \\
& \mathbf{p}^{(i)} \in \mathbb{R}^{n_p^{(i)}} \\
& \varphi^{(i)} : \mathbb{R}^{n_x^{(i)}} \times \mathbb{R}^{n_x^{(i)}} \times \mathbb{R}^{n_p^{(i)}} \times \mathbb{R} \times \mathbb{R} \to \mathbb{R} \\
& L^{(i)} : \mathbb{R}^{n_x^{(i)}} \times \mathbb{R}^{n_u^{(i)}} \times \mathbb{R}^{n_p^{(i)}} \times \left[t_0^{(i)}, t_f^{(i)}\right] \to \mathbb{R} \\
& \mathbf{f}^{(i)} : \mathbb{R}^{n_x^{(i)}} \times \mathbb{R}^{n_u^{(i)}} \times \mathbb{R}^{n_p^{(i)}} \times \left[t_0^{(i)}, t_f^{(i)}\right] \to \mathbb{R}^{n_x^{(i)}} \\
& \mathbf{g}^{(i)} : \mathbb{R}^{n_x^{(i)}} \times \mathbb{R}^{n_u^{(i)}} \times \mathbb{R}^{n_p^{(i)}} \times \left[t_0^{(i)}, t_f^{(i)}\right] \to \mathbb{R}^{n_g^{(i)}} \\
& \mathbf{e}^{(i)} : \mathbb{R}^{n_x^{(i)}} \times \mathbb{R}^{n_u^{(i)}} \times \mathbb{R}^{n_x^{(i)}} \times \mathbb{R}^{n_u^{(i)}} \times \mathbb{R}^{n_p^{(i)}} \times \mathbb{R} \times \mathbb{R} \to \mathbb{R}^{n_e^{(i)}} \\
& \boldsymbol{\psi} : U_\Psi \to \mathbb{R}^{n_\Psi}
\end{aligned}
\tag{2.7}
$$

onde U_Ψ é o domínio da função $\boldsymbol{\psi}$.

2.2 Formulação por Colocação Direta

Os métodos diretos discretizam o PCO, de forma a converte-lo em um problema de PNL. Tal processo é conhecido como transcrição. Para isso, as funções contínuas do problema serão aproximadas por *splines*, as quais são funções compostas por vários segmentos polinomiais (KELLY, 2017).

Antes de apresentar a discretização das funções usando a colocação trapezoidal, faz-se necessário apresentar o problema de PNL. Tais problemas contêm um ou mais termos não-lineares, seja na sua função objetivo ou nas suas restrições. A típica formulação de um problema de PNL é apresentado na equação 2.8.

$$
\begin{aligned}
\min_{\mathbf{z}} J(\mathbf{z}) \qquad & \text{sujeito a} \\
& \mathbf{f}(\mathbf{z}) = \mathbf{0}, \\
& \mathbf{g}(\mathbf{z}) \leq \mathbf{0}, \\
& \mathbf{z}_L \leq \mathbf{z} \leq \mathbf{z}_U
\end{aligned}
\tag{2.8}
$$

Por simplicidade e clareza, dispensaremos a notação (i), indicativa da fase, na sequência deste documento. Iniciando a discretização das funções do PCO, seguindo o exposto em (BETTS, 2010), a duração da fase é dividida em n_s segmentos

$$t_0 = t_1 < t_2 < \cdots < t_M = t_f, \tag{2.9}$$

onde cada ponto é chamado de *nó*. O número de nós é dado por $M \equiv n_s + 1$. Quanto às variáveis do problema, a notação utilizada será $\mathbf{x}_k \equiv \mathbf{x}(t_k)$ para as variáveis de estado e $\mathbf{u}_k \equiv \mathbf{u}(t_k)$ para as variáveis de controle. Além disso, denota-se $\mathbf{f}_k \equiv \mathbf{f}\left[\mathbf{x}(t_k), \mathbf{u}(t_k), \mathbf{p}, t_k\right]$.

Dessa forma, visando a transcrição para um problema de PNL, as equações diferenciais do PCO são substituídas por um conjunto de restrições. Assim, as restrições do PCO - equações 2.2 a 2.4 - serão substituídas pelas restrições do problema de PNL:

$$\mathbf{c}_L \leq \mathbf{c}(\mathbf{x}) \leq \mathbf{c}_U, \tag{2.10}$$

onde

$$\mathbf{c}_L = \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \\ \vdots \\ \mathbf{0} \\ \mathbf{g}_L \\ \mathbf{g}_L \\ \vdots \\ \mathbf{g}_L \\ \mathbf{e}_L \\ \mathbf{e}_L \\ \boldsymbol{\psi}_L \\ \boldsymbol{\psi}_L \end{bmatrix} \qquad \mathbf{c}(\mathbf{x}) = \begin{bmatrix} \boldsymbol{\zeta}_1 \\ \boldsymbol{\zeta}_2 \\ \vdots \\ \boldsymbol{\zeta}_{M-1} \\ \mathbf{g}_1 \\ \mathbf{g}_2 \\ \vdots \\ \mathbf{g}_M \\ \mathbf{e}_0 \\ \mathbf{e}_f \\ \boldsymbol{\psi}_0 \\ \boldsymbol{\psi}_f \end{bmatrix} \qquad \mathbf{c}_U = \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \\ \vdots \\ \mathbf{0} \\ \mathbf{g}_U \\ \mathbf{g}_U \\ \vdots \\ \mathbf{g}_U \\ \mathbf{e}_U \\ \mathbf{e}_U \\ \boldsymbol{\psi}_U \\ \boldsymbol{\psi}_U \end{bmatrix} \tag{2.11}$$

As primeiras $n_x n_s$ restrições exigem que os vetores $\boldsymbol{\zeta}$ sejam zero, satisfazendo aproximadamente as equações diferenciais do PCO - dadas pela equação 2.2.

2.2.1 Colocação Trapezoidal

Focando no método trapezoidal, a variável de controle é discretizada usando uma *spline* linear

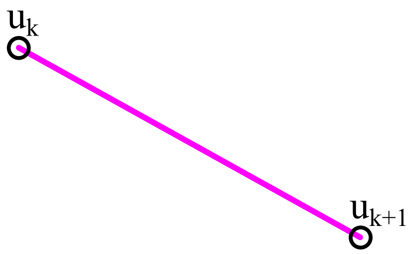
$$\mathbf{u}(t) \approx \mathbf{u}_k + \frac{\tau}{h_k}(\mathbf{u}_{k+1} - \mathbf{u}_k), \tag{2.12}$$

onde $\tau \equiv t - t_k$ e $h_k \equiv t_{k+1} - t_k$, e a variável de estado é discretizada usando uma *spline* quadrática

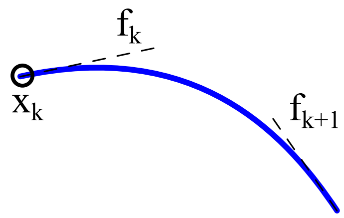
$$\mathbf{f}(t) = \dot{\mathbf{x}}(t) \approx \mathbf{f}_k + \frac{\tau}{h_k}(\mathbf{f}_{k+1} - \mathbf{f}_k), \tag{2.13}$$

que, ao ser integrada, resulta em

$$\mathbf{x}(t) \approx \mathbf{x}_k + \mathbf{f}_k\tau + \frac{\tau^2}{2h_k}(\mathbf{f}_{k+1} - \mathbf{f}_k). \tag{2.14}$$



(a) *Spline* linear de controle



(b) *Spline* quadrática de estado

FIGURA 2.1 – Representações das *splines* de controle e de estado

Desse modo, as variáveis são definidas como

$$\mathbf{z}^\intercal = (\mathbf{x}_1, \mathbf{u}_1, \cdots, \mathbf{x}_M, \mathbf{u}_M), \tag{2.15}$$

e as restrições

$$\boldsymbol{\zeta}_k = \mathbf{x}_{k+1} - \mathbf{x}_k - \frac{h_k}{2}(\mathbf{f}_{k+1} + \mathbf{f}_k), \tag{2.16}$$

Assim, pode-se reescrever o índice de desempenho - equação 2.1 - como

$$J = \varphi\left[\mathbf{x}(t_0), \mathbf{x}(t_f), \mathbf{p}, t_0, t_f\right] + \sum_{k=0}^{M-1} \frac{1}{2} h_k (L_{k+1} + L_k), \tag{2.17}$$

3 Metodologia

Relembrando, o objetivo deste trabalho é criar uma biblioteca no MATLAB que permita a resolução de problemas de otimização de trajetória de forma simplificada, de forma que o usuário precise apenas modelar o problema, sem a necessidade de implementar a lógica da solução numérica. Para isso, é necessário que a biblioteca forneça uma interface de dados, de modo a permitir que o usuário entre com a modelagem do seu problema. Além disso, deve-se implementar a colocação trapezoidal como método de solução.

Para validação da implementação a ser realizada, foram implementadas as soluções para alguns problemas exemplos. Tais problemas incluem o estudo de otimização de trajetória de um movimento simples em uma dimensão e braquistócrona, que são problemas clássicos na literatura de otimização de trajetórias, e o estudo de otimização de trajetória de subida de eVTOL em (COSTA, 2023), o qual utiliza o software PSOPT citado na Seção 1.3.

3.1 Interface de Dados

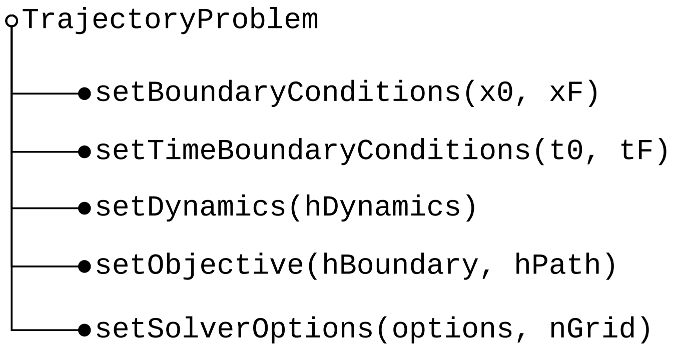
Diferente dos softwares OptimTraj (KELLY, 2022) e PSOPT (BECERRA, 2022), a implementação será realizada utilizando uma abordagem de POO, com uma classe MATLAB principal que conterá métodos para a configuração do problema e cálculo da solução. Além da classe principal, a biblioteca contará com funções auxiliares para cálculo da colocação trapezoidal e manipulação de dados.

Os arquivos da biblioteca, assim como os exemplos de uso, estão disponíveis no repositório (SIMPLICIO, 2024), e serão descritos na sequência. Para maior clareza, inicialmente será apresentada a classe principal, de modo que será possível entender a estrutura de dados e a lógica de funcionamento da biblioteca. Em seguida, serão apresentadas as funções auxiliares.

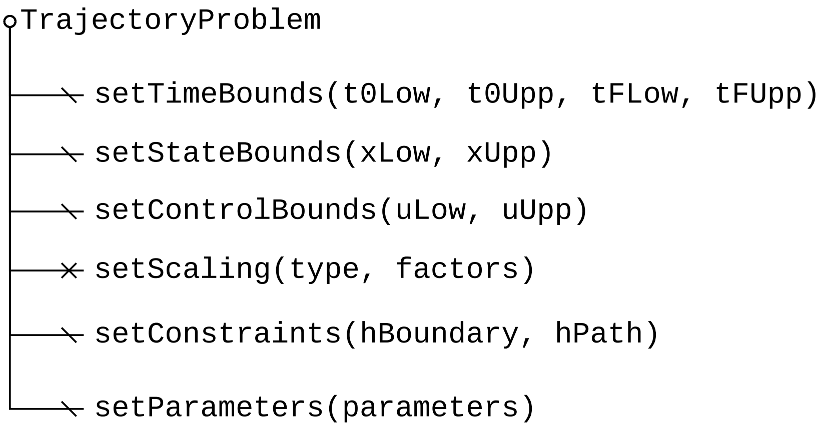
3.1.1 `TrajectoryProblem.m`

A classe `TrajectoryProblem` é a classe principal da biblioteca, e contém métodos para a configuração do problema e cálculo da solução. A classe conta com um construtor, que inicializa os parâmetros e variáveis necessárias para a resolução do problema com valores padrão, e métodos para a configuração dos limites e objetivo do problema, além de um método para a solução numérica do problema.

Dentre os métodos disponíveis, alguns são obrigatórios, ou seja, são necessários para a resolução de qualquer problema, e outros são opcionais, podendo ser utilizados ou não dependendo da necessidade do problema em questão, como pode ser visto na Figura 3.1.



(a) Métodos obrigatórios



(b) Métodos opcionais

FIGURA 3.1 – Métodos da classe `TrajectoryProblem`

Além disso, há também métodos auxiliares para a verificação e validação do problema, assim como métodos privados, utilizados internamente pela classe, como pode ser visto na Figura 3.2.

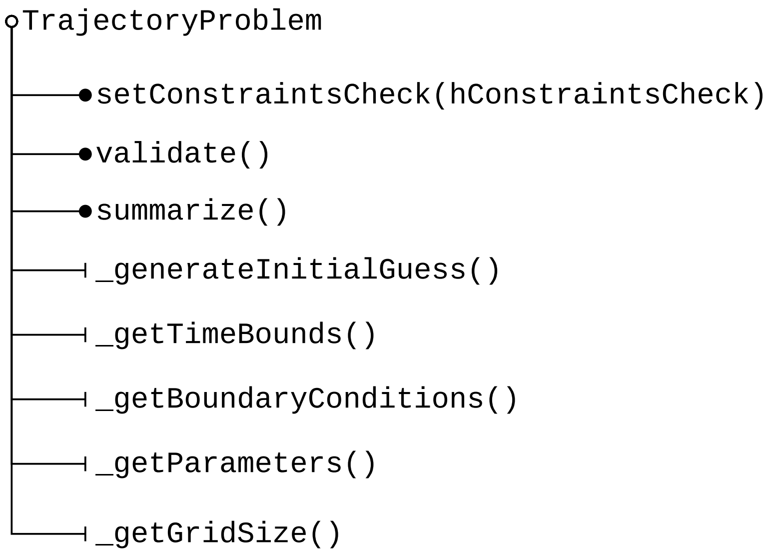


FIGURA 3.2 – Métodos auxiliares da classe `TrajectoryProblem`

Por fim, há o método para a solução do problema com a colocação trapezoidal, apresentado na Figura 3.3, o qual retorna um `array` de `structures`, com a solução do problema obtida em cada iteração, além de informações adicionais relacionadas a elas, como apresentado na Figura 3.4.

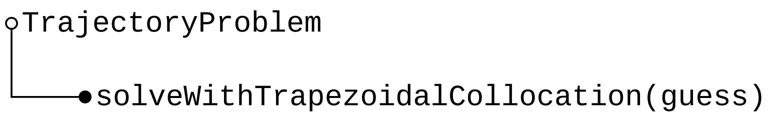


FIGURA 3.3 – Método para a solução do `TrajectoryProblem` com a colocação trapezoidal

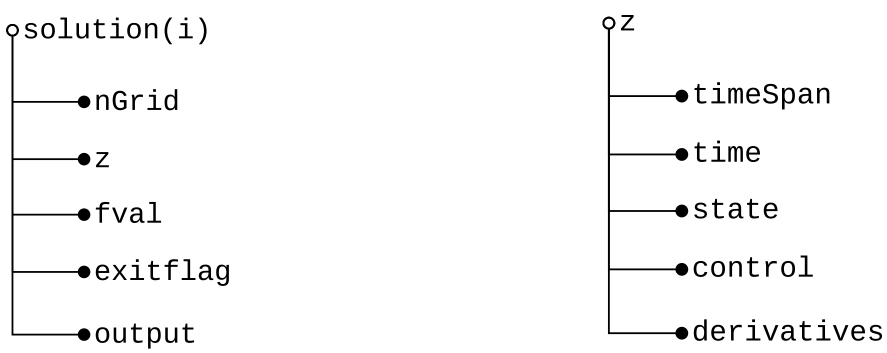


FIGURA 3.4 – Formato da `structure` `solution`

Na sequência, serão apresentados os métodos da classe, descrevendo sua funcionalidade e uso. O código fonte pode ser encontrado no anexo A.1.1.

3.1.1.1 `.setBoundaryConditions(x0, xF)`

Este método recebe como argumentos as condições de contorno do estado do problema, passadas como vetores coluna, os valida e os atribui às propriedades `x0` e `xF` da classe.

3.1.1.2 `.setTimeBoundaryConditions(t0, tF)`

Este método recebe como argumentos os limites de tempo do problema, passados como escalares, os valida e os atribui às propriedades `t0` e `tF` da classe.

3.1.1.3 `.setTimeBounds(t0Low, t0Upp, tFLow, tFUpp)`

Este método recebe como argumentos os limites de tempo do problema, passados como escalares, os valida e os atribui às propriedades `t0Low`, `t0Upp`, `tFLow` e `tFUpp` da classe.

3.1.1.4 `.setStateBounds(xLow, xUpp)`

Este método recebe como argumentos os limites de estado do problema, passados como vetores coluna, os valida e os atribui às propriedades `xLow` e `xUpp` da classe.

3.1.1.5 `.setControlBounds(uLow, uUpp)`

Este método recebe como argumentos os limites de controle do problema, passados como vetores coluna, os valida e os atribui às propriedades `uLow` e `uUpp` da classe.

3.1.1.6 `.setScaling(type, factors)`

Este método recebe como argumentos o tipo de variável a ser escalonada e os fatores de escalonamento, passados como vetores coluna, os valida e os atribui às propriedades `stateScaling`, `controlScaling` e `timeScaling` da classe.

3.1.1.7 `.setDynamics(hDynamics)`

Este método recebe como argumento uma função `handle` que representa a dinâmica do sistema, a qual deve ser definida pelo usuário, a valida e a atribui à propriedade `dynamics` da classe.

3.1.1.8 `.setObjective(hBoundaryObjective, hPathObjective)`

Este método recebe como argumento duas funções `handle` que representam os dois termos da função objetivo do problema, o termo de contorno (Mayer) e o termo de caminho (Lagrange) conforme explicado na Seção 2.1. Tais funções devem ser definidas pelo usuário e serão validadas e atribuídas à propriedade `objective` da classe, por meio da função `evaluateObjective`, descrita na Seção 3.1.6.

3.1.1.9 `.setConstraints(hBoundaryConstraints, hPathConstraints)`

Este método recebe como argumento duas funções `handle` que representam as restrições do problema. Tais funções devem ser definidas pelo usuário e serão validadas e atribuídas à propriedade `constraints` da classe, por meio da função `evaluateConstraints`, descrita na Seção 3.1.5.

3.1.1.10 `.setParameters(parameters)`

Este método recebe como argumento um vetor coluna de parâmetros do problema e os atribui à propriedade `parameters` da classe.

3.1.1.11 `.setVariableNames(stateNames, controlNames)`

Este método recebe como argumento os nomes das variáveis de estado e de controle do problema, passados como vetores de células, e os atribui às propriedades `stateNames` e `controlNames` da classe.

3.1.1.12 `.setSolverOptions(options, nGrid)`

Este método recebe como argumento as opções de solução do problema, passadas como uma `optimoptions('fmincon')` ou uma célula de tais opções, e o número de pontos de colocação da malha trapezoidal, e os atribui às propriedades `solverOptions` e `nGrid` da classe. Caso o argumento seja uma célula, o número de elementos da célula deve ser igual ao número de iterações que serão realizadas. Caso o argumento seja uma única opção de solução, ela será replicada para todas as iterações.

3.1.1.13 `.setConstraintsCheck(hConstraintsCheck)`

Este método recebe como argumento uma função `handle` que representa a função de verificação de restrições do problema, a qual deve ser definida pelo usuário, a valida e a atribui à propriedade `constraintsCheck` da classe.

3.1.1.14 `.validate()`

Este método verifica se todas as propriedades da classe foram definidas, retornando `true` caso todas as propriedades tenham sido definidas e `false` caso contrário. Além disso, caso alguma propriedade não tenha sido definida, é exibida uma mensagem de aviso detalhando quais propriedades estão faltando.

3.1.1.15 `.summarize()`

Este método retorna um `struct` com um resumo das propriedades da classe.

3.1.1.16 `.generateInitialGuess()`

Este método gera uma estimativa inicial para o problema, retornando um `struct` com os valores das variáveis de estado e de controle no instante inicial e final, além de uma malha de tempo. Os estados são interpolados linearmente entre os pontos inicial e final, enquanto os controles são considerados constantes e nulos.

3.1.1.17 `.solveWithTrapezoidalCollocation(guess)`

Este método resolve o problema com a colocação trapezoidal, recebendo como argumento uma estimativa inicial para o problema, e retornando um `array` de `structures`, com a solução do problema obtida em cada iteração, além de informações adicionais relacionadas a elas, como apresentado na Figura 3.4. Caso o argumento seja omitido, a estimativa inicial é gerada pelo método `.generateInitialGuess()`.

A partir da primeira iteração, a estimativa inicial é obtida por interpolação dos resultados da iteração anterior, utilizando a função `spline2` descrita na Seção 3.1.7.

Na chamada do solver `fmincon`, é interessante notar que, além da função objetivo, da estimativa inicial e das restrições não-lineares, são passadas as restrições inferiores e superiores, que utilizam os limites inferiores e superiores, respectivamente, do tempo inicial e final, dos estados e controles, multiplicados pelos fatores de escalonamento.

3.1.1.18 `.getTimeBounds()`

Este método retorna os limites de tempo do problema.

3.1.1.19 `.getBoundaryConditions()`

Este método retorna as condições de contorno do problema.

3.1.1.20 `.getParameters()`

Este método retorna os parâmetros do problema.

3.1.1.21 `.getGridSize()`

Este método retorna o número de pontos de colocação da malha trapezoidal.

3.1.2 `packZ.m`

Esta função recebe como argumentos um intervalo de tempo, a matriz de estado, a matriz de controle e os fatores de escalonamento, e realiza o empacotamento dessas variáveis de decisão em um único vetor. A função também retorna uma `struct` com informações sobre o empacotamento, de modo que seja possível desempacotar as variáveis de decisão posteriormente.

Tal função é necessária para a inclusão dos tempos inicial e final como variáveis de decisão do problema. Uma opção mais simples seria incluir o vetor de tempos na matriz com os estados e controles, porém, como os tempos intermediários são bem definidos para dados intervalos de tempo e número de pontos de colocação, isso apenas incrementaria a quantidade de variáveis passadas ao solver, sem adicionar informações extras.

3.1.3 `unpackZ.m`

Esta função recebe como argumentos o vetor de variáveis de decisão do problema, a `struct` com informações sobre o empacotamento e uma `flag` indicando se as variáveis devem ser desescalonadas. A função retorna os valores das variáveis de estado e de controle, desempacotados e eventualmente desescalonados, e o vetor de tempos.

3.1.4 `computeDefects.m`

Esta função recebe como argumentos o intervalo de tempo, a matriz de estado e a matriz de derivadas de estado, e retorna a matriz de defeitos da colocação trapezoidal. A matriz de defeitos é calculada utilizando a Equação 2.16, conforme descrito na Seção 2.2.1.

3.1.5 `evaluateConstraints.m`

Esta função recebe como argumentos o vetor de variáveis de decisão do problema, a `struct` com informações sobre o empacotamento, a função `handle` que define a dinâmica do sistema, as funções `handle` de restrições de defeitos, de restrições de caminho e de restrições de contorno, e retorna dois vetores de restrições não lineares, de desigualdade e de igualdade.

Inicialmente desempacotam-se os valores das variáveis de decisão, obtendo-se o tempo, o estado e o controle, escalonados e não escalonados. As últimas serão utilizadas para o cálculo das derivadas de estado, que devem grandezas físicas. Em seguida, calcula-se as derivadas de estado, escala-se e calcula-se os defeitos. Por fim, avaliam-se as restrições de defeitos, de caminho e de contorno, retornando os vetores de restrições de desigualdade e de igualdade.

3.1.6 `evaluateObjective.m`

Esta função recebe como argumentos o vetor de variáveis de decisão do problema, a `struct` com informações sobre o empacotamento, as funções `handle` de termo de contorno e de termo de caminho da função objetivo, e retorna o valor da função objetivo.

Inicialmente desempacotam-se os valores das variáveis de decisão, obtendo-se o tempo, o estado e o controle. Em seguida, avaliam-se os termos de contorno e de caminho da função objetivo, retornando o seu valor final. Ressalta-se a utilização da função `trapz` para a integração numérica do termo de caminho, conforme Equação 2.17.

3.1.7 `spline2.m`

Esta função recebe como argumentos o vetor de tempos antigo, a matriz de estados antiga, a matriz de derivadas de estado antigas e o vetor de tempos novo, e retorna a matriz de estados e a matriz de derivadas de estado interpolados e avaliados no tempo novo.

A função utiliza um *loop* para calcular os valores interpolados um por vez. Dentro do *loop*, primeiramente encontra-se o intervalo de tempo no qual o tempo de interesse se encontra, utilizando a função `find` para localizar o índice do primeiro elemento maior ou igual ao tempo de interesse. Caso o tempo de interesse seja menor que o tempo inicial, o primeiro intervalo é utilizado. Caso o tempo de interesse seja maior que o tempo final, o último intervalo é utilizado.

Em seguida, calcula-se o passo de tempo local, dado pela diferença entre o tempo final e o tempo inicial do intervalo, e, utilizando o index encontrado previamente, obtém-se os valores das variáveis de estado e de suas derivadas no intervalo. Por fim, utiliza-se a fórmula de interpolação de segundo grau, conforme Equação 2.14, para realizar a interpolação dos valores das variáveis de estado.

3.2 Problemas Exemplos

Nesta seção, descrevem-se três problemas exemplo de aplicação da biblioteca desenvolvida. Em ordem crescente de complexidade, são apresentados um problema de minimização de tempo de percurso de uma partícula em uma dimensão na Seção 3.2.2, um problema clássico de otimização, o problema da braquistócrona, na Seção 3.2.3, e um problema de minimização de consumo energético de um veículo multirrotor, na Seção 3.2.4.

Além disso, para não poluir a leitura com a os códigos usados em cada problema, apresenta-se um modelo de utilização da biblioteca, descrito na Seção 3.2.1.

3.2.1 Modelo de utilização

A seguir, apresenta-se um modelo de utilização da biblioteca, que pode ser adaptado para a resolução de outros problemas, alterando-se apenas as funções de dinâmica, objetivo e restrições. Exemplos de implementação de cada uma dessas funções são apresentados no Anexo A.2.

```
clear; clc; close all;

% Adicione o caminho para a classe TrajectoryProblem
addpath('..');

% Crie uma instância do problema
nx = 2;  % Número de estados
nu = 1;  % Número de controles
problem = TrajectoryProblem(nx, nu);

% Defina as propriedades básicas do problema
problem.setTimeBoundaryConditions(0, 1);  % Intervalo de tempo [0,1]
problem.setBoundaryConditions(x0, xF);  % Condições iniciais e finais
problem.setStateBounds(xLow, xUpp);  % Limites de estado
problem.setControlBounds(uLow, uUpp);  % Limites de controle
```

```matlab

% Defina as funções necessárias
problem.setDynamics(@systemDynamics);  % Dinâmica do sistema
problem.setObjective(@boundaryObj, @pathObj);  % Funções objetivo
problem.setConstraints(@boundaryConst, @pathConst);  % Restrições opcionais

% Defina as opções do solver
options = optimoptions('fmincon', 'Display', 'iter');
nGrid = [50, 100, 200];  % Várias iterações com refinamento de malha
problem.setSolverOptions(options, nGrid);

% Resolva o problema
solution = problem.solveWithTrapezoidalCollocation();
```

Código Fonte 1 – Modelo de implementação

3.2.2 Movimento simples em uma dimensão

Considere um ponto material de massa m que se move sobre uma linha reta, sujeito a uma força **F**, conforme ilustrado na Figura 3.5. O problema é modelado utilizando-se a posição e velocidade da partícula como variáveis de estado e a força como variável de controle. O vetor estado, o vetor controle e a dinâmica do sistema são descritos pela Equação 3.1.

Por fim, o problema consiste em encontrar a trajetória que minimiza o quadrado da força no percurso entre dois pontos - Equação 3.2.

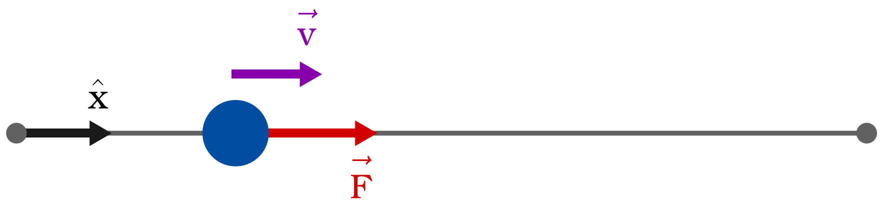


FIGURA 3.5 – Diagrama de forças no movimento simples

$$\mathbf{x} = \begin{bmatrix} s_x \\ v_x \end{bmatrix}, \qquad \mathbf{u} = \begin{bmatrix} F \end{bmatrix}, \qquad \dot{\mathbf{x}} = \begin{bmatrix} v \\ F/m \end{bmatrix}, \tag{3.1}$$

$$J = \int_{t_0}^{t_f} \left[F^2 \right] dt. \tag{3.2}$$

3.2.3 Braquistócrona

O problema da braquistócrona, proposto por Johann Bernoulli em 1696, consiste em encontrar a curva que minimiza o tempo de descida de uma partícula entre dois pontos sob ação da gravidade, desconsiderando o atrito. O nome vem do grego "*brachistos*" (mais curto) e "*chronos*" (tempo).

Embora possa parecer intuitivo que a linha reta seria a solução ótima por ser o caminho mais curto entre dois pontos, a solução do problema é uma curva cicloide. Isso ocorre porque, apesar do caminho ser mais longo, a partícula consegue atingir velocidades maiores ao longo da trajetória devido à maior inclinação inicial, resultando em um tempo total menor.

Este problema é considerado um dos marcos históricos do cálculo variacional e da otimização de trajetória, tendo sido resolvido por grandes matemáticos como Newton, Leibniz e o próprio Bernoulli. Sua solução analítica pode ser obtida através do princípio de Hamilton, mas aqui será utilizada uma abordagem numérica através da colocação trapezoidal.

Para modelizar o problema, considere um ponto material de massa m sob ação da gravidade, que se move entre dois pontos fixos, conforme ilustrado na Figura 3.6. Apesar de sabermos que se trata de uma descida, a figura ilustra o ponto final com posições positivas ($s_x, f > 0$ e $s_y, f > 0$), de modo que o ângulo de inclinação da trajetória seja positivo e que se evitem problemas com convenções de sinais.

Utilizam-se as posições horizontal e vertical da partícula, além do valor absoluto da velocidade, como variáveis de estado e o ângulo de inclinação da trajetória como variável de controle. O vetor estado, o vetor controle e a dinâmica do sistema são descritos pelas Equações 3.3.

Por fim, o problema consiste em encontrar a trajetória que minimiza o tempo de descida entre dois pontos, conforme Equação 3.4.

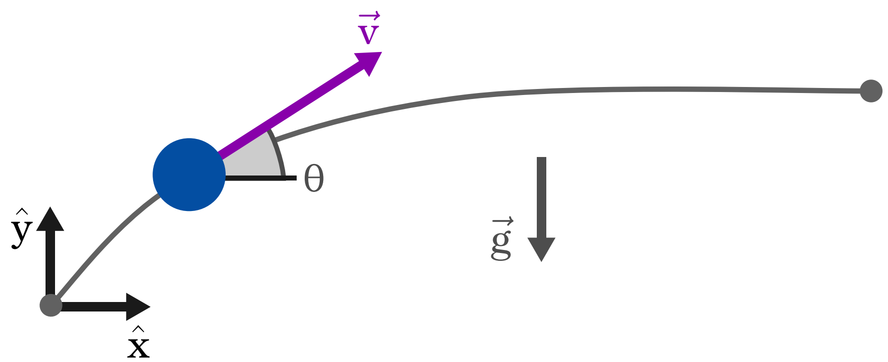


FIGURA 3.6 – Diagrama de forças no braquistócrona

$$\mathbf{x} = \begin{bmatrix} s_x \\ s_y \\ v \end{bmatrix}, \qquad \mathbf{u} = \begin{bmatrix} \theta \end{bmatrix}, \qquad \dot{\mathbf{x}} = \begin{bmatrix} v\cos(\theta) \\ v\sin(\theta) \\ -g\sin(\theta) \end{bmatrix}, \tag{3.3}$$

$$J = t_f. \tag{3.4}$$

3.2.4 Trajetória de subida de eVTOL

Por fim, apresenta-se o problema descrito em (COSTA, 2023), que consiste em encontrar a trajetória de subida que minimize o consumo energético de um veículo multirrotor. Para modelar o problema, as variáveis de decisão utilizadas foram as posições $[s_x, s_y]$, velocidades $[v_x, v_y]$, energia total da bateria utilizada E e comandos de tração dos rotores $[T_x, T_y]$, descritos pelas Equações 3.5.

$$\mathbf{x} = \begin{bmatrix} s_x \\ s_y \\ v_x \\ v_y \\ E \end{bmatrix}, \qquad \mathbf{u} = \begin{bmatrix} T_x \\ T_y \end{bmatrix}, \tag{3.5}$$

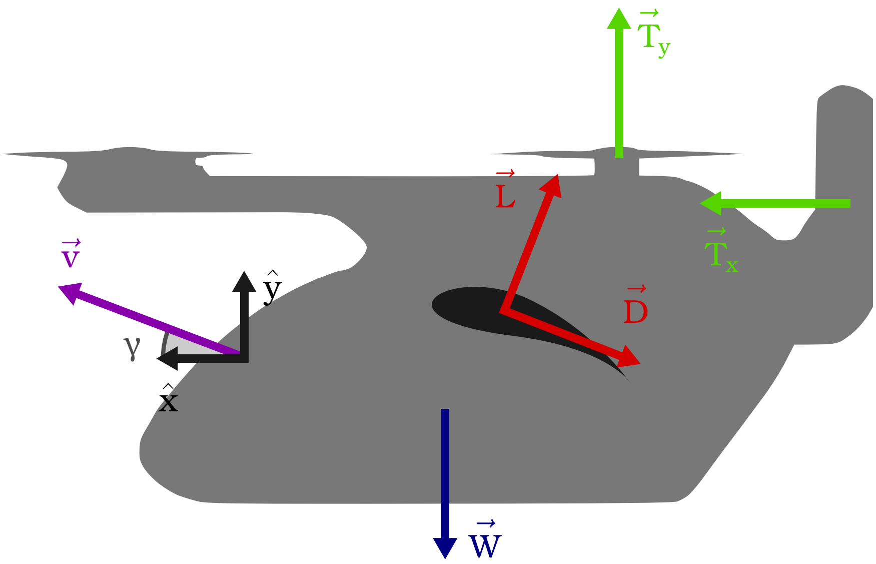


FIGURA 3.7 – Diagrama de forças no eVTOL

Conforme a Figura 3.7, a dinâmica do sistema pode ser descrita pelas Equações 3.6.

$$
\begin{aligned}
m\ddot{p}_x &= F_x = T_x - D(\mathbf{v})\cos(\gamma) - L(\mathbf{v})\sin(\gamma) \\
m\ddot{p}_y &= F_y = T_y - D(\mathbf{v})\sin(\gamma) + L(\mathbf{v})\cos(\gamma) - mg,
\end{aligned}
\tag{3.6}
$$

de modo que as restrições diferenciais escrevem-se

$$
\begin{aligned}
\dot{s}_x &= v_x \\
\dot{s}_y &= v_y \\
\dot{v}_x &= \frac{1}{m}F_x \\
\dot{v}_y &= \frac{1}{m}F_y,
\end{aligned}
,\qquad
\begin{aligned}
\dot{E} &= P_{total} \\
&= \begin{aligned}&(2P_{ind,rotor,x}(1+\chi) + T_x|\mathbf{v}|\cos(-\gamma)) \\ &+ (4P_{ind,rotor,y}(1+\chi) + T_y|\mathbf{v}|\sin(-\gamma))\,.\end{aligned}
\end{aligned}
\tag{3.7}
$$

Para finalizar a formulação do PCO, dado que se almeja o consumo mínimo da bateria, define-se a função objetivo como

$$
J = E_f. \tag{3.8}
$$

4 Resultados

Neste capítulo, são apresentados os resultados obtidos através da aplicação da biblioteca desenvolvida aos problemas-exemplo descritos no Capítulo 3. Para cada problema, são mostrados os gráficos das trajetórias encontradas, bem como a evolução temporal das variáveis de estado e controle. Além disso, são feitas análises comparativas com resultados da literatura, quando disponíveis, e discussões sobre a qualidade das soluções obtidas.

A apresentação dos resultados está organizada na mesma ordem dos problemas-exemplo: inicialmente, são mostrados os resultados do problema de movimento simples em uma dimensão na Seção 3.2.2; em seguida, os resultados do problema da braquistócrona na Seção 3.2.3; e, por fim, os resultados do problema de trajetória do eVTOL na Seção 3.2.4. Para cada problema, são também discutidos aspectos específicos da implementação e eventuais desafios encontrados durante o processo de otimização.

4.1 Movimento simples em uma dimensão

Como parâmetros para a resolução deste problema, utilizou-se $m = 1\,\mathrm{kg}$, $s_{x,0} = 0\,\mathrm{m}$, $s_{x,f} = 1\,\mathrm{m}$, $t_0 = 0\,\mathrm{s}$ e $t_f = 1\,\mathrm{s}$. Como chute inicial, interpolou-se linearmente os valores iniciais e finais das variáveis de estado e entre os valores $1\,\mathrm{N}$ e $-1\,\mathrm{N}$ para a variável de controle, como mostrado na Figura 4.1.

Como opções do solver, foram utilizados $n = 30$ pontos de colocação, sem necessidade de refinamento da malha e nem de ajustes extras de parâmetros para que houvesse convergência para a solução ótima. O resultado é apresentado na Figura 4.2.

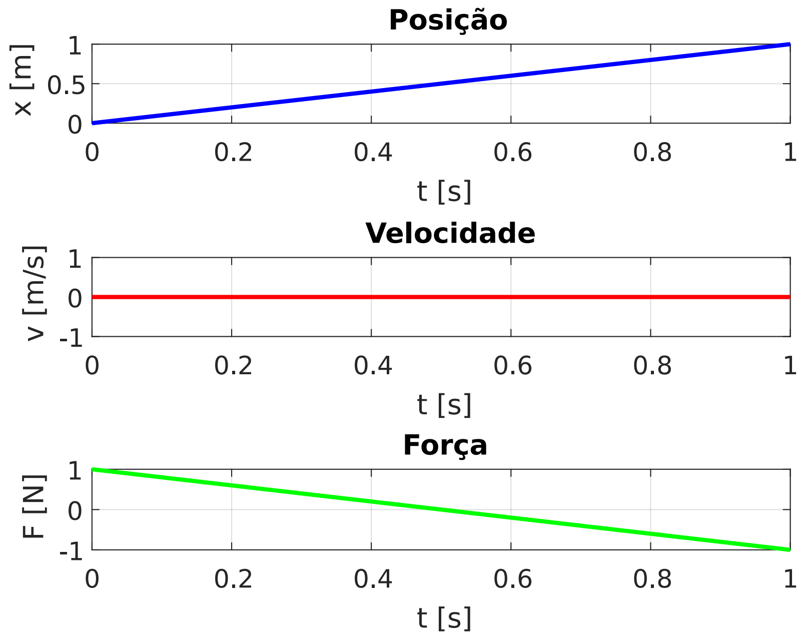


FIGURA 4.1 – Chute inicial para o problema de movimento simples em uma dimensão.

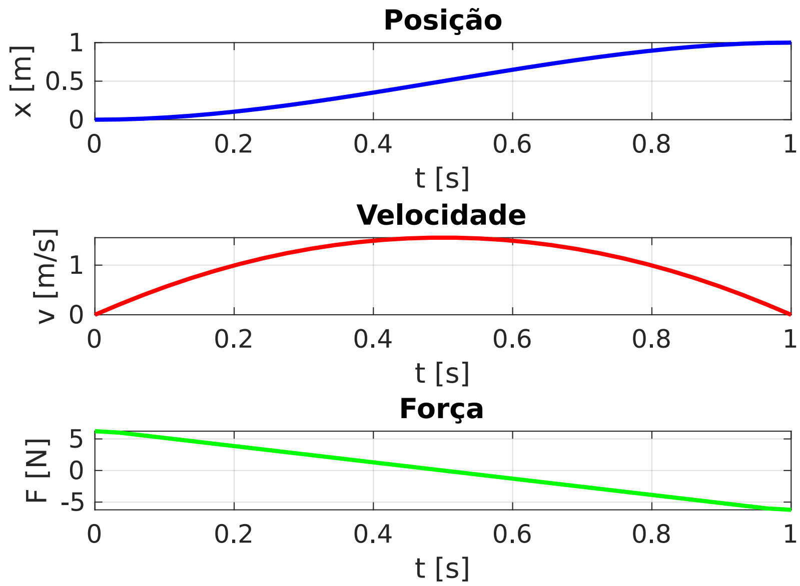


FIGURA 4.2 – Trajetória ótima do problema de movimento simples em uma dimensão.

4.2 Braquistócrona

Para a resolução deste problema, utilizou-se $m = 1\,\mathrm{kg}$, $s_0 = (0, 0)\,\mathrm{m}$, $s_f = (5, -5)\,\mathrm{m}$ e $v_0 = 0\,\mathrm{m\,s^{-1}}$ como parâmetros. Como chute inicial, utilizou-se uma parábola horizontal, como mostrado na Figura 4.3.

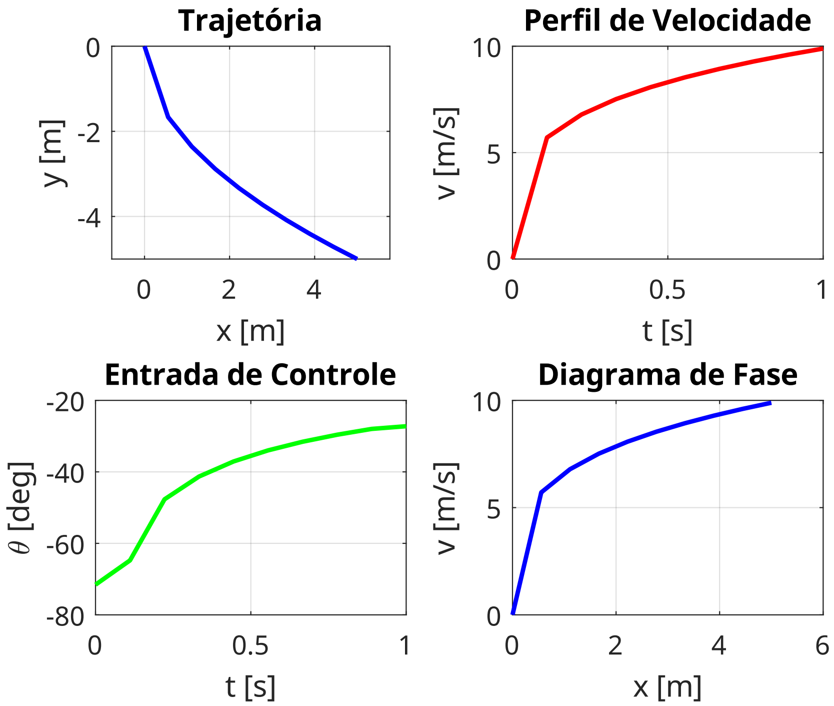


FIGURA 4.3 – Chute inicial para o problema de braquistócrona.

Como opções do solver, foi utilizada uma sequência de refinamento da malha, com $n = [10, 20, 40]$ pontos de colocação. Além disso, foram utilizadas diversas outras opções de parâmetros para que o solver convergisse para a solução ótima, como o uso de um algoritmo de otimização mais robusto e uma tolerância para a verificação de restrições mais restritiva. Apesar dos esforços, o solver não convergiu para a solução ótima na última iteração, de modo que não se encontrou a solução ótima. Ainda assim, encontrou-se um resultado factível, o qual é apresentado na Figura 4.4.

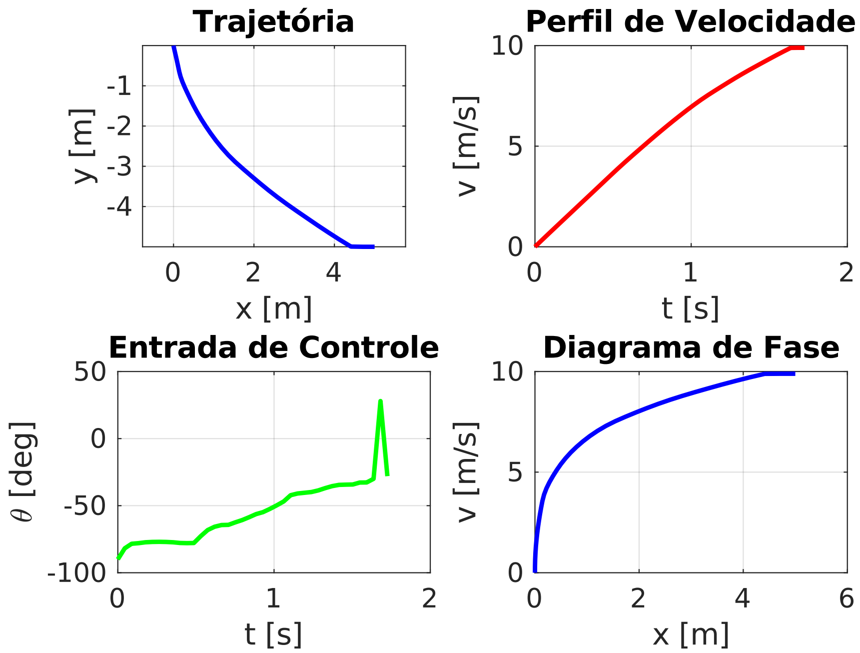


FIGURA 4.4 – Trajetória ótima do problema de braquistócrona.

Na Figura 4.5 é apresentada a comparação entre a solução obtida e a solução analítica do problema, a qual é uma ciclóide.

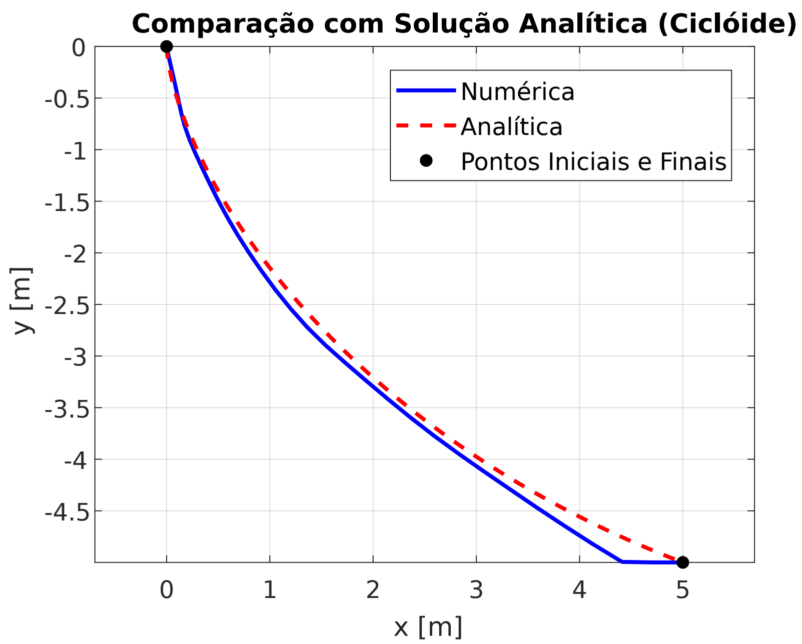


FIGURA 4.5 – Comparação entre a solução obtida e a solução analítica.

4.3 Trajetória do eVTOL

O problema considerado é o caso base do trabalho de referência, utilizando-se os mesmos parâmetros nele apresentados. Testou-se também a utilização do mesmo chute inicial, mas sem sucesso. Desse modo, construiu-se um chute inicial fisicamente plausível utilizando-se um polinômio de terceiro grau para a trajetória e respeitando as relações entre as variáveis de estado e controle, como mostrado na Figura 4.6.

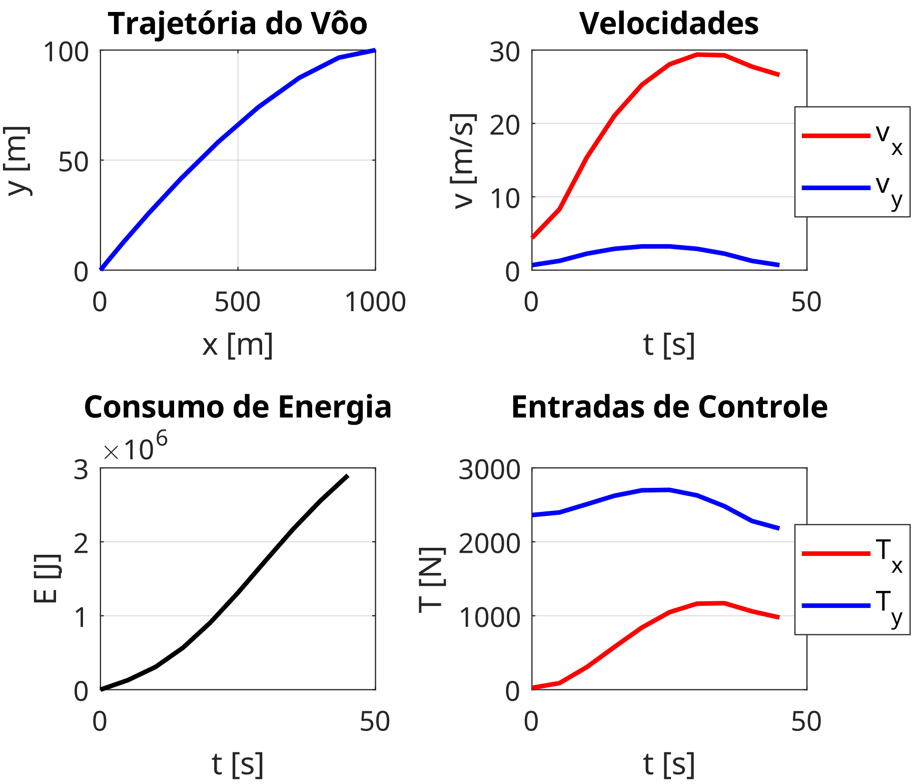


FIGURA 4.6 – Chute inicial para o problema de trajetória do eVTOL.

Como opções do solver, utilizou-se uma sequência de refinamento da malha, com $n = [10, 20, 40, 80]$ pontos de colocação. Além disso, assim como no problema da braquistócrona, foram utilizadas diversas outras opções de parâmetros para que o solver convergisse para a solução ótima, como o uso de um algoritmo de otimização mais robusto e uma tolerância para a verificação de restrições mais restritiva.

Apesar dos esforços, o solver não encontrou um ótimo local nem mesmo nas iterações com menos pontos na malha. O melhor resultado obtido é apresentado na Figura 4.7.

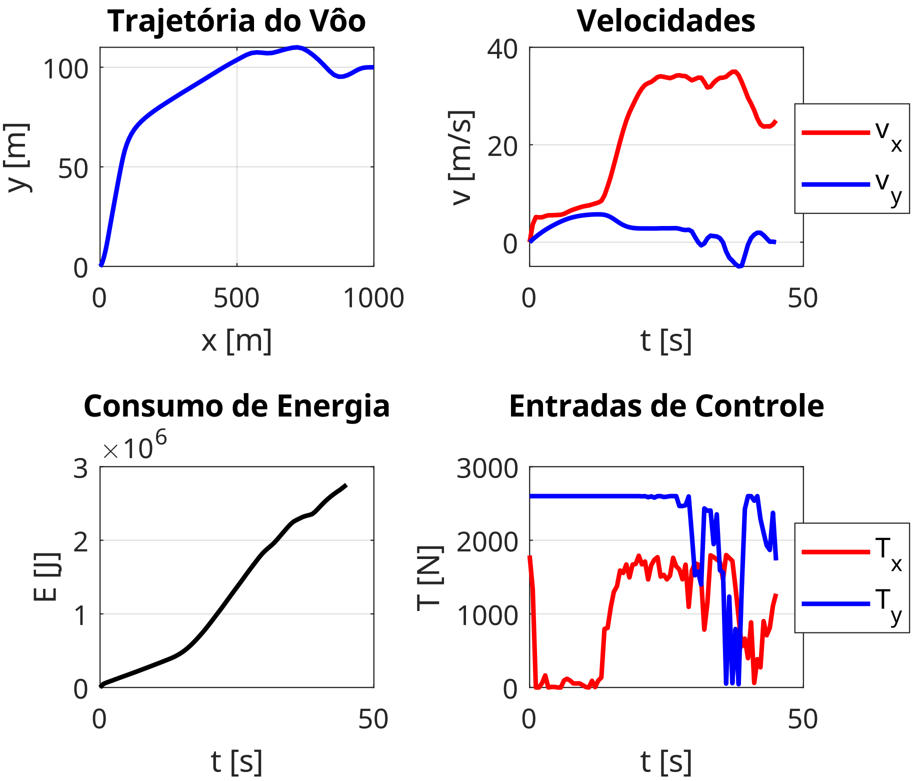


FIGURA 4.7 – Trajetória ótima do problema de trajetória do eVTOL.

Apesar da solução não ser a ótima e as entradas de controle não serem suaves, a trajetória encontrada é fisicamente plausível e o consumo de energia é menor do que o da solução de referência, sendo $E = 2,75484\,\text{MJ}$ para a solução obtida comparado a $E = 2,94342\,\text{MJ}$ da referência. A Figura 4.8 mostra a comparação entre a solução obtida e a solução de referência.

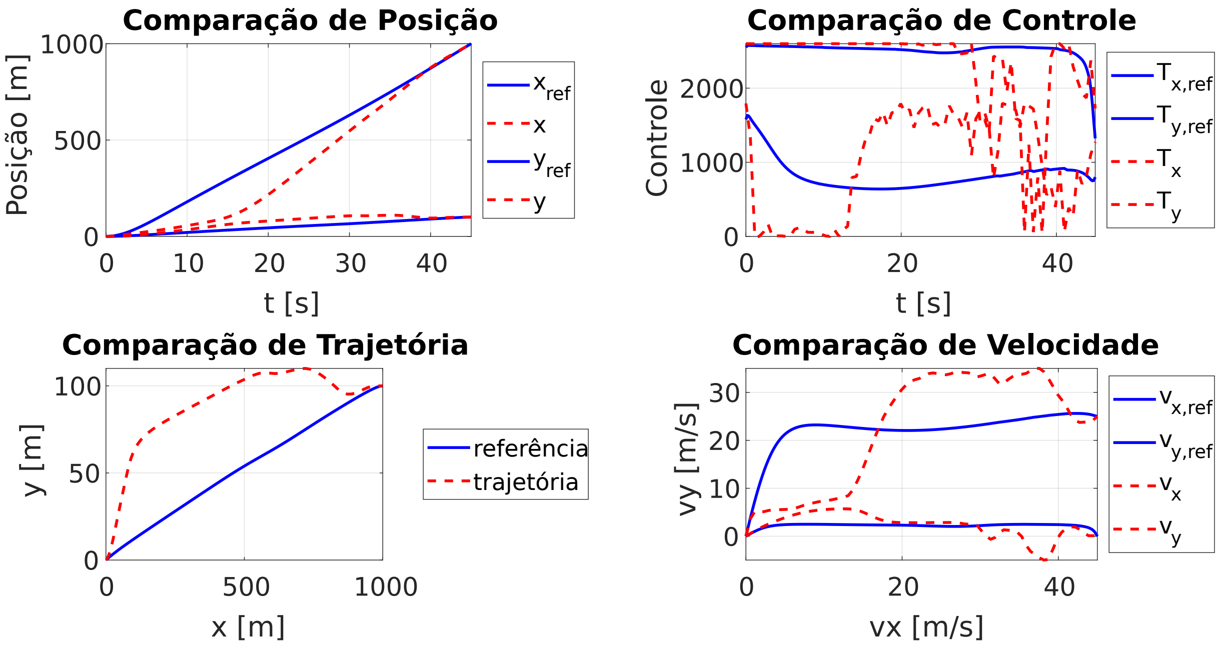


FIGURA 4.8 – Comparação entre a solução obtida e a solução de referência.

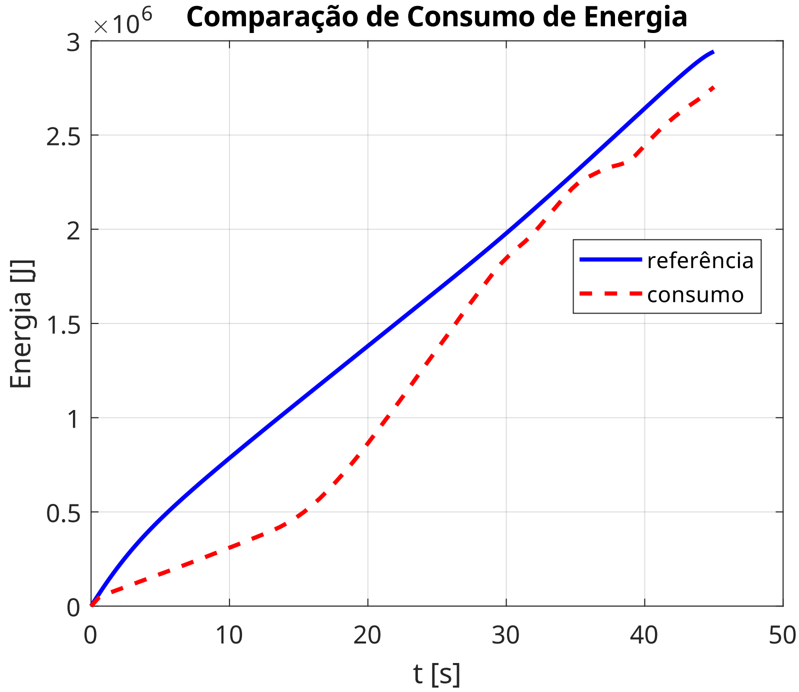


FIGURA 4.9 – Comparação da energia.

5 Conclusão

Este trabalho apresentou o desenvolvimento de uma biblioteca em MATLAB para otimização de trajetórias, com foco em problemas de controle ótimo. A biblioteca foi implementada de forma modular e flexível, permitindo sua aplicação a diferentes tipos de problemas através de uma interface unificada.

Os resultados obtidos nos problemas exemplo demonstraram a eficácia da biblioteca em realizar a transcrição de problemas de otimização de trajetória para problemas de programação não-linear, permitindo sua solução através de métodos de otimização numérica. Quanto aos problemas propostos, obteve-se a solução ótima apenas para aqueles de menor complexidade. Para os problemas mais complexos, a solução encontrada não foi de boa qualidade, necessitando persistir na otimização dos parâmetros utilizados para o solver, ou ainda indicando falhas de implementação na biblioteca.

Algumas das principais contribuições deste trabalho incluem o desenvolvimento de uma estrutura modular e extensível para problemas de otimização de trajetória, uma interface unificada que facilita a definição e solução de novos problemas, e a validação através de problemas exemplo com baixos níveis de complexidade.

Como sugestões para trabalhos futuros, sugere-se a verificação da implementação do escalonamento das variáveis de decisão, a depuração dos problemas exemplos mais complexos e a implementação de métodos adicionais de discretização, como o de Hermite-Simpson (KELLY, 2017; BETTS, 2010). Além disso, propõe-se o desenvolvimento de uma interface gráfica para facilitar a definição de problemas, a aplicação da biblioteca a outros problemas práticos mais complexos e a otimização do desempenho computacional para problemas de grande escala.

Por fim, conclui-se que os objetivos iniciais do trabalho foram parcialmente alcançados, resultando em uma ferramenta que fornece uma base para o estudo e solução de problemas de otimização de trajetória. A biblioteca desenvolvida pode servir como base para futuros estudos na área, desde que sejam corrigidos os problemas encontrados.

Referências

ATHANS, M.; FALB, P. L. **Optimal Control: An Introduction to the Theory and Its Applications**. [S.l.]: Courier Corporation, 2007. Google-Books-ID: XJJDTSZ2HEEC. ISBN 978-0-486-45328-6. 15

BECERRA, V. **PSOPT Optimal Control Solver: User Manual**. 2022. Disponível em: <https://github.com/PSOPT/psopt/releases/tag/5.02>. 14, 15, 16, 18, 23

BECERRA, V. M. Optimal control. **Scholarpedia**, v. 3, n. 1, p. 5354, jan. 2008. ISSN 1941-6016. Disponível em: <http://www.scholarpedia.org/article/Optimal_control>. 15

BELLMAN, R. **Dynamic Programming**. [S.l.]: Princeton University Press, 2010. Google-Books-ID: 92aYDwAAQBAJ. ISBN 978-0-691-14668-3. 15

BETTS, J. T. Survey of Numerical Methods for Trajectory Optimization. **Journal of Guidance, Control, and Dynamics**, v. 21, n. 2, p. 193–207, mar. 1998. ISSN 0731-5090. Publisher: American Institute of Aeronautics and Astronautics. Disponível em: <https://arc.aiaa.org/doi/10.2514/2.4231>. 16

BETTS, J. T. **Practical Methods for Optimal Control and Estimation Using Nonlinear Programming: Second Edition**. [S.l.]: SIAM, 2010. Google-Books-ID: n9hLriD8Lb8C. ISBN 978-0-89871-857-7. 15, 16, 17, 20, 43

BRYSON, A. E. **Applied Optimal Control: Optimization, Estimation and Control**. [S.l.]: Routledge, 2018. Google-Books-ID: LFUPEAAAQBAJ. ISBN 978-1-351-46592-2. 15

COSTA, A. L. S. **Otimização de Trajetória de Subida de eVTOL utilizando o Método de Colocação Direta**. Tese (Mestrado) — Instituto Tecnológico de Aeronáutica, São José dos Campos, 2023. 23, 34

KELLY, M. An Introduction to Trajectory Optimization: How to Do Your Own Direct Collocation. **SIAM Review**, v. 59, n. 4, p. 849–904, jan. 2017. ISSN 0036-1445. Publisher: Society for Industrial and Applied Mathematics. Disponível em: <https://epubs.siam.org/doi/10.1137/16M1062569>. 16, 18, 20, 43

KELLY, M. P. **OptimTraj: Trajectory Optimization for Matlab**. 2022. Disponível em: <https://github.com/MatthewPeterKelly/OptimTraj>. 14, 15, 16, 23

KIRK, D. E. **Optimal Control Theory: An Introduction**. [S.l.]: Courier Corporation, 2004. Google-Books-ID: fCh2SAtWIdwC. ISBN 978-0-486-43484-1. 15

KáLMáN, R. E. Contributions to the Theory of Optimal Control. In: . [s.n.], 1960. Disponível em: <https://www.semanticscholar.org/paper/Contributions-to-the-Theory-of-Optimal-Control-K%C3%A1lm%C3%A1n/4602a97c4965a9f6c41c9a7eeaef5be8333dbaef>. 15

KáLMáN, R. E. A New Approach to Linear Filtering and Prediction Problems. **Journal of Basic Engineering**, v. 82, n. 1, p. 35–45, mar. 1960. ISSN 0021-9223. Disponível em: <https://doi.org/10.1115/1.3662552>. 15

LEWIS, F. L. *et al.* **Optimal Control**. [S.l.]: John Wiley & Sons, 2012. Google-Books-ID: U3Gtlot_hYEC. ISBN 978-1-118-12272-3. 15, 16

PAINE, G. **The application of the method of quasilinearization to the computation of optimal control**. [S.l.], 1967. Disponível em: <https://ntrs.nasa.gov/api/citations/19680002402/downloads/19680002402.pdf>. 16

PONTRYAGIN, L. S. **Mathematical Theory of Optimal Processes**. [S.l.]: CRC Press, 1987. Google-Books-ID: kwzq0F4cBVAC. ISBN 978-2-88124-077-5. 15

SIMPLICIO, H. **hsimplicio/tg-ita**. 2024. Disponível em: <https://github.com/hsimplicio/tg-ita>. 23, 46

SUSSMANN, H.; WILLEMS, J. 300 years of optimal control: from the brachystochrone to the maximum principle. **IEEE Control Systems Magazine**, v. 17, n. 3, p. 32–44, jun. 1997. ISSN 1941-000X. Conference Name: IEEE Control Systems Magazine. Disponível em: <https://ieeexplore.ieee.org/document/588098>. 15

Anexo A - Arquivos de Código Fonte

Nessa seção, serão apresentados os arquivos de código fonte do projeto. Os mesmo arquivos estão disponíveis no repositório do projeto (SIMPLICIO, 2024). Para melhor visualização, sugere-se que o leitor abra os arquivos no editor de sua preferência.

A.1 Arquivos da biblioteca

A.1.1 `TrajectoryProblem.m`

```
classdef TrajectoryProblem < handle
    % Class to define a trajectory optimization problem

    properties (SetAccess = private)
        % Problem dimensions
        nx  % Number of states
        nu  % Number of controls

        % Time boundary conditions
        t0 {mustBeNumeric}  % Initial time
        tF {mustBeNumeric}  % Final time
        timeSpan = {}  % Cell array of time spans

        % Boundary conditions
        x0  % Initial state
        xF  % Final state

        % Boundary conditions struct
        boundaryConditions

        % Time bounds
```

```
        t0Low {mustBeNumeric}  % Lower bound for initial time
        t0Upp {mustBeNumeric}  % Upper bound for initial time
        tFLow {mustBeNumeric}  % Lower bound for final time
        tFUpp {mustBeNumeric}  % Upper bound for final time

        % State and control bounds
        xLow  % Lower state bounds
        xUpp  % Upper state bounds
        uLow  % Lower control bounds
        uUpp  % Upper control bounds

        % Function handles
        dynamics  % System dynamics
        constraints  % Constraint function
        objective  % Objective function
        constraintsCheck  % Function handle for problem-specific
        % constraint checking

        % Constraint functions
        boundaryConstraints  % Boundary constraints function
        pathConstraints  % Path constraints function

        % Objective functions
        boundaryObjective  % Mayer term
        pathObjective  % Lagrange term

        % Parameters
        parameters  % Problem parameters

        % Solver properties
        nGrid = []  % Array of grid points for each iteration
        solverOptions = {}  % Cell array of options for each
        % iteration

        % Scaling factors
        stateScaling
        controlScaling
        timeScaling
        scaling % Struct containing stateScaling, controlScaling,
        % and timeScaling
```

```
    end

    properties
        % Optional properties
        description = ''  % Problem description
        stateNames = {}  % Names of state variables
        controlNames = {}  % Names of control variables
    end

    methods

        function obj = TrajectoryProblem(nx, nu)
            % Constructor
            if nargin < 2
                error('Must specify number of states and controls');
            end

            obj.nx = nx;
            obj.nu = nu;

            % Initialize state and control bounds
            obj.xLow = -1e8 * ones(nx, 1);
            obj.xUpp = 1e8 * ones(nx, 1);
            obj.uLow = -1e8 * ones(nu, 1);
            obj.uUpp = 1e8 * ones(nu, 1);

            % Initialize time bounds
            obj.t0Low = -1e8;
            obj.t0Upp = 1e8;
            obj.tFLow = -1e8;
            obj.tFUpp = 1e8;

            % Default time boundary conditions
            obj.t0 = 0;
            obj.tF = 1;
            obj.timeSpan{1} = [obj.t0, obj.tF];

            % Initialize boundary conditions with empty arrays
            obj.x0 = zeros(nx, 1);
            obj.xF = zeros(nx, 1);
```

```

            % Initialize scaling factors to 1
            obj.stateScaling = ones(nx, 1);
            obj.controlScaling = ones(nu, 1);
            obj.timeScaling = 1;
            obj.scaling = struct('stateScaling', obj.stateScaling, ...
                                 'controlScaling', obj.controlScaling, ...
                                 'timeScaling', obj.timeScaling);

            % Initialize boundary conditions struct
            obj.boundaryConditions = struct();
            obj.boundaryConditions.t0 = obj.t0;
            obj.boundaryConditions.tF = obj.tF;
            obj.boundaryConditions.x0 = obj.x0;
            obj.boundaryConditions.xF = obj.xF;
        end

        function setBoundaryConditions(obj, x0, xF)
            % Set boundary conditions
            validateattributes(x0, {'numeric'}, ...
                               {'vector', 'numel', obj.nx});
            validateattributes(xF, {'numeric'}, ...
                               {'vector', 'numel', obj.nx});
            obj.x0 = x0(:);  % Ensure column vector
            obj.xF = xF(:);
            obj.boundaryConditions.x0 = x0;
            obj.boundaryConditions.xF = xF;
        end

        function setTimeBoundaryConditions(obj, t0, tF)
            % Set time bounds
            validateattributes(t0, {'numeric'}, {'scalar'});
            validateattributes(tF, {'numeric'}, {'scalar', '>', t0});
            obj.t0 = t0;
            obj.tF = tF;
            obj.timeSpan{1} = [obj.t0, obj.tF];
            obj.boundaryConditions.t0 = t0;
            obj.boundaryConditions.tF = tF;
        end

```

```
        function setTimeBounds(obj, t0Low, t0Upp, tFLow, tFUpp)
            % Set bounds for initial and final times
            validateattributes(t0Low, {'numeric'}, {'scalar'});
            validateattributes(t0Upp, {'numeric'}, {'scalar'});
            validateattributes(tFLow, {'numeric'}, {'scalar'});
            validateattributes(tFUpp, {'numeric'}, {'scalar'});

            assert(t0Low <= t0Upp, ...
                   'Lower bound for t0 must be <= upper bound');
            assert(tFLow <= tFUpp, ...
                   'Lower bound for tF must be <= upper bound');
            assert(t0Upp <= tFLow, ...
                   'Upper bound for t0 must be <= lower bound for tF');

            obj.t0Low = t0Low;
            obj.t0Upp = t0Upp;
            obj.tFLow = tFLow;
            obj.tFUpp = tFUpp;
        end

        function setStateBounds(obj, xLow, xUpp)
            % Set state bounds
            validateattributes(xLow, {'numeric'}, ...
                               {'vector', 'numel', obj.nx});
            validateattributes(xUpp, {'numeric'}, ...
                               {'vector', 'numel', obj.nx});
            assert(all(xLow <= xUpp), ...
                   'Lower bounds must be <= upper bounds');
            obj.xLow = xLow(:);
            obj.xUpp = xUpp(:);
        end

        function setControlBounds(obj, uLow, uUpp)
            % Set control bounds
            validateattributes(uLow, {'numeric'}, ...
                               {'vector', 'numel', obj.nu});
            validateattributes(uUpp, {'numeric'}, ...
                               {'vector', 'numel', obj.nu});
            assert(all(uLow <= uUpp), ...
                   'Lower bounds must be <= upper bounds');
```

```
            obj.uLow = uLow(:);
            obj.uUpp = uUpp(:);
        end

        function setScaling(obj, type, factors)
            % Set scaling factors for variables
            % type: 'state', 'control', or 'time'
            % factors: vector of scaling factors

            switch lower(type)
                case 'state'
                    validateattributes(factors, {'numeric'}, ...
                                       {'vector', 'positive', ...
                                       'numel', obj.nx});
                    obj.stateScaling = factors(:);
                    obj.scaling.stateScaling = factors(:);
                case 'control'
                    validateattributes(factors, {'numeric'}, ...
                                       {'vector', 'positive', ...
                                       'numel', obj.nu});
                    obj.controlScaling = factors(:);
                    obj.scaling.controlScaling = factors(:);
                case 'time'
                    validateattributes(factors, {'numeric'}, ...
                                       {'scalar', 'positive'});
                    obj.timeScaling = factors;
                    obj.scaling.timeScaling = factors;
                otherwise
                    error(['Invalid scaling type. Must be', ...
                           '''state'', ''control'', or ''time''']);
            end
        end

        function setDynamics(obj, hDynamics)
            % Set dynamics function
            validateattributes(hDynamics, {'function_handle'}, {});
            % Test the function with dummy inputs
            time_test = zeros(1, 1);
            x_test = zeros(obj.nx, 1);
            u_test = zeros(obj.nu, 1);
```

```
                try
                    dx = hDynamics(time_test, x_test, ...
                                   u_test, obj.parameters);
                    validateattributes(dx, {'numeric'}, ...
                                       {'vector', 'numel', obj.nx});
                catch ME
                    error('Invalid dynamics function: %s', ME.message);
                end
                obj.dynamics = @(time, state, control) ...
                               hDynamics(time, state, ...
                                         control, obj.parameters);
            end

            function setObjective(obj, hBoundaryObjective, hPathObjective)
                % Set objective functions
                % Both inputs are optional - pass [] to skip

                % Validate boundary objective if provided
                if ~isempty(hBoundaryObjective)
                    validateattributes(hBoundaryObjective, ...
                                       {'function_handle'}, {});
                    try
                        val = hBoundaryObjective(zeros(obj.nx, 1), ...
                                                 zeros(obj.nx, 1), ...
                                                 obj.t0, obj.tF, ...
                                                 obj.parameters);
                        validateattributes(val, {'numeric'}, {'scalar'});
                    catch ME
                        error('Invalid boundary objective function: %s', ...
                              ME.message);
                    end
                    obj.boundaryObjective = @(x0, xF, t0, tF) ...
                                            hBoundaryObjective(x0, xF, ...
                                                               t0, tF, ...
                                                               obj.parameters);
                else
                    obj.boundaryObjective = [];
                end

                % Validate path objective if provided
```

```
            if ~isempty(hPathObjective)
                validateattributes(hPathObjective, ...
                                   {'function_handle'}, {});
                try
                    time_test = [obj.t0, obj.tF];
                    val = hPathObjective(time_test, ...
                                         zeros(obj.nx, 2), ...
                                         zeros(obj.nu, 2), ...
                                         obj.parameters);
                    validateattributes(val, {'numeric'}, ...
                                       {'size', [1, numel(time_test)]});
                catch ME
                    error('Invalid path objective function: %s', ...
                          ME.message);
                end
                obj.pathObjective = @(time, state, control) ...
                                    hPathObjective(time, state, ...
                                                   control, ...
                                                   obj.parameters);
            else
                obj.pathObjective = [];
            end

            % Create combined objective function
            obj.objective = @(z, packInfo) ...
                            evaluateObjective(z, packInfo, ...
                                              obj.boundaryObjective, ...
                                              obj.pathObjective);
        end

        function setConstraints(obj, hBoundaryConstraints, hPathConstraints)
            % Set constraint functions
            % Both inputs are optional - pass [] to skip

            % Validate boundary constraints if provided
            if ~isempty(hBoundaryConstraints)
                validateattributes(hBoundaryConstraints, ...
                                   {'function_handle'}, {});
                try
                    [c, ceq] = hBoundaryConstraints(zeros(obj.nx, 1), ...
```

```
                                                     zeros(obj.nx, 1), ...
                                                     obj.t0, obj.tF, ...
                                                     obj.boundaryConditions);
                     if ~isempty(c)
                         validateattributes(c, {'numeric'}, ...
                                            {'vector'});
                     end
                     if ~isempty(ceq)
                         validateattributes(ceq, {'numeric'}, ...
                                            {'vector'});
                     end
                 catch ME
                     error('Invalid boundary constraints function: %s', ...
                           ME.message);
                 end

                 % Scale boundary conditions
                 bndConditions = struct();
                 if isfield(obj.boundaryConditions, 't0')
                     bndConditions.t0 = obj.boundaryConditions.t0 / ...
                                        obj.timeScaling;
                 end
                 if isfield(obj.boundaryConditions, 'tF')
                     bndConditions.tF = obj.boundaryConditions.tF / ...
                                        obj.timeScaling;
                 end
                 if isfield(obj.boundaryConditions, 'x0')
                     bndConditions.x0 = obj.boundaryConditions.x0 / ...
                                        obj.stateScaling;
                 end
                 if isfield(obj.boundaryConditions, 'xF')
                     bndConditions.xF = obj.boundaryConditions.xF / ...
                                        obj.stateScaling;
                 end
                 obj.boundaryConstraints = @(x0, xF, t0, tF) ...
                                          hBoundaryConstraints(x0, xF, ...
                                                               t0, tF, ...
                                                               bndConditions);
             else
                 obj.boundaryConstraints = [];
```

```
            end

            % Validate path constraints if provided
            if ~isempty(hPathConstraints)
                validateattributes(hPathConstraints, ...
                                   {'function_handle'}, {});
                try
                    time_test = [obj.t0, obj.tF];
                    [c, ceq] = hPathConstraints(time_test, ...
                                                zeros(obj.nx, 2), ...
                                                zeros(obj.nu, 2), ...
                                                obj.parameters);
                    if ~isempty(c)
                        validateattributes(c, {'numeric'}, ...
                                           {'vector'});
                    end
                    if ~isempty(ceq)
                        validateattributes(ceq, {'numeric'}, ...
                                           {'vector'});
                    end
                catch ME
                    error('Invalid path constraints function: %s', ...
                          ME.message);
                end
                obj.pathConstraints = @(time, state, control) ...
                                      hPathConstraints(time, state, ...
                                                       control, ...
                                                       obj.parameters);
            else
                obj.pathConstraints = [];
            end

            % Create combined constraints function
            obj.constraints = @(z, packInfo) ...
                              evaluateConstraints(z, packInfo, ...
                                                  obj.dynamics, ...
                                                  @computeDefects, ...
                                                  obj.pathConstraints, ...
                                                  obj.boundaryConstraints);
        end
```

```

        function setParameters(obj, parameters)
            % Set problem parameters
            obj.parameters = parameters;
        end

        function setVariableNames(obj, stateNames, controlNames)
            % Set names for variables (optional)
            if nargin > 1 && ~isempty(stateNames)
                validateattributes(stateNames, {'cell'}, ...
                                  {'numel', obj.nx});
                obj.stateNames = stateNames;
            end
            if nargin > 2 && ~isempty(controlNames)
                validateattributes(controlNames, {'cell'}, ...
                                  {'numel', obj.nu});
                obj.controlNames = controlNames;
            end
        end

        function setSolverOptions(obj, options, nGrid)
            % Set solver options and grid points
            % options: either a single optimoptions('fmincon') object
            %          or cell array of options for each iteration
            % nGrid: array of grid points for each iteration

            validateattributes(nGrid, {'numeric'}, ...
                              {'vector', 'positive', 'integer'});

            if ~iscell(options)
                % If single options provided,
                % replicate for all iterations
                validateattributes(options, ...
                                  {'optim.options.Fmincon'}, {});
                obj.solverOptions = repmat({options}, 1, length(nGrid));
            else
                % If cell array provided, validate length matches nGrid
                validateattributes(options, {'cell'}, ...
                                  {'numel', length(nGrid)});
                cellfun(@(opt) validateattributes(opt, ...
```

```
                                                {'optim.options.Fmincon'}, ...
                                                {}), options);
                obj.solverOptions = options;
            end

            obj.nGrid = nGrid;
        end

        function setConstraintsCheck(obj, hConstraintsCheck)
            % Set function handle for problem-specific
            % constraint checking
            validateattributes(hConstraintsCheck, ...
                               {'function_handle'}, {});

            % Test the function with dummy inputs
            time_test = linspace(obj.t0, obj.tF, 2);
            state_test = zeros(obj.nx, 2);
            control_test = zeros(obj.nu, 2);
            try
                violations = hConstraintsCheck(time_test, state_test, ...
                                               control_test, ...
                                               obj.parameters);
                validateattributes(violations, {'struct'}, {});
            catch ME
                error('Invalid constraints check function: %s', ...
                      ME.message);
            end

            obj.constraintsCheck = hConstraintsCheck;
        end

        function valid = validate(obj)
            % Validate that all required properties are set

            % Check each requirement individually
            hasDynamics = ~isempty(obj.dynamics);
            hasObjective = ~isempty(obj.objective);
            hasFiniteBounds = ~any(isinf([obj.xLow; obj.xUpp; ...
                                          obj.uLow; obj.uUpp]));
            hasGrid = ~isempty(obj.nGrid);
```

```
            hasOptions = ~isempty(obj.solverOptions);
            gridMatchesOptions = length(obj.nGrid) == ...
                                 length(obj.solverOptions);

            % Combine all checks
            valid = hasDynamics && hasObjective && hasFiniteBounds && ...
                    hasGrid && hasOptions && gridMatchesOptions;

            % Provide detailed warning if invalid
            if ~valid
                warning('TrajectoryProblem:Incomplete', ...
                        ['Problem definition is incomplete.', ...
                         ' Missing requirements:\n%s%s%s%s%s%s', ...
                         conditional_msg(~hasDynamics, ...
                                         '- Dynamics function not set'), ...
                         conditional_msg(~hasObjective, ...
                                         '- Objective function not set'), ...
                         conditional_msg(~hasFiniteBounds, ...
                                         '- Infinite bounds detected'), ...
                         conditional_msg(~hasGrid, ...
                                         '- Grid points not set'), ...
                         conditional_msg(~hasOptions, ...
                                         '- Solver options not set'), ...
                         conditional_msg(~gridMatchesOptions, ...
                                         ['- Number of grid points does', ...
                                          ' not match number of solver options'])]);
            end

            function msg = conditional_msg(condition, message)
                if condition
                    msg = message;
                else
                    msg = '';
                end
            end

        end

        function info = summarize(obj)
            % Generate problem summary
```

```
            info = struct();
            info.numStates = obj.nx;
            info.numControls = obj.nu;
            info.timespan = [obj.t0, obj.tF];
            info.stateBounds = [obj.xLow, obj.xUpp];
            info.controlBounds = [obj.uLow, obj.uUpp];
            if ~isempty(obj.stateNames)
                info.stateNames = obj.stateNames;
            end
            if ~isempty(obj.controlNames)
                info.controlNames = obj.controlNames;
            end
        end

        function [state, control] = generateInitialGuess(obj)
            % Generate initial guess with reasonable values for
            % free final states

            % Validate boundary conditions exist
            if isempty(obj.x0) || isempty(obj.xF)
                error(['TrajectoryProblem:NoBoundaryConditions: ', ...
                       'Either pass initial guess or set', ...
                       ' boundary conditions to generate a', ...
                       ' linear interpolation initial guess.']);
            end

            timeGrid = linspace(obj.t0, obj.tF, obj.nGrid(1));

            % Initialize state array
            state = zeros(obj.nx, obj.nGrid(1));

            % For each state variable
            for i = 1:obj.nx
                if abs(obj.xF(i)) > 1e6  % Detect "free" final states
                    % Use a reasonable final value instead
                    % of the large number
                    if obj.xF(i) > 0
                        % Use half of upper bound
                        finalVal = max(obj.x0(i), ...
                                       obj.xUpp(i) / 2);
```

```
                    else
                        % Use half of lower bound
                        finalVal = min(obj.x0(i), ...
                                       obj.xLow(i) / 2);
                    end
                    state(i, :) = interp1([obj.t0, obj.tF], ...
                                          [obj.x0(i), finalVal], ...
                                          timeGrid);
                else
                    % Normal linear interpolation
                    % for fixed final states
                    state(i, :) = interp1([obj.t0, obj.tF], ...
                                          [obj.x0(i), obj.xF(i)], ...
                                          timeGrid);
                end
            end

            % Initialize controls to zeros
            control = zeros(obj.nu, obj.nGrid(1));
        end

        function solution = solveWithTrapezoidalCollocation(obj, guess)
            % Solve the trajectory optimization problem
            % with trapezoidal collocation
            if ~obj.validate()
                error('TrajectoryProblem:InvalidProblem', ...
                      'Problem is not completely defined');
            end

            nIter = length(obj.nGrid);
            solution(nIter) = struct();

            for i = 1:nIter
                disp(['Iteration ', num2str(i), ' of ', ...
                      num2str(nIter)]);

                % Generate or interpolate initial guess
                if i == 1
                    timeGrid = linspace(obj.t0, obj.tF, obj.nGrid(1));
                    if nargin < 2 || isempty(guess)
```

```
                    [stateGuess, controlGuess] = ...
                        obj.generateInitialGuess();
                    [zGuess, packInfo] = packZ([obj.t0, obj.tF], ...
                                                stateGuess, ...
                                                controlGuess, ...
                                                obj.scaling);
                else
                    [zGuess, ...
                     packInfo] = packZ([obj.t0, obj.tF], ...
                                        guess(1:obj.nx, :), ...
                                        guess(obj.nx + 1:end, :), ...
                                        obj.scaling);
                end
            else
                % Validate time span
                assert(all(isfinite(obj.timeSpan{i - 1})), ...
                       'Non-finite values in previous time span');
                assert(all(isfinite(obj.timeSpan{i})), ...
                       'Non-finite values in current time span');
                assert(obj.timeSpan{i - 1}(2) >= ...
                       obj.timeSpan{i - 1}(1), ...
                       'Invalid previous time span');
                assert(obj.timeSpan{i}(2) >= obj.timeSpan{i}(1), ...
                       'Invalid current time span');

                timeGrid = linspace(obj.timeSpan{i}(1), ...
                                    obj.timeSpan{i}(2), ...
                                    obj.nGrid(i));
                timeOld = linspace(obj.timeSpan{i - 1}(1), ...
                                   obj.timeSpan{i - 1}(2), ...
                                   obj.nGrid(i - 1));

                % Validate previous solution
                assert(all(isfinite(solution(i - 1).z.state(:))), ...
                       'Non-finite values in previous state');
                assert(all(isfinite(solution(i - 1).z.control(:))), ...
                       'Non-finite values in previous control');

                % Perform interpolation with validation
                stateGuess = zeros(obj.nx, obj.nGrid(i));
```

```
                    controlGuess = zeros(obj.nu, obj.nGrid(i));

                    % Interpolate each state and control
                    % separately to maintain dimensions
                    for j = 1:obj.nx
                        [stateGuess(j, :), ~] = ...
                            spline2(timeOld, ...
                                    solution(i - 1).z.state(j, :), ...
                                    solution(i - 1).z.derivatives(j, :), ...
                                    timeGrid);
                    end
                    for j = 1:obj.nu
                        controlGuess(j, :) = ...
                            interp1(timeOld, ...
                                    solution(i - 1).z.control(j, :), ...
                                    timeGrid, ...
                                    'linear', 'extrap');
                    end

                    % Validate interpolation results
                    assert(all(isfinite(stateGuess(:))), ...
                           'Non-finite values in interpolated state');
                    assert(all(isfinite(controlGuess(:))), ...
                           'Non-finite values in interpolated control');

                    % Pack with validation
                    [zGuess, packInfo] = packZ(obj.timeSpan{i}, ...
                                              stateGuess, ...
                                              controlGuess, ...
                                              obj.scaling);
                    assert(all(isfinite(zGuess)), ...
                           'Non-finite values in packed guess');
                end

                % Set up the problem
                fun = @(z)(obj.objective(z, packInfo));
                A = [];
                b = [];
                Aeq = [];
                beq = [];
```

```
            lb = [
                   obj.t0Low / obj.scaling.timeScaling
                   obj.tFLow / obj.scaling.timeScaling
                   reshape(repmat(obj.xLow ./ ...
                                  obj.scaling.stateScaling, 1, ...
                                  obj.nGrid(i)), [], 1)
                   reshape(repmat(obj.uLow ./ ...
                                  obj.scaling.controlScaling, 1, ...
                                  obj.nGrid(i)), [], 1)
                 ];
            ub = [
                   obj.t0Upp / obj.scaling.timeScaling
                   obj.tFUpp / obj.scaling.timeScaling
                   reshape(repmat(obj.xUpp ./ ...
                                  obj.scaling.stateScaling, 1, ...
                                  obj.nGrid(i)), [], 1)
                   reshape(repmat(obj.uUpp ./ ...
                                  obj.scaling.controlScaling, 1, ...
                                  obj.nGrid(i)), [], 1)
                 ];
            nonlcon = @(z)(obj.constraints(z, packInfo));

            % Solve scaled problem
            [z_scaled, ...
             fval, ...
             exitflag, ...
             output] = fmincon(fun, zGuess, ...
                               A, b, ...
                               Aeq, beq, ...
                               lb, ub, ...
                               nonlcon, ...
                               obj.solverOptions{i});

            % Unpack solution with scaling up
            [time, state, control] = unpackZ(z_scaled, ...
                                             packInfo, true);

            % Set time span for next iteration
            obj.timeSpan{i + 1} = [time(1), time(end)];

```

```
                solution(i).nGrid = obj.nGrid(i);
                solution(i).z = struct();
                solution(i).z.timeSpan = obj.timeSpan{i + 1};
                solution(i).z.time = timeGrid;
                solution(i).z.state = state;
                solution(i).z.control = control;
                solution(i).z.derivatives = ...
                    obj.dynamics(timeGrid, state, control);
                solution(i).fval = fval;
                solution(i).exitflag = exitflag;
                solution(i).output = output;

                % Check constraints if a check function is provided
                if ~isempty(obj.constraintsCheck)
                    solution(i).violations = ...
                        obj.constraintsCheck(timeGrid, ...
                                             solution(i).z.state, ...
                                             solution(i).z.control, ...
                                             obj.parameters);
                end
            end
        end

        function [t0, tF] = getTimeBounds(obj)
            % Get time bounds
            t0 = obj.t0;
            tF = obj.tF;
        end

        function [x0, xF] = getBoundaryConditions(obj)
            % Get boundary conditions
            x0 = obj.x0;
            xF = obj.xF;
        end

        function parameters = getParameters(obj)
            % Get problem parameters
            parameters = obj.parameters;
        end

```

```
        function nGrid = getGridSize(obj)
            % Get grid size
            nGrid = obj.nGrid;
        end

    end
end
```

Código Fonte 2 – Classe `TrajectoryProblem`

A.1.2 `packZ.m`

```
function [z, packInfo] = packZ(tSpan, state, control, scaling)
    % Pack time bounds and trajectories into optimization vector
    % Inputs:
    %   tSpan: [1,2] = time span
    %   state: [nx,nGrid] = state trajectory
    %   control: [nu,nGrid] = control trajectory
    %   scaling: struct = scaling factors
    % Outputs:
    %   z: [nz,1] = decision variables
    %   packInfo: struct = information about the problem

    % Pack packInfo
    packInfo = struct();
    packInfo.nx = size(state, 1);
    packInfo.nu = size(control, 1);
    packInfo.nGrid = size(state, 2);
    packInfo.scaling = scaling;

    % Scale variables
    tScaled = tSpan / scaling.timeScaling;
    stateScaled = state ./ repmat(scaling.stateScaling, 1, ...
                                  size(state, 2));
    controlScaled = control ./ repmat(scaling.controlScaling, 1, ...
                                      size(control, 2));

    % Pack time bounds and trajectories into optimization vector
    z = [
```

```
        tScaled(:)
        stateScaled(:)
        controlScaled(:)
       ];
end
```

Código Fonte 3 – Função packZ

A.1.3 unpackZ.m

```
function [time, state, control] = unpackZ(z, packInfo, scalingFlag)
    % Unpack optimization vector into components
    % Inputs:
    %   z: [nz,1] = decision variables
    %   packInfo: struct = information about the problem
    %   scalingFlag: logical = flag to indicate if scaling
    %                          should be applied
    % Outputs:
    %   time: [1,nGrid] = time vector
    %   state: [nx,nGrid] = state variables
    %   control: [nu,nGrid] = control variables

    % Unpack packInfo
    nx = packInfo.nx;
    nu = packInfo.nu;
    nGrid = packInfo.nGrid;

    % Calculate expected sizes
    nStates = nx * nGrid;
    nControls = nu * nGrid;
    expectedLength = 2 + nStates + nControls;  % accounts for time bounds

    % Validate z length
    assert(length(z) == expectedLength, ...
           'z length mismatch. Expected %d elements but got %d', ...
           expectedLength, length(z));

    % Unpack optimization vector into components
    tSpan = z(1:2);
```

```
    z_rest = z(3:end);

    % Reshape state and control trajectories
    nStates = nx * nGrid;
    state = reshape(z_rest(1:nStates), [nx, nGrid]);
    control = reshape(z_rest(nStates + 1:end), [nu, nGrid]);

    % Apply scaling up if requested
    if nargin >= 3 && scalingFlag
        tSpan = tSpan * packInfo.scaling.timeScaling;
        state = state .* repmat(packInfo.scaling.stateScaling, ...
                                1, nGrid);
        control = control .* repmat(packInfo.scaling.controlScaling, ...
                                    1, nGrid);
    end

    % Create time vector
    time = linspace(tSpan(1), tSpan(2), nGrid);
end
```

Código Fonte 4 – Função unpackZ

A.1.4 computeDefects.m

```
function defects = computeDefects(timeStep, state, stateDerivatives)
    % This function computes the defects for
    % direct transcription using the Trapezoidal Rule.
    %
    % Inputs:
    %   timeStep: scalar = time step
    %   state: [nStates, nGrid] = matrix of states
    %   stateDerivatives: [nStates, nGrid-1] = matrix of
    %                                           state derivatives
    %
    % Outputs:
    %   defects: [nStates, nGrid-1] = matrix of defects

    nGrid = size(state, 2);

```

```
    indexLower = 1:(nGrid - 1);
    indexUpper = 2:nGrid;

    stateLower = state(:, indexLower);
    stateUpper = state(:, indexUpper);

    derivativesLower = stateDerivatives(:, indexLower);
    derivativesUpper = stateDerivatives(:, indexUpper);

    % Apply the Trapezoidal Rule
    defects = stateUpper - stateLower - ...
              0.5 * timeStep * (derivativesLower + derivativesUpper);
end
```

Código Fonte 5 – Função `computeDefects`

A.1.5 evaluateConstraints.m

```
function [c, ceq] = evaluateConstraints(z, packInfo, ...
                                        dynamics, ...
                                        defectConstraints, ...
                                        pathConstraints, ...
                                        boundaryConstraints)
    % Define the collocation constraints
    % Inputs:
    %   z: [nz,1] = decision variables
    %   packInfo: struct = information about the problem
    %   dynamics: function = dynamics function
    %   defectConstraints: function = defect constraints function
    %   pathConstraints: function = path constraints function
    %   boundaryConstraints: function = boundary constraints function
    % Outputs:
    %   c = [m,1] = inequality constraints
    %   ceq = [m,1] = equality constraints

    % Check for NaN values in z
    if any(isnan(z))
        error('NaN values detected in decision variables');
    end
```

```

    % Unpack z
    [time, state, control] = unpackZ(z, packInfo);
    [physicalTime, ...
     physicalState, ...
     physicalControl] = unpackZ(z, packInfo, true);

    % Initialize inequality and equality constraints
    c = [];
    ceq = [];

    % Evaluate defects constraints
    if isempty(defectConstraints)
        error('Defect constraints function is not provided');
    end
    if isempty(dynamics)
        error('Dynamics function is not provided');
    end
    % Evaluate derivatives
    derivatives = dynamics(physicalTime, ...
                           physicalState, ...
                           physicalControl);
    % Scale derivatives
    derivatives = derivatives ./ ...
                    packInfo.scaling.stateScaling * ...
                    packInfo.scaling.timeScaling;
    % Evaluate defects
    timeStep = (time(end) - time(1)) / packInfo.nGrid;
    defects = defectConstraints(timeStep, state, derivatives);
    ceq = [ceq; defects(:)];


    % Evaluate boundary constraints
    if ~isempty(boundaryConstraints)
        [boundaryIneq, ...
         boundaryEq] = boundaryConstraints(state(:, 1), ...
                                           state(:, end), ...
                                           time(1), ...
                                           time(end));
        c = [c; boundaryIneq];
```

```
        ceq = [ceq; boundaryEq];
    end

    % Evaluate path constraints
    if ~isempty(pathConstraints)
        [pathIneq, pathEq] = pathConstraints(time, state, control);
        c = [c; pathIneq];
        ceq = [ceq; pathEq];
    end
end
```

Código Fonte 6 – Função `evaluateConstraints`

A.1.6 `evaluateObjective.m`

```
function J = evaluateObjective(z, packInfo, ...
                               boundaryObjective, ...
                               pathObjective)
    % Evaluate the combined objective function
    % Inputs:
    %   z: [nz,1] = decision variables
    %   packInfo: struct = information about the problem
    %   boundaryObjective: function = Mayer term (phi)
    %   pathObjective: function = Lagrange term (L)
    %
    % Outputs:
    %   J = scalar = objective value

    % Unpack z
    [time, state, control] = unpackZ(z, packInfo);

    % Initialize objective
    J = 0;

    % Add Mayer term if provided
    if ~isempty(boundaryObjective)
        phi = boundaryObjective(state(:, 1), ...
                                state(:, end), ...
                                time(1), ...
```

```
                                    time(end));
        J = J + phi;
    end

    % Add Lagrange term if provided
    if ~isempty(pathObjective)
        % Trapezoidal integration of the path objective
        integrand = pathObjective(time, state, control);
        L = trapz(time, integrand);
        J = J + L;
    end
end
```

Código Fonte 7 – Função `evaluateObjective`

A.1.7 `spline2.m`

```
function [yInterp, dyInterp] = spline2(tOld, yOld, dyOld, tNew)
    % Second-order interpolation using the formula:
    % x(t) ~ xk + fk*tau + (tau^2/2hk)*(fk+1 - fk)
    % where tau is local time and hk is the time step
    %
    % Inputs:
    %   tOld: old time vector [1, nOld]
    %   yOld: old state matrix [nStates, nOld]
    %   dyOld: old derivative matrix [nStates, nOld]
    %   tNew: new time vector [1, nNew]
    %
    % Outputs:
    %   yInterp: interpolated state matrix [nStates, nNew]
    %   dyInterp: interpolated derivative matrix [nStates, nNew]

    nStates = size(yOld, 1);
    yInterp = zeros(nStates, length(tNew));
    dyInterp = zeros(nStates, length(tNew));

    for i = 1:length(tNew)
        t = tNew(i);

```

```
        % Find the interval containing t
        idx = find(tOld(1:end - 1) <= t & t <= tOld(2:end), 1);

        if isempty(idx)
            if t < tOld(1)
                idx = 1;
            else
                idx = length(tOld) - 1;
            end
        end

        % Get local time step
        hk = tOld(idx + 1) - tOld(idx);

        % Local time tau
        tau = t - tOld(idx);

        % Get values and derivatives (now matrices)
        xk = yOld(:, idx);
        fk = dyOld(:, idx);
        fk1 = dyOld(:, idx + 1);

        % Apply the formula (element-wise operations)
        yInterp(:, i) = xk + fk * tau + ...
                        (tau^2 / (2 * hk)) * (fk1 - fk);
        dyInterp(:, i) = fk + (tau / hk) * (fk1 - fk);
    end
end
```

Código Fonte 8 – Função `spline2`

A.2 Arquivos de modelo

A.2.1 `mainTemplate.m`

```
% Main script for template trajectory optimization problem
clear; clc; close all;

```

```
% Add path to the TrajectoryProblem class
addpath('..');

%% Create problem instance
% TODO: Set the number of states and controls for your problem
nx = 2;  % Number of states
nu = 1;  % Number of controls
problem = TrajectoryProblem(nx, nu);

%% Set time bounds
t0Low = 0;
t0Upp = 0;
tFLow = 0;
tFUpp = 1;
problem.setTimeBounds(t0Low, t0Upp, tFLow, tFUpp);

%% Set time boundary conditions
t0 = 0;
tF = 1;
problem.setTimeBoundaryConditions(t0, tF);

%% Set state bounds
% TODO: Define your state bounds
xLow = [-inf; -inf];  % Lower bounds for each state
xUpp = [inf; inf];    % Upper bounds for each state
problem.setStateBounds(xLow, xUpp);

%% Set control bounds
% TODO: Define your control bounds
uLow = [-inf];  % Lower bounds for each control
uUpp = [inf];   % Upper bounds for each control
problem.setControlBounds(uLow, uUpp);

%% Set parameters
% TODO: Create and set your problem parameters
params = templateParams();
problem.setParameters(params);

%% Set boundary conditions
% TODO: Define your boundary conditions
```

```
x0 = [0; 0];  % Initial state
xF = [1; 0];  % Final state
problem.setBoundaryConditions(x0, xF);

%% Set functions
% TODO: Set your dynamics function
problem.setDynamics(@templateDynamics);

% TODO: Set your objective function
problem.setObjective(@boundaryObjective, @pathObjective);

% TODO: Set your constraints function
problem.setConstraints(@boundaryConstraints, @pathConstraints);

%% Set solver options
% TODO: Adjust grid size and solver options as needed
nGrid = [50, 100, 200];  % Number of grid points for each iteration
options = optimoptions('fmincon');
options.Display = 'iter';
options.MaxFunEvals = 1e5;
problem.setSolverOptions(options, nGrid);

%% Optional: Set constraint checking function
% TODO: Set your constraint checking function
problem.setConstraintsCheck(@checkConstraints);

%% Optional: Set variable names
% TODO: Set meaningful names for your states and controls
problem.setVariableNames({'x1', 'x2'}, {'u1'});

%% Validate problem definition
problem.validate();

%% Solve the problem
solution = problem.solveWithTrapezoidalCollocation();

%% Save the solution
mkdir('results');
save('results/solution.mat', 'solution');

```

```
%% Plot results
plotResults('Solution', solution(end).z, true);
```

Código Fonte 9 – Arquivo principal do template

A.2.2 templateDynamics.m

```
function dx = templateDynamics(time, state, control, params)
    % Define the dynamics for your problem
    % Inputs:
    %   time: [1,n] time vector
    %   state: [nx,n] state matrix
    %   control: [nu,n] control matrix
    %   params: parameter struct
    % Outputs:
    %   dx: [nx,n] state derivative matrix

    arguments
        time double
        state double
        control double
        params struct
    end

    % TODO: Implement your system dynamics
    % Example for a double integrator:
    dx = [
        state(2,:);                 % dx1/dt = x2
        control(1,:)                % dx2/dt = u1
    ];
end
```

Código Fonte 10 – Função templateDynamics

A.2.3 templateParams.m

```
function params = templateParams()
    % Returns the problem parameters
```

```
    % TODO: Define your problem parameters here

    % Example parameters
    params.param1 = 1.0;
    params.param2 = 2.0;
end
```

Código Fonte 11 – Função `templateParams`

A.2.4 boundaryConstraints.m

```
function [c, ceq] = boundaryConstraints(x0, xF, t0, tF, boundaryConditions)
    % Evaluates boundary constraints
    % Inputs:
    %   x0: [nx,1] initial state vector
    %   xF: [nx,1] final state vector
    %   t0: initial time
    %   tF: final time
    %   boundaryConditions: struct with boundary conditions
    % Outputs:
    %   c: inequality constraints at boundaries
    %   ceq: equality constraints at boundaries

    arguments
        x0 double
        xF double
        t0 double
        tF double
        boundaryConditions struct
    end

    % TODO: Implement your boundary constraints
    c = [];
    ceq = [];
end
```

Código Fonte 12 – Função `boundaryConstraints`

A.2.5 pathConstraints.m

```
function [c, ceq] = pathConstraints(time, state, control, params)
    % Evaluates path constraints
    % Inputs:
    %   time: [1,n] time vector
    %   state: [nx,n] state matrix
    %   control: [nu,n] control matrix
    %   params: parameter struct
    % Outputs:
    %   c: inequality constraints
    %   ceq: equality constraints

    arguments
        time double
        state double
        control double
        params struct
    end

    % TODO: Implement your path constraints
    c = [];
    ceq = [];
end
```

Código Fonte 13 – Função pathConstraints

A.2.6 boundaryObjective.m

```
function cost = boundaryObjective(x0, xF, t0, tF, params)
    % Evaluates boundary cost (Mayer term)
    % Inputs:
    %   x0: [nx,1] initial state vector
    %   xF: [nx,1] final state vector
    %   t0: initial time
    %   tF: final time
    %   params: parameter struct
    % Outputs:
    %   cost: scalar cost value
```

```

    arguments
        x0 double
        xF double
        t0 double
        tF double
        params struct
    end

    % TODO: Implement your boundary objective
    cost = 0;
end
```

Código Fonte 14 – Função `boundaryObjective`

A.2.7 `pathObjective.m`

```
function L = pathObjective(time, state, control, params)
    % Evaluates running cost (Lagrange term)
    % Inputs:
    %   time: [1,n] time vector
    %   state: [nx,n] state matrix
    %   control: [nu,n] control matrix
    %   params: parameter struct
    % Outputs:
    %   L: instantaneous cost value

    arguments
        time double
        state double
        control double
        params struct
    end

    % TODO: Implement your path objective
    L = 0;
end
```

Código Fonte 15 – Função `pathObjective`

FOLHA DE REGISTRO DO DOCUMENTO

1. CLASSIFICAÇÃO/TIPO	2. DATA	3. DOCUMENTO Nº	4. Nº DE PÁGINAS
TC	¿¿ de novembro de 2024	DCTA/ITA/DM-¿¿¿/2024	78

5. TÍTULO E SUBTÍTULO:

Implementação do método de colocação direta para otimização de trajetória usando o software MATLAB

6. AUTOR(ES):

Henrique Silva Simplicio

7. INSTITUIÇÃO(ÕES)/ÓRGÃO(S) INTERNO(S)/DIVISÃO(ÕES):

Instituto Tecnológico de Aeronáutica – ITA

8. PALAVRAS-CHAVE SUGERIDAS PELO AUTOR:

Controle Ótimo; Otimização de Trajetória; Colocação Direta

9. PALAVRAS-CHAVE RESULTANTES DE INDEXAÇÃO:

¿; ¿; ¿

10. APRESENTAÇÃO: **(X) Nacional** **( ) Internacional**

Trabalho de Graduação, ITA, São José dos Campos, 2024. 78 páginas.

11. RESUMO:

Este trabalho apresenta o desenvolvimento de uma biblioteca em MATLAB para solução de problemas de otimização de trajetória utilizando o método de colocação direta trapezoidal. A biblioteca visa simplificar a resolução deste tipo de problema, permitindo que o usuário concentre-se apenas na modelagem, sem necessidade de implementar a lógica matemática da solução numérica. A implementação é validada através de três casos de teste: um problema de movimento unidimensional simples, o problema clássico da braquistócrona e um problema de otimização de trajetória de subida de uma aeronave eVTOL. Os resultados obtidos são comparados com soluções analíticas, quando disponíveis, e com resultados da literatura utilizando outros softwares de otimização. O trabalho contribui para a área de controle ótimo ao disponibilizar uma ferramenta que facilita a implementação de soluções numéricas para problemas de otimização de trajetória, sendo particularmente útil para aplicações em engenharia aeroespacial e áreas correlatas.

12. GRAU DE SIGILO:

(X) OSTENSIVO **( ) RESERVADO** **( ) SECRETO**