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.
123f.

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. 123f. 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

É concedida ao Instituto Tecnológico de Aeronáutica permissão para reproduzir cópias deste trabalho de graduação e para emprestar ou vender cópias somente para propósitos acadêmicos e científicos. O autor reserva outros direitos de publicação e nenhuma parte deste trabalho de graduação pode ser reproduzida sem a autorização do autor.

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, 14 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 co-

locação 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

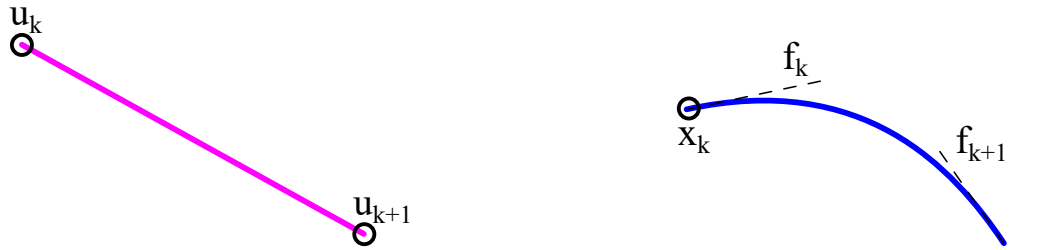
$$\mathbf{u}(t) \approx \mathbf{u}_k + \frac{\tau}{h_k}(\mathbf{u}_{k+1} - \mathbf{u}_k), \tag{2.12}$$

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

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

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

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

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

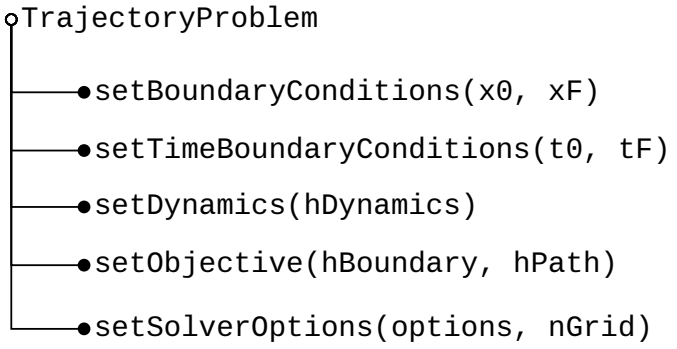
Diferente dos softwares OptimTraj (KELLY, 2022) e PSOPT (BECERRA, 2022), a implementação será realizada utilizando uma abordagem de POO, com uma classe MATLAB principal que conterá métodos para a configuração do problema e cálculo da solução. Além da classe principal, a biblioteca contará com funções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

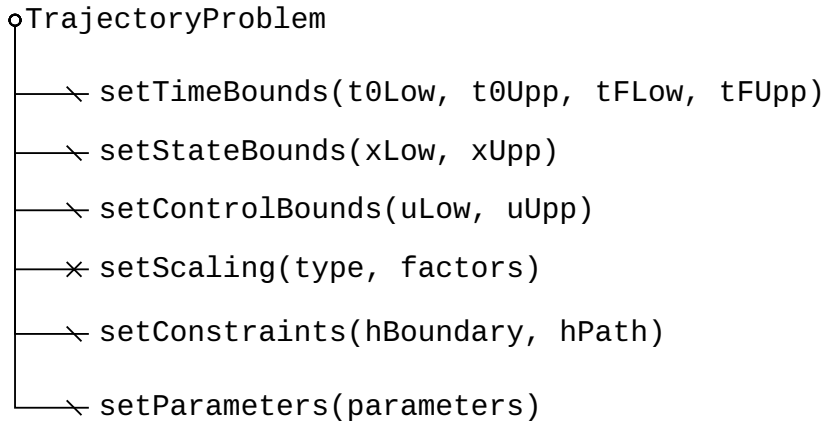
3.1.1 `TrajectoryProblem.m`

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.

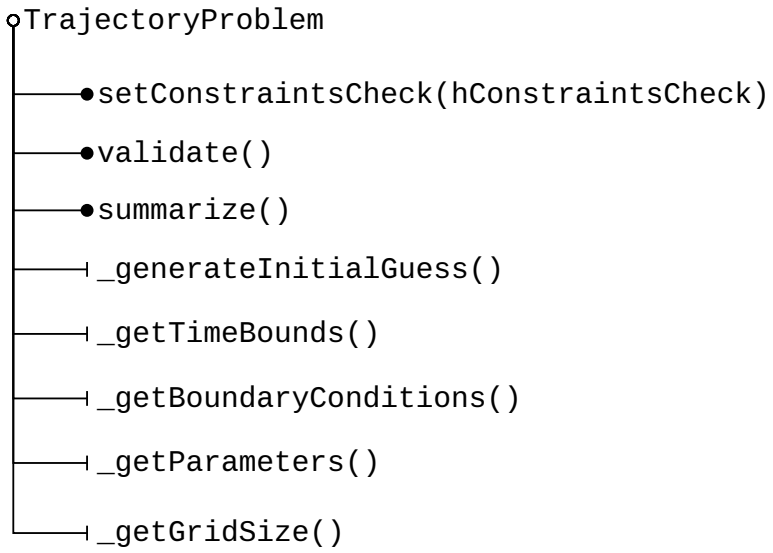


FIGURA 3.2 – Métodos auxiliares da classe `TrajectoryProblem`

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.

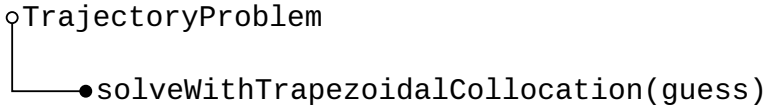


FIGURA 3.3 – Método para a solução do `TrajectoryProblem` com a colocação trapezoidal

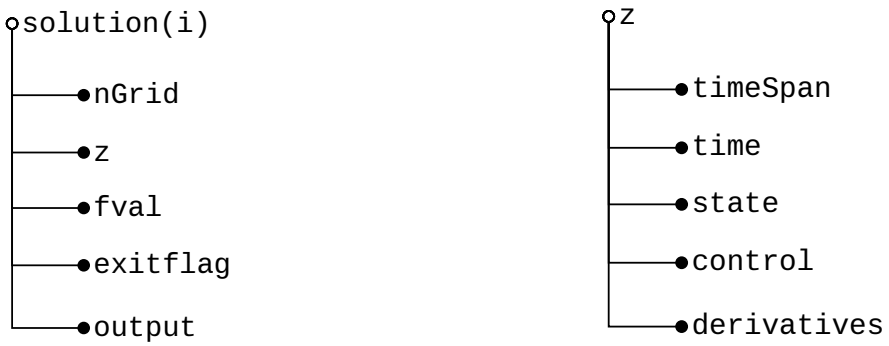


FIGURA 3.4 – Formato da `structure solution`

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
```

```
16
17  % Defina as funções necessárias
18  problem.setDynamics(@systemDynamics);  % Dinâmica do sistema
19  problem.setObjective(@boundaryObj, @pathObj);  % Funções objetivo
20  problem.setConstraints(@boundaryConst, @pathConst);  % Restrições opcionais
21
22  % Defina as opções do solver
23  options = optimoptions('fmincon', 'Display', 'iter');
24  nGrid = [50, 100, 200];  % Várias iterações com refinamento de malha
25  problem.setSolverOptions(options, nGrid);
26
27  % Resolva o problema
28  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 $\mathbf{F}$, 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.

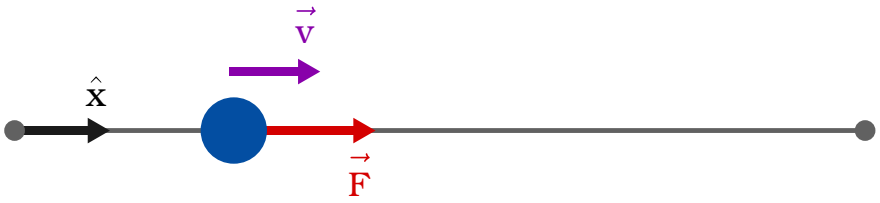


FIGURA 3.5 – Diagrama de forças no movimento simples

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

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

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.

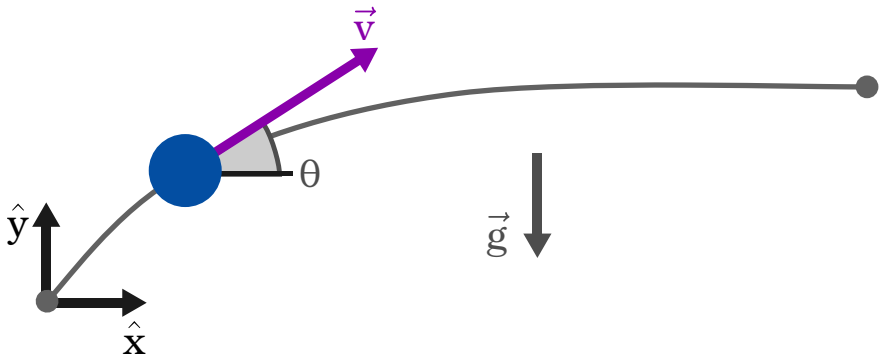


FIGURA 3.6 – Diagrama de forças no braquistócrona

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

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

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.

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

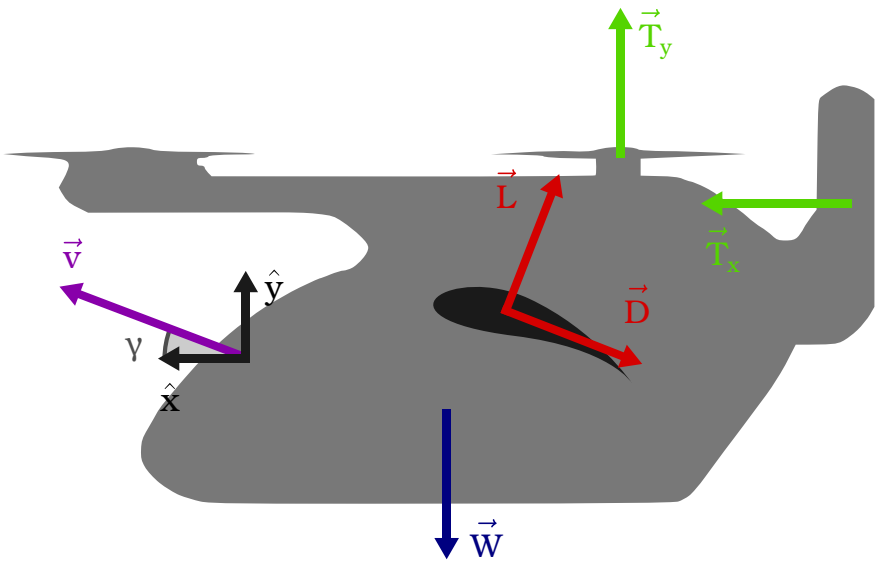


FIGURA 3.7 – Diagrama de forças no eVTOL

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

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

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.

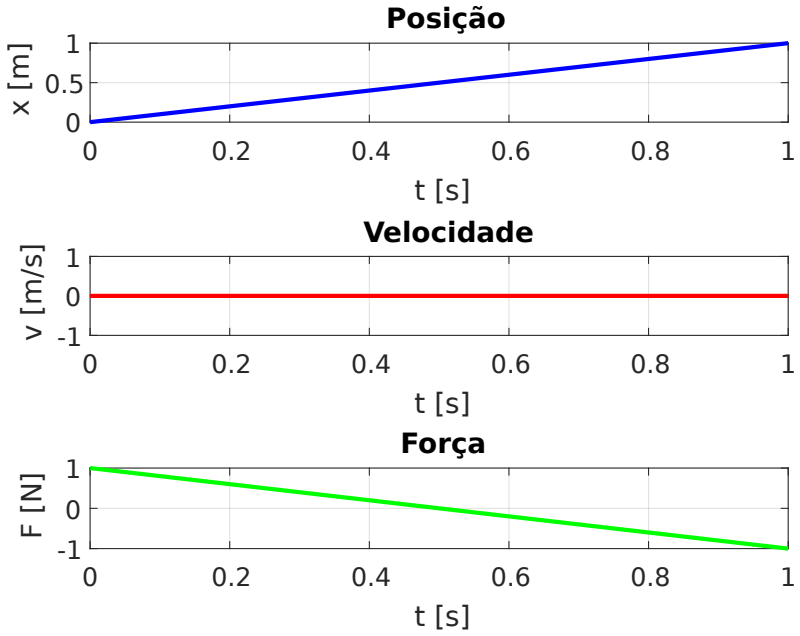


FIGURA 4.1 – Chute inicial para o problema de movimento simples em uma dimensão.

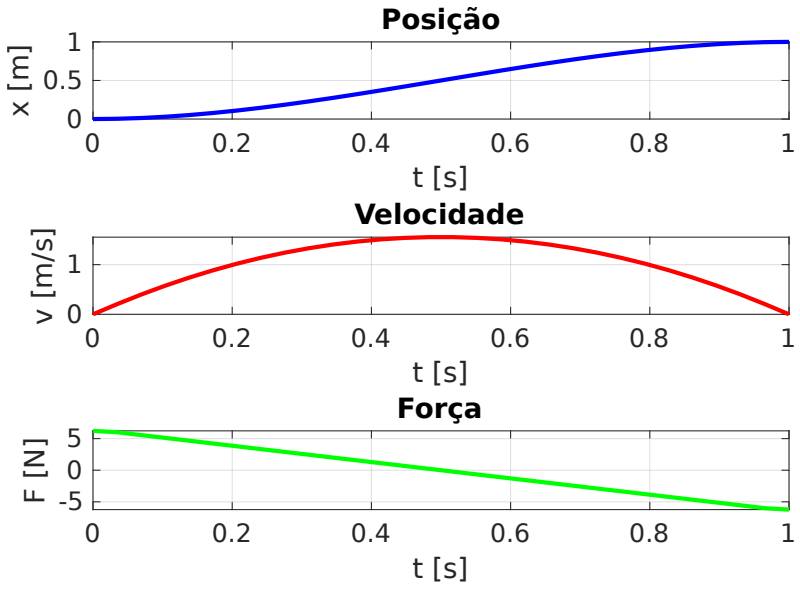


FIGURA 4.2 – Trajetória ótima do problema de movimento simples em uma dimensão.

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.

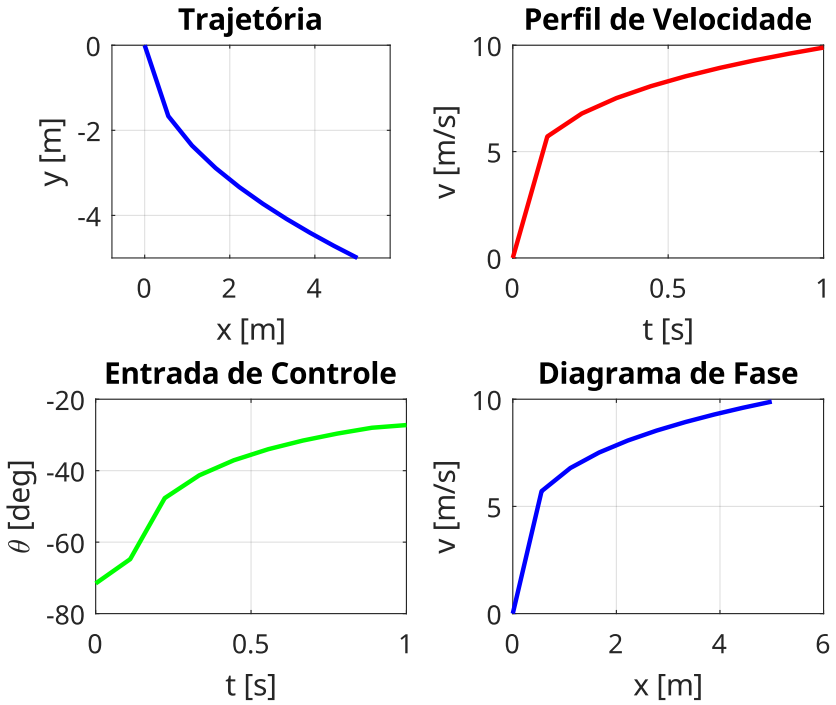


FIGURA 4.3 – Chute inicial para o problema de braquistócrona.

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.

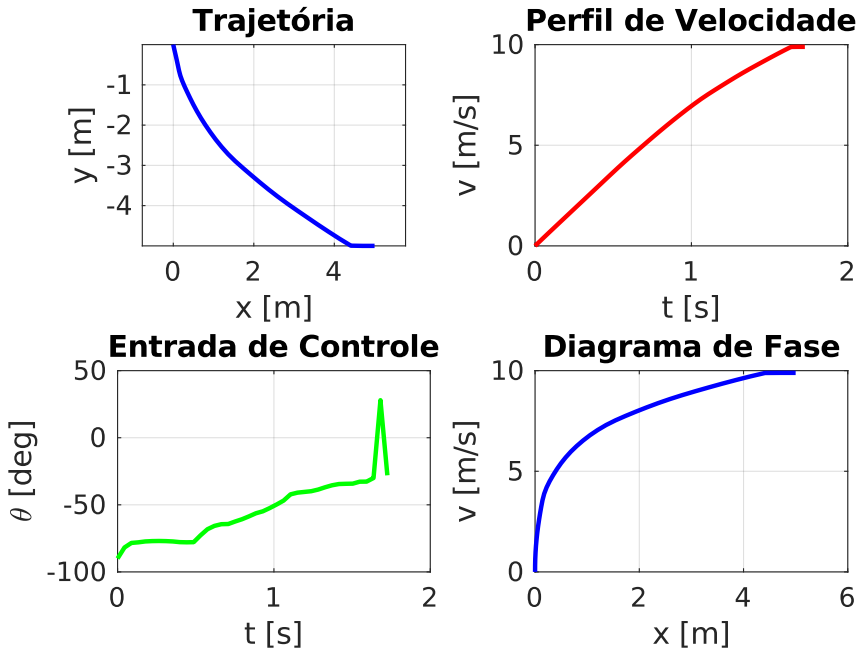


FIGURA 4.4 – Trajetória ótima do problema de braquistócrona.

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

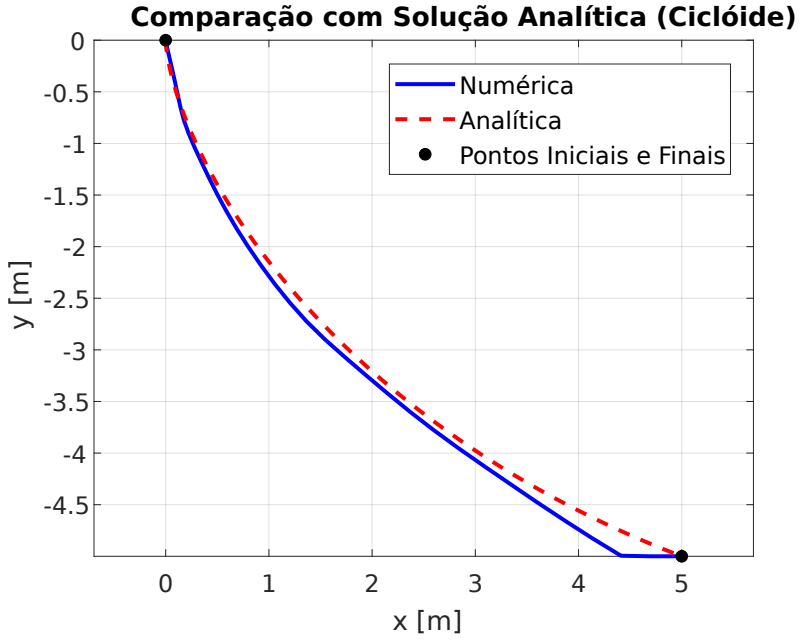


FIGURA 4.5 – Comparação entre a solução obtida e a solução analítica.

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 ini-

cial, 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.

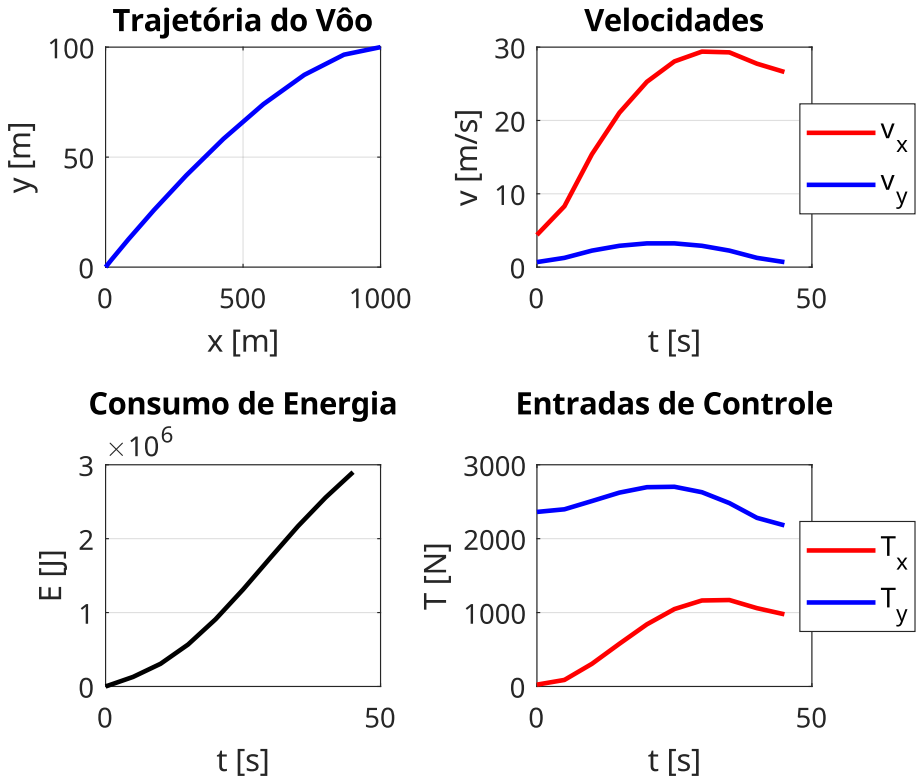


FIGURA 4.6 – Chute inicial para o problema de trajetória do eVTOL.

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.

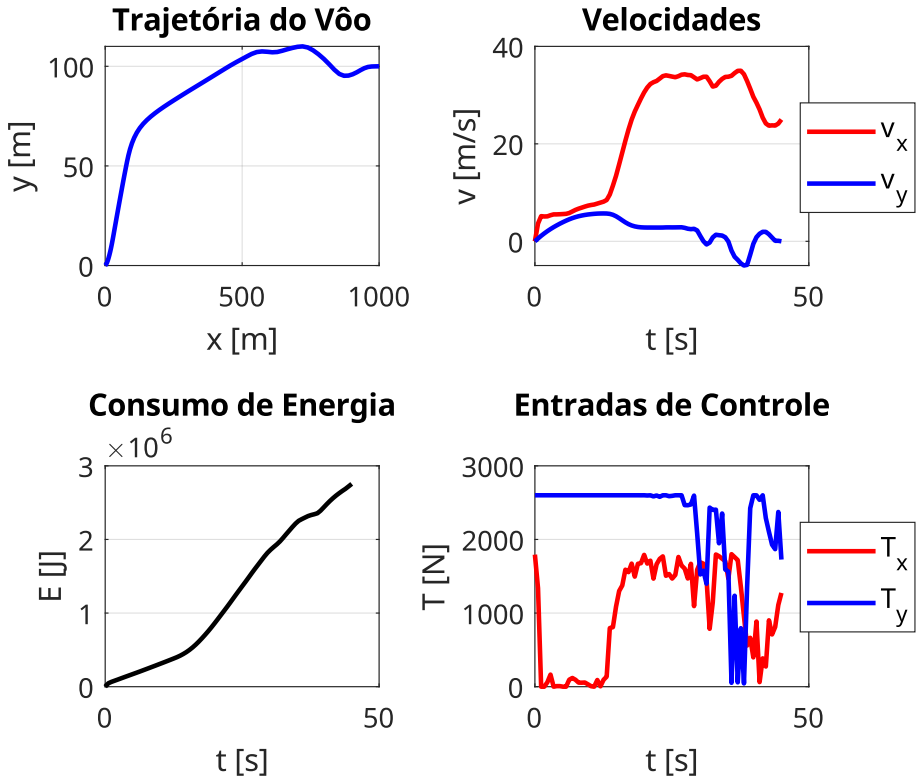


FIGURA 4.7 – Trajetória ótima do problema de trajetória do eVTOL.

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.

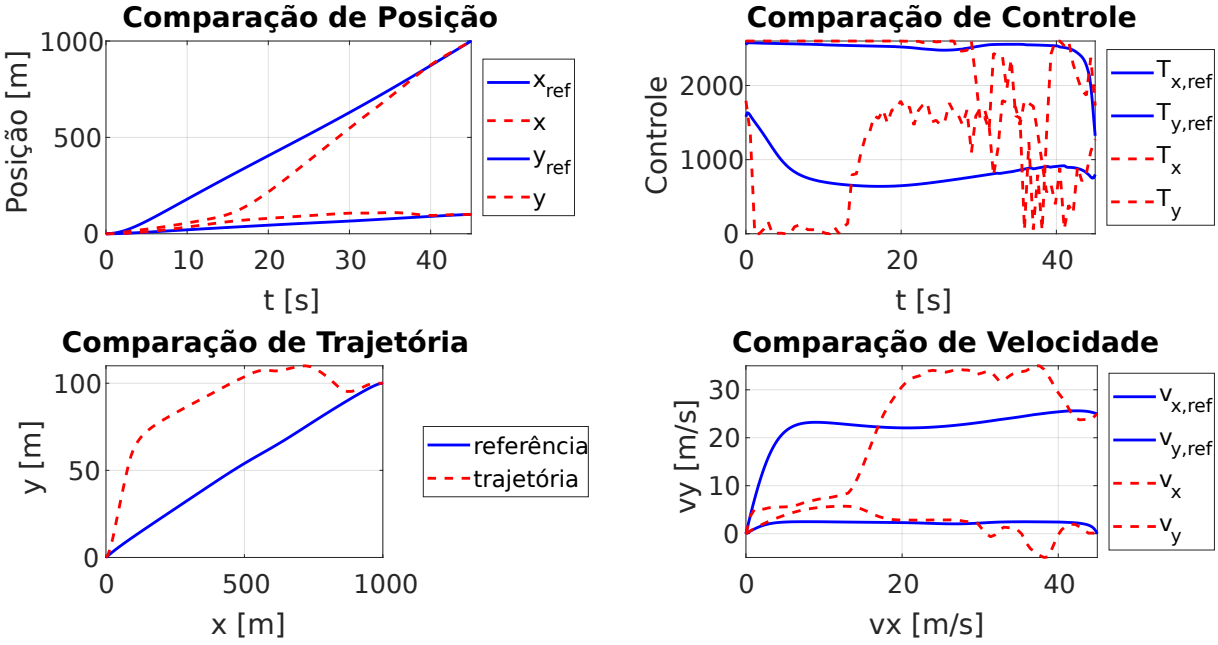


FIGURA 4.8 – Comparação entre a solução obtida e a solução de referência.

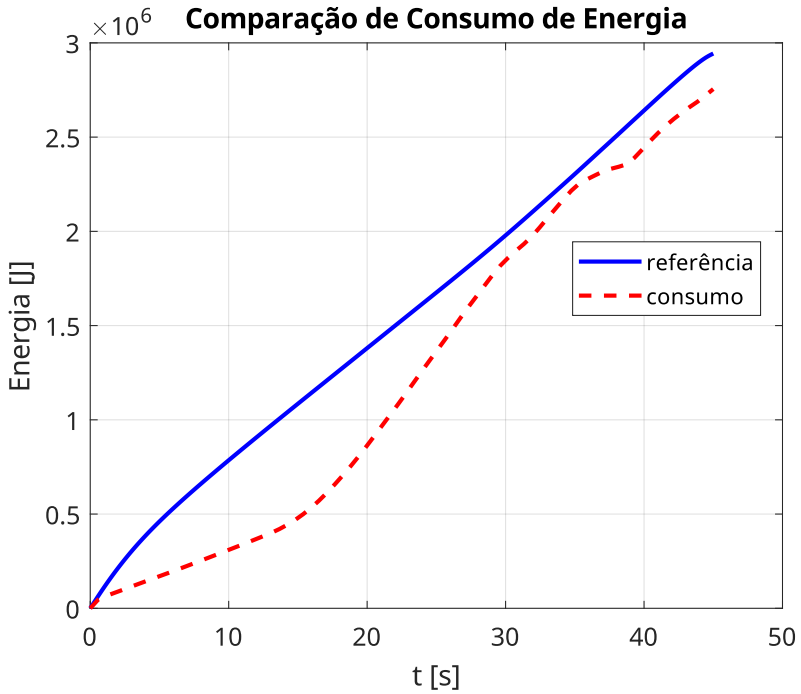


FIGURA 4.9 – Comparação da energia.

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. 18

BECERRA, V. **PSOPT Optimal Control Solver: User Manual**. 2022. Disponível em: <https://github.com/PSOPT/psopt/releases/tag/5.02>. 17, 18, 19, 21, 26

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>. 18

BELLMAN, R. **Dynamic Programming**. [S.l.]: Princeton University Press, 2010. Google-Books-ID: 92aYDwAAQBAJ. ISBN 978-0-691-14668-3. 18

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>. 19

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. 18, 19, 20, 23, 46

BRYSON, A. E. **Applied Optimal Control: Optimization, Estimation and Control**. [S.l.]: Routledge, 2018. Google-Books-ID: LFUPEAAAQBAJ. ISBN 978-1-351-46592-2. 18

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. 26, 37

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>. 19, 21, 23, 46

KELLY, M. P. **OptimTraj: Trajectory Optimization for Matlab**. 2022. Disponível em: <https://github.com/MatthewPeterKelly/OptimTraj>. 17, 18, 19, 26

KIRK, D. E. **Optimal Control Theory: An Introduction**. [S.l.]: Courier Corporation, 2004. Google-Books-ID: fCh2SAtWIdwC. ISBN 978-0-486-43484-1. 18

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>. 18

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>. 18

LEWIS, F. L. *et al.* **Optimal Control**. [S.l.]: John Wiley & Sons, 2012. Google-Books-ID: U3Gtlot_hYEC. ISBN 978-1-118-12272-3. 18, 19

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>. 19

PONTRYAGIN, L. S. **Mathematical Theory of Optimal Processes**. [S.l.]: CRC Press, 1987. Google-Books-ID: kwzq0F4cBVAC. ISBN 978-2-88124-077-5. 18

SIMPLICIO, H. **hsimplicio/tg-ita**. 2024. Disponível em: <https://github.com/hsimplicio/tg-ita>. 26, 49

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>. 18

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
```

```
62      end
63
64      properties
65          % Optional properties
66          description = ''  % Problem description
67          stateNames = {}  % Names of state variables
68          controlNames = {}  % Names of control variables
69      end
70
71      methods
72
73          function obj = TrajectoryProblem(nx, nu)
74              % Constructor
75              if nargin < 2
76                  error('Must specify number of states and controls');
77              end
78
79              obj.nx = nx;
80              obj.nu = nu;
81
82              % Initialize state and control bounds
83              obj.xLow = -1e8 * ones(nx, 1);
84              obj.xUpp = 1e8 * ones(nx, 1);
85              obj.uLow = -1e8 * ones(nu, 1);
86              obj.uUpp = 1e8 * ones(nu, 1);
87
88              % Initialize time bounds
89              obj.t0Low = -1e8;
90              obj.t0Upp = 1e8;
91              obj.tFLow = -1e8;
92              obj.tFUpp = 1e8;
93
94              % Default time boundary conditions
95              obj.t0 = 0;
96              obj.tF = 1;
97              obj.timeSpan{1} = [obj.t0, obj.tF];
98
99              % Initialize boundary conditions with empty arrays
100             obj.x0 = zeros(nx, 1);
101             obj.xF = zeros(nx, 1);
```

```
102
103             % Initialize scaling factors to 1
104             obj.stateScaling = ones(nx, 1);
105             obj.controlScaling = ones(nu, 1);
106             obj.timeScaling = 1;
107             obj.scaling = struct('stateScaling', obj.stateScaling, ...
108                                  'controlScaling', obj.controlScaling, ...
109                                  'timeScaling', obj.timeScaling);
110
111             % Initialize boundary conditions struct
112             obj.boundaryConditions = struct();
113             obj.boundaryConditions.t0 = obj.t0;
114             obj.boundaryConditions.tF = obj.tF;
115             obj.boundaryConditions.x0 = obj.x0;
116             obj.boundaryConditions.xF = obj.xF;
117         end
118
119         function setBoundaryConditions(obj, x0, xF)
120             % Set boundary conditions
121             validateattributes(x0, {'numeric'}, ...
122                                {'vector', 'numel', obj.nx});
123             validateattributes(xF, {'numeric'}, ...
124                                {'vector', 'numel', obj.nx});
125             obj.x0 = x0(:);  % Ensure column vector
126             obj.xF = xF(:);
127             obj.boundaryConditions.x0 = x0;
128             obj.boundaryConditions.xF = xF;
129         end
130
131         function setTimeBoundaryConditions(obj, t0, tF)
132             % Set time bounds
133             validateattributes(t0, {'numeric'}, {'scalar'});
134             validateattributes(tF, {'numeric'}, {'scalar', '>', t0});
135             obj.t0 = t0;
136             obj.tF = tF;
137             obj.timeSpan{1} = [obj.t0, obj.tF];
138             obj.boundaryConditions.t0 = t0;
139             obj.boundaryConditions.tF = tF;
140         end
141
```

```
142         function setTimeBounds(obj, t0Low, t0Upp, tFLow, tFUpp)
143             % Set bounds for initial and final times
144             validateattributes(t0Low, {'numeric'}, {'scalar'});
145             validateattributes(t0Upp, {'numeric'}, {'scalar'});
146             validateattributes(tFLow, {'numeric'}, {'scalar'});
147             validateattributes(tFUpp, {'numeric'}, {'scalar'});
148 
149             assert(t0Low <= t0Upp, ...
150                    'Lower bound for t0 must be <= upper bound');
151             assert(tFLow <= tFUpp, ...
152                    'Lower bound for tF must be <= upper bound');
153             assert(t0Upp <= tFLow, ...
154                    'Upper bound for t0 must be <= lower bound for tF');
155 
156             obj.t0Low = t0Low;
157             obj.t0Upp = t0Upp;
158             obj.tFLow = tFLow;
159             obj.tFUpp = tFUpp;
160         end
161 
162         function setStateBounds(obj, xLow, xUpp)
163             % Set state bounds
164             validateattributes(xLow, {'numeric'}, ...
165                                {'vector', 'numel', obj.nx});
166             validateattributes(xUpp, {'numeric'}, ...
167                                {'vector', 'numel', obj.nx});
168             assert(all(xLow <= xUpp), ...
169                    'Lower bounds must be <= upper bounds');
170             obj.xLow = xLow(:);
171             obj.xUpp = xUpp(:);
172         end
173 
174         function setControlBounds(obj, uLow, uUpp)
175             % Set control bounds
176             validateattributes(uLow, {'numeric'}, ...
177                                {'vector', 'numel', obj.nu});
178             validateattributes(uUpp, {'numeric'}, ...
179                                {'vector', 'numel', obj.nu});
180             assert(all(uLow <= uUpp), ...
181                    'Lower bounds must be <= upper bounds');
```

```
182             obj.uLow = uLow(:);
183             obj.uUpp = uUpp(:);
184         end
185 
186         function setScaling(obj, type, factors)
187             % Set scaling factors for variables
188             % type: 'state', 'control', or 'time'
189             % factors: vector of scaling factors
190 
191             switch lower(type)
192                 case 'state'
193                     validateattributes(factors, {'numeric'}, ...
194                                        {'vector', 'positive', ...
195                                        'numel', obj.nx});
196                     obj.stateScaling = factors(:);
197                     obj.scaling.stateScaling = factors(:);
198                 case 'control'
199                     validateattributes(factors, {'numeric'}, ...
200                                        {'vector', 'positive', ...
201                                        'numel', obj.nu});
202                     obj.controlScaling = factors(:);
203                     obj.scaling.controlScaling = factors(:);
204                 case 'time'
205                     validateattributes(factors, {'numeric'}, ...
206                                        {'scalar', 'positive'});
207                     obj.timeScaling = factors;
208                     obj.scaling.timeScaling = factors;
209                 otherwise
210                     error(['Invalid scaling type. Must be', ...
211                            '''state'', ''control'', or ''time''']);
212             end
213         end
214 
215         function setDynamics(obj, hDynamics)
216             % Set dynamics function
217             validateattributes(hDynamics, {'function_handle'}, {});
218             % Test the function with dummy inputs
219             time_test = zeros(1, 1);
220             x_test = zeros(obj.nx, 1);
221             u_test = zeros(obj.nu, 1);
```

```
222             try
223                 dx = hDynamics(time_test, x_test, ...
224                                u_test, obj.parameters);
225                 validateattributes(dx, {'numeric'}, ...
226                                    {'vector', 'numel', obj.nx});
227             catch ME
228                 error('Invalid dynamics function: %s', ME.message);
229             end
230             obj.dynamics = @(time, state, control) ...
231                            hDynamics(time, state, ...
232                                      control, obj.parameters);
233         end
234 
235         function setObjective(obj, hBoundaryObjective, hPathObjective)
236             % Set objective functions
237             % Both inputs are optional - pass [] to skip
238 
239             % Validate boundary objective if provided
240             if ~isempty(hBoundaryObjective)
241                 validateattributes(hBoundaryObjective, ...
242                                    {'function_handle'}, {});
243                 try
244                     val = hBoundaryObjective(zeros(obj.nx, 1), ...
245                                              zeros(obj.nx, 1), ...
246                                              obj.t0, obj.tF, ...
247                                              obj.parameters);
248                     validateattributes(val, {'numeric'}, {'scalar'});
249                 catch ME
250                     error('Invalid boundary objective function: %s', ...
251                           ME.message);
252                 end
253                 obj.boundaryObjective = @(x0, xF, t0, tF) ...
254                                         hBoundaryObjective(x0, xF, ...
255                                                            t0, tF, ...
256                                                            obj.parameters);
257             else
258                 obj.boundaryObjective = [];
259             end
260 
261             % Validate path objective if provided
```

```
262             if ~isempty(hPathObjective)
263                 validateattributes(hPathObjective, ...
264                                    {'function_handle'}, {});
265                 try
266                     time_test = [obj.t0, obj.tF];
267                     val = hPathObjective(time_test, ...
268                                          zeros(obj.nx, 2), ...
269                                          zeros(obj.nu, 2), ...
270                                          obj.parameters);
271                     validateattributes(val, {'numeric'}, ...
272                                        {'size', [1, numel(time_test)]});
273                 catch ME
274                     error('Invalid path objective function: %s', ...
275                           ME.message);
276                 end
277                 obj.pathObjective = @(time, state, control) ...
278                                     hPathObjective(time, state, ...
279                                                    control, ...
280                                                    obj.parameters);
281             else
282                 obj.pathObjective = [];
283             end
284 
285             % Create combined objective function
286             obj.objective = @(z, packInfo) ...
287                             evaluateObjective(z, packInfo, ...
288                                               obj.boundaryObjective, ...
289                                               obj.pathObjective);
290         end
291 
292         function setConstraints(obj, hBoundaryConstraints, hPathConstraints)
293             % Set constraint functions
294             % Both inputs are optional - pass [] to skip
295 
296             % Validate boundary constraints if provided
297             if ~isempty(hBoundaryConstraints)
298                 validateattributes(hBoundaryConstraints, ...
299                                    {'function_handle'}, {});
300                 try
301                     [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 = [];
```

```
342             end
343
344             % Validate path constraints if provided
345             if ~isempty(hPathConstraints)
346                 validateattributes(hPathConstraints, ...
347                                    {'function_handle'}, {});
348                 try
349                     time_test = [obj.t0, obj.tF];
350                     [c, ceq] = hPathConstraints(time_test, ...
351                                                 zeros(obj.nx, 2), ...
352                                                 zeros(obj.nu, 2), ...
353                                                 obj.parameters);
354                     if ~isempty(c)
355                         validateattributes(c, {'numeric'}, ...
356                                            {'vector'});
357                     end
358                     if ~isempty(ceq)
359                         validateattributes(ceq, {'numeric'}, ...
360                                            {'vector'});
361                     end
362                 catch ME
363                     error('Invalid path constraints function: %s', ...
364                           ME.message);
365                 end
366                 obj.pathConstraints = @(time, state, control) ...
367                                       hPathConstraints(time, state, ...
368                                                        control, ...
369                                                        obj.parameters);
370             else
371                 obj.pathConstraints = [];
372             end
373
374             % Create combined constraints function
375             obj.constraints = @(z, packInfo) ...
376                               evaluateConstraints(z, packInfo, ...
377                                                   obj.dynamics, ...
378                                                   @computeDefects, ...
379                                                   obj.pathConstraints, ...
380                                                   obj.boundaryConstraints);
381         end
```

```
382
383         function setParameters(obj, parameters)
384             % Set problem parameters
385             obj.parameters = parameters;
386         end
387
388         function setVariableNames(obj, stateNames, controlNames)
389             % Set names for variables (optional)
390             if nargin > 1 && ~isempty(stateNames)
391                 validateattributes(stateNames, {'cell'}, ...
392                                    {'numel', obj.nx});
393                 obj.stateNames = stateNames;
394             end
395             if nargin > 2 && ~isempty(controlNames)
396                 validateattributes(controlNames, {'cell'}, ...
397                                    {'numel', obj.nu});
398                 obj.controlNames = controlNames;
399             end
400         end
401
402         function setSolverOptions(obj, options, nGrid)
403             % Set solver options and grid points
404             % options: either a single optimoptions('fmincon') object
405             %          or cell array of options for each iteration
406             % nGrid: array of grid points for each iteration
407
408             validateattributes(nGrid, {'numeric'}, ...
409                                {'vector', 'positive', 'integer'});
410
411             if ~iscell(options)
412                 % If single options provided,
413                 % replicate for all iterations
414                 validateattributes(options, ...
415                                    {'optim.options.Fmincon'}, {});
416                 obj.solverOptions = repmat({options}, 1, length(nGrid));
417             else
418                 % If cell array provided, validate length matches nGrid
419                 validateattributes(options, {'cell'}, ...
420                                    {'numel', length(nGrid)});
421                 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);
```

```
462             hasOptions = ~isempty(obj.solverOptions);
463             gridMatchesOptions = length(obj.nGrid) == ...
464                                  length(obj.solverOptions);
465 
466             % Combine all checks
467             valid = hasDynamics && hasObjective && hasFiniteBounds && ...
468                     hasGrid && hasOptions && gridMatchesOptions;
469 
470             % Provide detailed warning if invalid
471             if ~valid
472                 warning('TrajectoryProblem:Incomplete', ...
473                         ['Problem definition is incomplete.', ...
474                          ' Missing requirements:\n%s%s%s%s%s%s', ...
475                          conditional_msg(~hasDynamics, ...
476                                          '- Dynamics function not set'), ...
477                          conditional_msg(~hasObjective, ...
478                                          '- Objective function not set'), ...
479                          conditional_msg(~hasFiniteBounds, ...
480                                          '- Infinite bounds detected'), ...
481                          conditional_msg(~hasGrid, ...
482                                          '- Grid points not set'), ...
483                          conditional_msg(~hasOptions, ...
484                                          '- Solver options not set'), ...
485                          conditional_msg(~gridMatchesOptions, ...
486                                          ['- Number of grid points does', ...
487                                           ' not match number of solver options'])]
488             end
489 
490             function msg = conditional_msg(condition, message)
491                 if condition
492                     msg = message;
493                 else
494                     msg = '';
495                 end
496             end
497 
498         end
499 
500         function info = summarize(obj)
501             % 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);
```

```
542                     else
543                         % Use half of lower bound
544                         finalVal = min(obj.x0(i), ...
545                                        obj.xLow(i) / 2);
546                     end
547                     state(i, :) = interp1([obj.t0, obj.tF], ...
548                                           [obj.x0(i), finalVal], ...
549                                           timeGrid);
550                 else
551                     % Normal linear interpolation
552                     % for fixed final states
553                     state(i, :) = interp1([obj.t0, obj.tF], ...
554                                           [obj.x0(i), obj.xF(i)], ...
555                                           timeGrid);
556                 end
557             end
558
559             % Initialize controls to zeros
560             control = zeros(obj.nu, obj.nGrid(1));
561         end
562
563         function solution = solveWithTrapezoidalCollocation(obj, guess)
564             % Solve the trajectory optimization problem
565             % with trapezoidal collocation
566             if ~obj.validate()
567                 error('TrajectoryProblem:InvalidProblem', ...
568                       'Problem is not completely defined');
569             end
570
571             nIter = length(obj.nGrid);
572             solution(nIter) = struct();
573
574             for i = 1:nIter
575                 disp(['Iteration ', num2str(i), ' of ', ...
576                       num2str(nIter)]);
577
578                 % Generate or interpolate initial guess
579                 if i == 1
580                     timeGrid = linspace(obj.t0, obj.tF, obj.nGrid(1));
581                     if nargin < 2 || isempty(guess)
```

```
582                         [stateGuess, controlGuess] = ...
583                             obj.generateInitialGuess();
584                         [zGuess, packInfo] = packZ([obj.t0, obj.tF], ...
585                                                    stateGuess, ...
586                                                    controlGuess, ...
587                                                    obj.scaling);
588                     else
589                         [zGuess, ...
590                          packInfo] = packZ([obj.t0, obj.tF], ...
591                                            guess(1:obj.nx, :), ...
592                                            guess(obj.nx + 1:end, :), ...
593                                            obj.scaling);
594                     end
595                 else
596                     % Validate time span
597                     assert(all(isfinite(obj.timeSpan{i - 1})), ...
598                            'Non-finite values in previous time span');
599                     assert(all(isfinite(obj.timeSpan{i})), ...
600                            'Non-finite values in current time span');
601                     assert(obj.timeSpan{i - 1}(2) >= ...
602                            obj.timeSpan{i - 1}(1), ...
603                            'Invalid previous time span');
604                     assert(obj.timeSpan{i}(2) >= obj.timeSpan{i}(1), ...
605                            'Invalid current time span');
606
607                     timeGrid = linspace(obj.timeSpan{i}(1), ...
608                                         obj.timeSpan{i}(2), ...
609                                         obj.nGrid(i));
610                     timeOld = linspace(obj.timeSpan{i - 1}(1), ...
611                                        obj.timeSpan{i - 1}(2), ...
612                                        obj.nGrid(i - 1));
613
614                     % Validate previous solution
615                     assert(all(isfinite(solution(i - 1).z.state(:))), ...
616                            'Non-finite values in previous state');
617                     assert(all(isfinite(solution(i - 1).z.control(:))), ...
618                            'Non-finite values in previous control');
619
620                     % Perform interpolation with validation
621                     stateGuess = zeros(obj.nx, obj.nGrid(i));
```

```
622                     controlGuess = zeros(obj.nu, obj.nGrid(i));
623
624                     % Interpolate each state and control
625                     % separately to maintain dimensions
626                     for j = 1:obj.nx
627                         [stateGuess(j, :), ~] = ...
628                             spline2(timeOld, ...
629                                     solution(i - 1).z.state(j, :), ...
630                                     solution(i - 1).z.derivatives(j, :), ...
631                                     timeGrid);
632                     end
633                     for j = 1:obj.nu
634                         controlGuess(j, :) = ...
635                             interp1(timeOld, ...
636                                     solution(i - 1).z.control(j, :), ...
637                                     timeGrid, ...
638                                     'linear', 'extrap');
639                     end
640
641                     % Validate interpolation results
642                     assert(all(isfinite(stateGuess(:))), ...
643                            'Non-finite values in interpolated state');
644                     assert(all(isfinite(controlGuess(:))), ...
645                            'Non-finite values in interpolated control');
646
647                     % Pack with validation
648                     [zGuess, packInfo] = packZ(obj.timeSpan{i}, ...
649                                                stateGuess, ...
650                                                controlGuess, ...
651                                                obj.scaling);
652                     assert(all(isfinite(zGuess)), ...
653                            'Non-finite values in packed guess');
654                 end
655
656                 % Set up the problem
657                 fun = @(z)(obj.objective(z, packInfo));
658                 A = [];
659                 b = [];
660                 Aeq = [];
661                 beq = [];
```

```
662                 lb = [
663                       obj.t0Low / obj.scaling.timeScaling
664                       obj.tFLow / obj.scaling.timeScaling
665                       reshape(repmat(obj.xLow ./ ...
666                                      obj.scaling.stateScaling, 1, ...
667                                      obj.nGrid(i)), [], 1)
668                       reshape(repmat(obj.uLow ./ ...
669                                      obj.scaling.controlScaling, 1, ...
670                                      obj.nGrid(i)), [], 1)
671                      ];
672                 ub = [
673                       obj.t0Upp / obj.scaling.timeScaling
674                       obj.tFUpp / obj.scaling.timeScaling
675                       reshape(repmat(obj.xUpp ./ ...
676                                      obj.scaling.stateScaling, 1, ...
677                                      obj.nGrid(i)), [], 1)
678                       reshape(repmat(obj.uUpp ./ ...
679                                      obj.scaling.controlScaling, 1, ...
680                                      obj.nGrid(i)), [], 1)
681                      ];
682                 nonlcon = @(z)(obj.constraints(z, packInfo));
683
684                 % Solve scaled problem
685                 [z_scaled, ...
686                  fval, ...
687                  exitflag, ...
688                  output] = fmincon(fun, zGuess, ...
689                                    A, b, ...
690                                    Aeq, beq, ...
691                                    lb, ub, ...
692                                    nonlcon, ...
693                                    obj.solverOptions{i});
694
695                 % Unpack solution with scaling up
696                 [time, state, control] = unpackZ(z_scaled, ...
697                                                  packInfo, true);
698
699                 % Set time span for next iteration
700                 obj.timeSpan{i + 1} = [time(1), time(end)];
701
```

```
                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

```

```
742         function nGrid = getGridSize(obj)
743             % Get grid size
744             nGrid = obj.nGrid;
745         end
746 
747     end
748 end
```

Código Fonte 2 – Classe `TrajectoryProblem`

A.1.2 packZ.m

```
1  function [z, packInfo] = packZ(tSpan, state, control, scaling)
2      % Pack time bounds and trajectories into optimization vector
3      % Inputs:
4      %   tSpan: [1,2] = time span
5      %   state: [nx,nGrid] = state trajectory
6      %   control: [nu,nGrid] = control trajectory
7      %   scaling: struct = scaling factors
8      % Outputs:
9      %   z: [nz,1] = decision variables
10     %   packInfo: struct = information about the problem
11 
12     % Pack packInfo
13     packInfo = struct();
14     packInfo.nx = size(state, 1);
15     packInfo.nu = size(control, 1);
16     packInfo.nGrid = size(state, 2);
17     packInfo.scaling = scaling;
18 
19     % Scale variables
20     tScaled = tSpan / scaling.timeScaling;
21     stateScaled = state ./ repmat(scaling.stateScaling, 1, ...
22                                   size(state, 2));
23     controlScaled = control ./ repmat(scaling.controlScaling, 1, ...
24                                       size(control, 2));
25 
26     % Pack time bounds and trajectories into optimization vector
27     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
```

```
22
23      % Unpack z
24      [time, state, control] = unpackZ(z, packInfo);
25      [physicalTime, ...
26       physicalState, ...
27       physicalControl] = unpackZ(z, packInfo, true);
28
29      % Initialize inequality and equality constraints
30      c = [];
31      ceq = [];
32
33      % Evaluate defects constraints
34      if isempty(defectConstraints)
35          error('Defect constraints function is not provided');
36      end
37      if isempty(dynamics)
38          error('Dynamics function is not provided');
39      end
40      % Evaluate derivatives
41      derivatives = dynamics(physicalTime, ...
42                             physicalState, ...
43                             physicalControl);
44      % Scale derivatives
45      derivatives = derivatives ./ ...
46                    packInfo.scaling.stateScaling * ...
47                    packInfo.scaling.timeScaling;
48      % Evaluate defects
49      timeStep = (time(end) - time(1)) / packInfo.nGrid;
50      defects = defectConstraints(timeStep, state, derivatives);
51      ceq = [ceq; defects(:)];
52
53
54      % Evaluate boundary constraints
55      if ~isempty(boundaryConstraints)
56          [boundaryIneq, ...
57           boundaryEq] = boundaryConstraints(state(:, 1), ...
58                                             state(:, end), ...
59                                             time(1), ...
60                                             time(end));
61          c = [c; boundaryIneq];
```

```
62          ceq = [ceq; boundaryEq];
63      end
64
65      % Evaluate path constraints
66      if ~isempty(pathConstraints)
67          [pathIneq, pathEq] = pathConstraints(time, state, control);
68          c = [c; pathIneq];
69          ceq = [ceq; pathEq];
70      end
71  end
```

Código Fonte 6 – Função `evaluateConstraints`

A.1.6 `evaluateObjective.m`

```
1   function J = evaluateObjective(z, packInfo, ...
2                                  boundaryObjective, ...
3                                  pathObjective)
4       % Evaluate the combined objective function
5       % Inputs:
6       %   z: [nz,1] = decision variables
7       %   packInfo: struct = information about the problem
8       %   boundaryObjective: function = Mayer term (phi)
9       %   pathObjective: function = Lagrange term (L)
10      %
11      % Outputs:
12      %   J = scalar = objective value
13
14      % Unpack z
15      [time, state, control] = unpackZ(z, packInfo);
16
17      % Initialize objective
18      J = 0;
19
20      % Add Mayer term if provided
21      if ~isempty(boundaryObjective)
22          phi = boundaryObjective(state(:, 1), ...
23                                  state(:, end), ...
24                                  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');

```

```
84 %% Plot results
85 plotResults('Solution', solution(end).z, true);
```

Código Fonte 9 – Arquivo principal do template

A.2.2 templateDynamics.m

```
1  function dx = templateDynamics(time, state, control, params)
2      % Define the dynamics for your problem
3      % Inputs:
4      %   time: [1,n] time vector
5      %   state: [nx,n] state matrix
6      %   control: [nu,n] control matrix
7      %   params: parameter struct
8      % Outputs:
9      %   dx: [nx,n] state derivative matrix
10
11     arguments
12         time double
13         state double
14         control double
15         params struct
16     end
17
18     % TODO: Implement your system dynamics
19     % Example for a double integrator:
20     dx = [
21         state(2,:);                % dx1/dt = x2
22         control(1,:)               % dx2/dt = u1
23     ];
24 end
```

Código Fonte 10 – Função templateDynamics

A.2.3 templateParams.m

```
1  function params = templateParams()
2      % 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
```

```
11
12      arguments
13          x0 double
14          xF double
15          t0 double
16          tF double
17          params struct
18      end
19
20      % TODO: Implement your boundary objective
21      cost = 0;
22  end
```

Código Fonte 14 – Função `boundaryObjective`

A.2.7 `pathObjective.m`

```
1   function L = pathObjective(time, state, control, params)
2       % Evaluates running cost (Lagrange term)
3       % Inputs:
4       %   time: [1,n] time vector
5       %   state: [nx,n] state matrix
6       %   control: [nu,n] control matrix
7       %   params: parameter struct
8       % Outputs:
9       %   L: instantaneous cost value
10
11      arguments
12          time double
13          state double
14          control double
15          params struct
16      end
17
18      % TODO: Implement your path objective
19      L = 0;
20  end
```

Código Fonte 15 – Função `pathObjective`

A.3 Arquivos do problema movimento simples

A.3.1 mainSimpleMass.m

```
% Simple mass moving in 1D
clear;
clc;
close all;

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

%% Create problem instance
nx = 2;  % States: [x; v] (position, velocity)
nu = 1;  % Control: [F] (force)
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
xLow = [-10; -5];    % Position and velocity lower bounds
xUpp = [10; 5];      % Position and velocity upper bounds
problem.setStateBounds(xLow, xUpp);

%% Set control bounds
uLow = [-50];         % Maximum force in negative direction
uUpp = [50];          % Maximum force in positive direction
problem.setControlBounds(uLow, uUpp);

```

```
%% Set parameters
params.MASS = 1.0;    % Mass of the object
problem.setParameters(params);

%% Set boundary conditions
x0 = [0; 0];          % Start at origin with zero velocity
xF = [1; 0];          % End at x=1 with zero velocity
problem.setBoundaryConditions(x0, xF);

%% Set scaling
% problem.setScaling('state', [1; 1]);    % States are already O(1)
% problem.setScaling('control', 1);       % Control is already O(1)
% problem.setScaling('time', 2);          % Time is already O(1)

%% Set functions
problem.setDynamics(@simpleMassDynamics);
problem.setObjective(@boundaryObjective, @pathObjective);
problem.setConstraints(@boundaryConstraints, @pathConstraints);

%% Set solver options
nGrid = [30];  % Start with very few grid points
options = optimoptions('fmincon');
options.Display = 'iter';
options.MaxFunEvals = 1e5;
% options.Algorithm = 'sqp';
% options.EnableFeasibilityMode = true;
% options.SubproblemAlgorithm = 'cg';
% options.OptimalityTolerance = 1e-8;
% options.ConstraintTolerance = 1e-8;
% options.StepTolerance = 1e-8;

problem.setSolverOptions(options, nGrid);

%% Generate initial guess
zGuess = generateSimpleMassGuess(problem);

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

%% Save solution
```

```
76 mkdir('results');
77 save('results/solution.mat', 'solution');
78
79 %% Plot results
80 plotResults('Simple Mass Solution', solution(end).z, true);
```

Código Fonte 16 – Arquivo principal do problema movimento simples

A.3.2 simpleMassDynamics.m

```
1  function dx = simpleMassDynamics(time, state, control, params)
2      arguments
3          time double
4          state double
5          control double
6          params struct
7      end
8
9      % Extract states and control
10     v = state(2, :);         % Velocity
11     F = control(1, :);       % Force
12     m = params.MASS;         % Mass
13
14     % System dynamics
15     dx = [
16           v                % dx/dt = v
17           F / m            % dv/dt = F/m
18          ];
19 end
```

Código Fonte 17 – Função `simpleMassDynamics`

A.3.3 boundaryConstraints.m

```
1  function [c, ceq] = boundaryConstraints(x0, xF, t0, tF, boundaryConditions)
2      arguments
3          x0 double
4          xF double
```

```
        t0 double
        tF double
        boundaryConditions struct
    end

    % Initial and final conditions
    ceq = [
           x0 - boundaryConditions.x0   % Initial state
           xF - boundaryConditions.xF   % Final state
          ];

    % No inequality constraints
    c = [];
end
```

Código Fonte 18 – Função `boundaryConstraints`

A.3.4 `pathConstraints.m`

```
function [c, ceq] = pathConstraints(time, state, control, params)
    arguments
        time double
        state double
        control double
        params struct
    end

    % No path constraints needed
    c = [];   % Inequality constraints
    ceq = []; % Equality constraints
end
```

Código Fonte 19 – Função `pathConstraints`

A.3.5 `boundaryObjective.m`

```
function cost = boundaryObjective(x0, xF, t0, tF, params)
    arguments
```

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

    % No boundary cost
    cost = 0;
end
```

Código Fonte 20 – Função `boundaryObjective`

A.3.6 `pathObjective.m`

```
function L = pathObjective(time, state, control, params)
    arguments
        time double
        state double
        control double
        params struct
    end

    % Minimize control effort
    L = control(1, :).^2;
end
```

Código Fonte 21 – Função `pathObjective`

A.3.7 `generateSimpleMassGuess.m`

```
function zGuess = generateSimpleMassGuess(problem)
    % Extract problem parameters
    [x0, xF] = problem.getBoundaryConditions();
    nGrid = problem.getGridSize();

    % Linear interpolation for states
    state = [
```

```
            linspace(x0(1), xF(1), nGrid(1))   % Position
            linspace(x0(2), xF(2), nGrid(1))   % Velocity
           ];

    % Linear interpolation for control
    control = linspace(1, -1, nGrid(1));

    % Combine states and control
    zGuess = [state; control];
end
```

Código Fonte 22 – Função `generateSimpleMassGuess`

A.4 Arquivos do problema da braquistócrona

A.4.1 `mainBrachistochrone.m`

```
% Brachistochrone problem:
% Find the path of fastest descent between two points
clear;
clc;
close all;

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

%% Create problem instance
nx = 3;  % States: [x; y; v] (horizontal position,
         %                    vertical position,
         %                    velocity)
nu = 1;  % Control: [theta] (angle of the path)
problem = TrajectoryProblem(nx, nu);

%% Set time bounds
t0Low = 0;
t0Upp = 0;
tFLow = 0;
tFUpp = 2 + 1e-6;
```

```
problem.setTimeBounds(t0Low, t0Upp, tFLow, tFUpp);

%% Set time boundary conditions
t0 = 0;
tF = 1;  % Initial guess for final time (will be optimized)
problem.setTimeBoundaryConditions(t0, tF);

%% Set state bounds
xLow = [0; -5; 0];      % Lower bounds: x>=0, y>=-5, v>=0
xUpp = [5; 0; 15];       % Upper bounds: x<=5, y<=0, v<=15
problem.setStateBounds(xLow, xUpp);

%% Set control bounds
uLow = [-pi / 2];         % Minimum angle
uUpp = [pi / 2];          % Maximum angle
problem.setControlBounds(uLow, uUpp);

%% Set parameters
params = brachistochroneParams();
problem.setParameters(params);

%% Set boundary conditions
x0 = [0; 0; 0];         % Start at origin with zero velocity
xF = [5; -5; xUpp(3)]; % End at (5,-5)
                        % with free final velocity up to vMax
problem.setBoundaryConditions(x0, xF);

%% Set scaling
% xScale = abs(xF);
% uScale = 1;          % Angle scale
% tScale = sqrt(2*abs(xF(2))/params.GRAVITY);   % Natural time scale

% problem.setScaling('state', xScale);    % Scale states to be O(1)
% problem.setScaling('control', uScale);  % Scale control to be O(1)
% problem.setScaling('time', tScale);     % Scale time to be O(1)

%% Set functions
problem.setDynamics(@brachistochroneDynamics);
problem.setObjective(@boundaryObjective, @pathObjective);
problem.setConstraints(@boundaryConstraints, @pathConstraints);
```

```
62
63  %% Set solver options
64  nGrid = [10, 20, 40];  % Number of grid points for each iteration
65  options = optimoptions('fmincon');
66  options.Display = 'iter';
67  options.MaxFunEvals = 1e5;
68  options.Algorithm = 'sqp';
69  options.EnableFeasibilityMode = true;
70  options.SubproblemAlgorithm = 'cg';
71  options.FiniteDifferenceType = 'central';  % More accurate gradients
72  options.FiniteDifferenceStepSize = 1e-6;   % Smaller step size
73  options.OptimalityTolerance = 1e-6;        % Tighter tolerance
74  options.ConstraintTolerance = 1e-6;        % Tighter tolerance
75  options.StepTolerance = 1e-10;             % Smaller steps
76  options.MaxIterations = 1000;              % Increase if needed
77  options.ScaleProblem = true;               % Add scaling
78  options.HessianApproximation = 'bfgs';     % Use BFGS approximation
79
80  problem.setSolverOptions(options, nGrid);
81
82  %% Optional: Set constraint checking function
83  problem.setConstraintsCheck(@checkConstraints);
84
85  %% Set variable names
86  problem.setVariableNames({'x', 'y', 'v'}, {'theta'});
87
88  %% Solve the problem
89  zGuess = generateBrachistochroneGuess(problem);
90  solution = problem.solveWithTrapezoidalCollocation(zGuess);
91
92  %% Display solution
93  disp('=== Solution Analysis ===');
94  disp('Time span:');
95  disp(solution(end).z.timeSpan);
96
97  disp('Initial state:');
98  disp(solution(end).z.state(:, 1)');
99  disp('Target initial state:');
100 disp(x0');
101
```

```
disp('Final state:');
disp(solution(end).z.state(:, end)');
disp('Target final state:');
disp(xF');

% Check energy conservation
E = 0.5 * solution(end).z.state(3, :).^2 + ...
params.GRAVITY * solution(end).z.state(2, :);
disp('Energy variation:');
disp(['Max: ', num2str(max(E) - min(E))]);

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

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

%% Compare with analytical solution
compareWithAnalytical(solution(end).z, true);
```

Código Fonte 23 – Arquivo principal do problema da braquistócrona

A.4.2 brachistochroneDynamics.m

```
function dx = brachistochroneDynamics(time, state, control, params)
    arguments
        time double
        state double
        control double
        params struct
    end

    % Extract states and control
    v = state(3, :);         % Velocity
    theta = control(1, :);   % Path angle
    g = params.GRAVITY;      % Gravity

    % System dynamics
```

```
    dx = [
          v .* cos(theta)            % dx/dt = v*cos(theta)
          v .* sin(theta)            % dy/dt = v*sin(theta)
          -g .* sin(theta)           % dv/dt = -g*sin(theta)
         ];
end
```

Código Fonte 24 – Função `brachistochroneDynamics`

A.4.3 brachistochroneParams.m

```
function params = brachistochroneParams()
    % Parameters for the brachistochrone problem
    params.GRAVITY = 9.78;  % Gravitational acceleration [m/s^2]
end
```

Código Fonte 25 – Função `brachistochroneParams`

A.4.4 boundaryConstraints.m

```
function [c, ceq] = boundaryConstraints(x0, xF, t0, tF, boundaryConditions)
    arguments
        x0 double
        xF double
        t0 double
        tF double
        boundaryConditions struct
    end

    % Initial conditions
    ceq = [
           x0(1) - boundaryConditions.x0(1)   % Initial x position
           x0(2) - boundaryConditions.x0(2)   % Initial y position
           x0(3) - boundaryConditions.x0(3)   % Initial velocity
           xF(1) - boundaryConditions.xF(1)   % Final x position
           xF(2) - boundaryConditions.xF(2)   % Final y position
          ];

```

```
    % Debug output
    % disp('Current boundary constraint values:');
    % disp(['Initial x: ', num2str(ceq(1))]);
    % disp(['Initial y: ', num2str(ceq(2))]);
    % disp(['Initial v: ', num2str(ceq(3))]);
    % disp(['Final x: ', num2str(ceq(4))]);
    % disp(['Final y: ', num2str(ceq(5))]);

    % No inequality constraints
    c = [];
end
```

Código Fonte 26 – Função `boundaryConstraints`

A.4.5 pathConstraints.m

```
function [c, ceq] = pathConstraints(time, state, control, params)
    arguments
        time double
        state double
        control double
        params struct
    end

    g = params.GRAVITY;

    % Extract states
    sx = state(1, :);       % Horizontal position
    sy = state(2, :);       % Vertical position
    v = state(3, :);        % Velocity

    % Path constraints
    c = [];
    % Energy conservation (optional)
    E = 0.5 * v.^2 + g * sy;
    E0 = g * sy(1);  % Initial potential energy
    ceq = (E - E0)';  % Energy should be conserved

    % Debug output
```

```
24      % disp('Energy conservation error:');
25      % disp(['Max error: ', num2str(max(abs(ceq)))]);
26      % disp(['Mean error: ', num2str(mean(abs(ceq)))]);
27  end
```

Código Fonte 27 – Função `pathConstraints`

A.4.6 `boundaryObjective.m`

```
1   function cost = boundaryObjective(x0, xF, t0, tF, params)
2       arguments
3           x0 double
4           xF double
5           t0 double
6           tF double
7           params struct
8       end
9
10      % Minimize time
11      cost = tF - t0;
12  end
```

Código Fonte 28 – Função `boundaryObjective`

A.4.7 `pathObjective.m`

```
1   function L = pathObjective(time, state, control, params)
2       arguments
3           time double
4           state double
5           control double
6           params struct
7       end
8
9       % No running cost
10      L = zeros(size(time));
11  end
```

Código Fonte 29 – Função `pathObjective`

A.4.8 generateBrachistochroneGuesses.m

```
function zGuess = generateBrachistochroneGuess(problem, plotFlag)
    % Generate an initial guess for the brachistochrone problem
    %
    % The guess is a parabolic path that connects the initial and final
    % points. The velocity is estimated using energy conservation.
    %
    % Inputs:
    %   problem - A TrajectoryProblem object
    %   plotFlag - A boolean flag to plot the guess
    %
    % Outputs:
    %   zGuess - A guess for the solution

    if nargin < 2
        plotFlag = false;
    end

    % Extract problem parameters
    params = problem.getParameters();
    g = params.GRAVITY;

    % Extract time bounds, boundary conditions, and grid size
    [t0, tF] = problem.getTimeBounds();
    [x0, xF] = problem.getBoundaryConditions();
    nGrid = problem.getGridSize();

    time = linspace(t0, tF, nGrid(1));

    % Initial and final positions
    sx0 = x0(1);
    sy0 = x0(2);
    sxf = xF(1);
    syf = xF(2);

    % Generate parabolic path as initial guess
```

```
    sx = linspace(sx0, sxf, nGrid(1));
    %
    % x - sx0 = a * (y - sy0)^2
    %
    a = (sxf - sx0) / (syf - sy0)^2;
    sy = sy0 + sign(syf - sy0) * sqrt((sx - sx0) / a);

    % Estimate velocity (using energy conservation)
    v = sqrt(2 * g * (sy0 - sy));

    % Estimate control angle
    dsx = gradient(sx, time);
    dsy = gradient(sy, time);
    theta = atan2(dsy, dsx);

    state = [sx; sy; v];
    control = theta;
    zGuess = [state; control];

    % Plot if requested
    if plotFlag
        guess = struct();
        guess.time = time;
        guess.state = state;
        guess.control = control;
        plotResults('Initial Guess', guess);
        compareWithAnalytical(guess);
    end
end
```

Código Fonte 30 – Função `generateBrachistochroneGuess`

A.4.9 `checkConstraints.m`

```
function violations = checkConstraints(time, state, control, params)
    violations = struct();
    violations.state = struct();
    violations.state.x = struct();
    violations.state.y = struct();
```

```
    violations.state.v = struct();
    violations.control = struct();
    violations.control.theta = struct();
    violations.energy = struct();
    violations.velocity = struct();

    % Check state bounds
    if any(state(1, :) < 0) || any(state(1, :) > 10)
        violations.state.x = struct( ...
                                    'max', max(state(1, :)), ...
                                    'min', min(state(1, :)), ...
                                    'bounds', [0, 10]);
    end

    if any(state(2, :) < -10) || any(state(2, :) > 0)
        violations.state.y = struct( ...
                                    'max', max(state(2, :)), ...
                                    'min', min(state(2, :)), ...
                                    'bounds', [-10, 0]);
    end

    if any(state(3, :) < 0) || any(state(3, :) > 20)
        violations.state.v = struct( ...
                                    'max', max(state(3, :)), ...
                                    'min', min(state(3, :)), ...
                                    'bounds', [0, 20]);
    end

    % Check control bounds
    if any(control < -pi / 2) || any(control > pi / 2)
        violations.control.theta = struct( ...
                                        'max', max(control), ...
                                        'min', min(control), ...
                                        'bounds', [-pi / 2, pi / 2]);
    end

    % Check energy conservation
    E = 0.5 * state(3, :).^2 + params.GRAVITY * state(2, :);
    dE = gradient(E, time);

```

```
46      if any(abs(dE) > 1e-6)
47          violations.energy = struct( ...
48                                      'maxChange', max(abs(dE)), ...
49                                      'meanChange', mean(abs(dE)));
50      end
51
52      % Check velocity consistency
53      vFromEnergy = sqrt(2 * params.GRAVITY * (state(2, 1) - state(2, :)));
54      velocityError = abs(state(3, :) - vFromEnergy);
55
56      if any(velocityError > 1e-6)
57          violations.velocity = struct( ...
58                                       'maxError', max(velocityError), ...
59                                       'meanError', mean(velocityError));
60      end
61
62      % Debug: print violations if any
63      % if ~isempty(fieldnames(violations))
64      %     disp('Constraint Violations Found:');
65      %     if ~isempty(fieldnames(violations.energy))
66      %         disp('Energy:');
67      %         disp(violations.energy);
68      %     end
69      %     if ~isempty(fieldnames(violations.velocity))
70      %         disp('Velocity:');
71      %         disp(violations.velocity);
72      %     end
73      %     if ~isempty(fieldnames(violations.state))
74      %         disp('State:');
75      %         if ~isempty(fieldnames(violations.state.x))
76      %             disp('sx:');
77      %             disp(violations.state.x);
78      %         end
79      %         if ~isempty(fieldnames(violations.state.y))
80      %             disp('sy:');
81      %             disp(violations.state.y);
82      %         end
83      %         if ~isempty(fieldnames(violations.state.v))
84      %             disp('v:');
85      %             disp(violations.state.v);
```

```
86      %         end
87      %     end
88      %     if ~isempty(fieldnames(violations.control))
89      %         disp('Control:');
90      %         if ~isempty(fieldnames(violations.control.theta))
91      %             disp('theta:');
92      %             disp(violations.control.theta);
93      %         end
94      %     end
95      % end
96  end
```

Código Fonte 31 – Função `checkConstraints`

A.4.10 `compareWithAnalytical.m`

```
1   function compareWithAnalytical(z, plotFlag)
2       % Compare numerical solution with analytical cycloid solution
3
4       if nargin < 2
5           plotFlag = false;
6       end
7
8       % Extract final conditions
9       xf = z.state(1, end);
10      yf = z.state(2, end);
11
12      % Calculate cycloid parameters
13
14      % Calculate final angle
15      f = @(theta_f) xf / yf - ...
16      (theta_f - sin(theta_f)) / (1 - cos(theta_f));
17      theta_f = fsolve(f, pi / 2, ...
18                       optimoptions('fsolve', 'Display', 'off'));
19
20      % Calculate radius
21      R = yf / (1 - cos(theta_f));
22
23      % Generate analytical solution points
```

```
    theta = linspace(0, theta_f, 200);
    x_analytical = R * (theta - sin(theta));
    y_analytical = R * (1 - cos(theta));

    % Plot comparison
    figure('Name', 'Comparison with Analytical Solution');
    plot(z.state(1, :), z.state(2, :), 'b-', ...
         'LineWidth', 2, 'DisplayName', 'Numérica');
    hold on;
    plot(x_analytical, y_analytical, 'r--', ...
         'LineWidth', 2, 'DisplayName', 'Analítica');
    plot([0 xf], [0 yf], 'k.', ...
         'MarkerSize', 20, 'DisplayName', 'Pontos Iniciais e Finais');
    grid on;
    xlabel('x [m]');
    ylabel('y [m]');
    title('Comparação com Solução Analítica (Ciclóide)');
    legend('Location', 'best');
    axis equal;

    if plotFlag
        mkdir('figures');
        savefig('figures/Brachistochrone Comparison.fig');
        saveas(gcf, 'figures/Brachistochrone Comparison.svg');
        saveas(gcf, 'figures/Brachistochrone Comparison.png');
    end
end
```

Código Fonte 32 – Função `compareWithAnalytical`

A.5 Arquivos do problema do eVTOL

A.5.1 `mainEvtol.m`

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

```
5
6   % Add path to the TrajectoryProblem class
7   addpath('..');
8
9   %% Create problem instance
10  problem = TrajectoryProblem(5, 2);  % 5 states, 2 controls
11
12  %% Set time bounds
13  t0Low = 0;
14  t0Upp = 0;
15  tFLow = 0;
16  tFUpp = 45;
17  problem.setTimeBounds(t0Low, t0Upp, tFLow, tFUpp);
18
19  %% Set time boundary conditions
20  t0 = 0;
21  tF = 45;
22  problem.setTimeBoundaryConditions(t0, tF);
23
24  %% Set state bounds
25  xLow = [0; -10; 0; -5; 0];
26  xUpp = [1000; 110; 35; 6; 1e8];
27  problem.setStateBounds(xLow, xUpp);
28
29  %% Set control bounds
30  uLow = [0; 0];
31  uUpp = [1800; 2600];
32  problem.setControlBounds(uLow, uUpp);
33
34  %% Set scaling
35  % problem.setScaling('time', tF - t0);
36  % problem.setScaling('state', xUpp - xLow);
37  % problem.setScaling('control', uUpp - uLow);
38
39  %% Set parameters
40  problem.setParameters(evtolParams());
41
42  %% Set boundary conditions
43  x0 = [0; 0; 0; 0; 0];
44  xF = [1000; 100; 25; 0; 1e8];  % Free final energy
```

```
45  problem.setBoundaryConditions(x0, xF);
46
47  %% Set functions
48  % Set dynamics
49  problem.setDynamics(@evtolDynamics);
50
51  % Set objective
52  hBoundaryObjective = @boundaryObjective;  % Mayer Term
53  hPathObjective = [];  % integrand of Lagrange Term
54  problem.setObjective(hBoundaryObjective, hPathObjective);
55
56  % Set constraints
57  hBoundaryConstraint = @boundaryConstraints;  % Boundary Constraints
58  hPathConstraint = [];  % Path Constraints
59  problem.setConstraints(hBoundaryConstraint, hPathConstraint);
60
61  %% Set solver options
62  nGrid = [10, 20, 40, 80];
63  options = optimoptions('fmincon');
64  options.Display = 'iter';
65  options.MaxFunEvals = 1e5;
66  options.Algorithm = 'sqp';
67  options.EnableFeasibilityMode = true;
68  options.SubproblemAlgorithm = 'cg';
69  options.FiniteDifferenceType = 'central';  % More accurate gradients
70  options.FiniteDifferenceStepSize = 1e-6;   % Smaller step size
71  options.OptimalityTolerance = 1e-6;        % Tighter tolerance
72  options.ConstraintTolerance = 1e-6;        % Tighter tolerance
73  options.StepTolerance = 1e-10;             % Smaller steps
74  options.MaxIterations = 1000;              % Increase if needed
75  options.ScaleProblem = true;               % Add scaling
76  options.HessianApproximation = 'bfgs';     % Use BFGS approximation
77
78  problem.setSolverOptions(options, nGrid);
79
80  % Optional: Set constraint checking function
81  problem.setConstraintsCheck(@checkConstraints);
82
83  % Optional: Set variable names
84  problem.setVariableNames({'sx', 'sy', 'vx', 'vy', 'E'}, {'Tx', 'Ty'});
```

```
85
86  % Validate problem definition
87  problem.validate();
88
89  % Optional: Get initial guess
90  zGuess = physicalInitialGuess(problem);
91
92  % Solve the problem
93  solution = problem.solveWithTrapezoidalCollocation(zGuess);
94
95  % Check constraints
96  violations = solution(end).violations;
97  % Display constraint violations
98  if ~isempty(fieldnames(violations.state))
99      disp('State Violations:');
100     disp(violations.state);
101 end
102
103 if ~isempty(fieldnames(violations.control))
104     disp('Control Violations:');
105     disp(violations.control);
106 end
107
108 if ~isempty(fieldnames(violations.physical))
109     disp('Physical Consistency:');
110     disp(violations.physical.velocity);
111     disp(violations.physical.flightPathAngle);
112 end
113
114 disp(solution(end).output);
115
116 % Save the solution
117 mkdir('results');
118 save(['results/solution-base-t', num2str(tF), '.mat'], 'solution');
119
120 % Plot results
121 plotResults('Final Solution', solution(end).z, true);
```

Código Fonte 33 – Arquivo principal do problema do eVTOL

A.5.2 evtolDynamics.m

```
function stateDerivatives = evtolDynamics(time, state, control, params)
    % Define the dynamics for the EVTOL problem
    %
    % Inputs:
    %   time: [1,n] time vector
    %   state: [5,n] = [posX; posY; velX; velY; energy] = state matrix
    %   control: [2,n] = [thrustX; thrustY] = control matrix
    %   params: parameter struct
    %
    % Outputs:
    %   stateDerivatives: [5,n] = [velX; velY; accelX; accelY; powerRate] = deri
    %
    arguments
        time double
        state double
        control double
        params struct
    end

    % Get the parameters
    GRAVITY = params.environment.GRAVITY;
    AIR_DENSITY = params.environment.AIR_DENSITY;
    mass = params.aircraft.mass;
    rotorArea = params.aircraft.rotorArea;
    chi = params.aircraft.chi;

    % Assign state variables
    posX = state(1, :);
    posY = state(2, :);
    velX = state(3, :);
    velY = state(4, :);
    energy = state(5, :);

    % Assign control variables
    thrustX = control(1, :);
    thrustY = control(2, :);

    % Compute aerodynamic parameters
```

```
39      V = sqrt(velX.^2 + velY.^2);
40      gamma = computeFlightAngle(velX, velY);
41      angleToVertical = pi / 2 - gamma;  % Angle between
42                                         % velocity and vertical
43      angleToHorizontal = gamma;  % Angle between
44                                  % velocity and horizontal
45
46      % Compute the aerodynamic forces
47      [lift, drag] = computeLiftDrag(velX, velY, params);
48
49      % Compute the thrust forces
50      rotorThrustX = thrustX / 2;
51      rotorThrustY = thrustY / 4;
52
53      % Induced velocity calculations
54      hoverVelocityX = sqrt(rotorThrustX / ...
55                            (2 * AIR_DENSITY * rotorArea));
56      hoverVelocityY = sqrt(rotorThrustY / ...
57                            (2 * AIR_DENSITY * rotorArea));
58
59      inducedVelocityX = computeInducedVelocity(hoverVelocityX, ...
60                                                V, angleToHorizontal);
61      inducedVelocityY = computeInducedVelocity(hoverVelocityY, ...
62                                                V, angleToVertical);
63
64      % Power calculations
65      rotorPowerX = rotorThrustX .* inducedVelocityX;
66      rotorPowerY = rotorThrustY .* inducedVelocityY;
67      armPowerX = 2.0 * rotorPowerX * (1 + chi);
68      armPowerY = 4.0 * rotorPowerY * (1 + chi);
69      inducedPower = armPowerX + armPowerY;
70
71      forwardPower = thrustX .* V .* sin(angleToHorizontal) + ...
72                     thrustY .* V .* sin(angleToVertical);
73
74      % Compute the derivatives
75      stateDerivatives = [
76                          velX  % posX derivative
77                          velY  % posY derivative
78                          (thrustX - drag .* cos(gamma) - ...
```

```
                        lift .* sin(gamma)) / mass  % velX derivative
                        (thrustY - drag .* sin(gamma) + ...
                            lift .* cos(gamma) - mass * GRAVITY) ...
                            / mass  % velY derivative
                        inducedPower + forwardPower  % energy derivative
                       ];
end
```

Código Fonte 34 – Função `evtolDynamics`

A.5.3 `computeLiftDrag.m`

```
function [lift, drag] = computeLiftDrag(velX, velY, params)
    % Computes aerodynamic forces for the EVTOL
    %
    % Inputs:
    %   velX: [1,n] = horizontal velocity component
    %   velY: [1,n] = vertical velocity component
    %   params: parameter struct
    %
    % Outputs:
    %   lift: [1,n] = lift force
    %   drag: [1,n] = drag force

    arguments
        velX double
        velY double
        params struct
    end

    % Extract parameters
    rho = params.environment.AIR_DENSITY;

    CDFus = params.aircraft.fuselage.CD;
    SfFus = params.aircraft.fuselage.Sf;
    StFus = params.aircraft.fuselage.St;

    SWing = params.aircraft.wing.S;
    eWing = params.aircraft.wing.e;
```

```
28      CDpWing = params.aircraft.wing.CDp;
29      ARWing = params.aircraft.wing.AR;
30      CL0Wing = params.aircraft.wing.CL0;
31      CLalphaWing = params.aircraft.wing.CLalpha;
32      alphaFus = params.aircraft.wing.alphaFus;
33
34      % Variables related to the wing lift coefficient
35      M = params.aircraft.wing.sigmoid.M;
36      alpha0 = params.aircraft.wing.sigmoid.alpha0;
37
38      gamma = computeFlightAngle(velX, velY);
39      alpha = -gamma;
40      alphaWing = alpha + alphaFus; % Wing angle of attack
41
42      % Compute Wing Lift Coefficient
43      straightCL = CL0Wing + CLalphaWing * alphaWing;
44
45      % Apply sigmoid function to merge linear and flat plate models
46      sigMinus = exp(-M * (alphaWing - alpha0));
47      sigPlus = exp(M * (alphaWing + alpha0));
48      sigma = (1 + sigMinus + sigPlus) ./ ...
49              ((1 + sigMinus) .* (1 + sigPlus));
50
51      CD = CDpWing + straightCL.^2 / (pi * eWing * ARWing);
52      CL = sigma .* (2 * sign(alphaWing) .* ...
53                     sin(alphaWing).^2 .* cos(alphaWing)) + ...
54          (1 - sigma) .* straightCL;
55
56      V = sqrt(velX.^2 + velY.^2);
57
58      LWing = 0.5 * rho * V.^2 * SWing .* CL;
59      DWing = 0.5 * rho * V.^2 * SWing .* CD;
60
61      % Fuselage drag
62      DfFus = 0.5 * rho * velX.^2 * CDFus * SfFus;
63      DtFus = 0.5 * rho * velY.^2 * CDFus * StFus;
64
65      % Final computation
66      lift = LWing;
67      drag = sqrt(DfFus.^2 + DtFus.^2) + DWing;
```

```
68 end
```

Código Fonte 35 – Função `computeLiftDrag`

A.5.4 `computeInducedVelocity.m`

```
1  function vi = computeInducedVelocity(vh, V, alpha)
2      % Computes induced velocity using momentum theory
3      %
4      % Inputs:
5      %   vh: [1,n] hover induced velocity
6      %   V: [1,n] free stream velocity
7      %   alpha: [1,n] angle between free stream and rotor disk normal
8      %
9      % Outputs:
10     %   vi: [1,n] induced velocity
11
12     arguments
13         vh double
14         V double
15         alpha double
16     end
17
18     % Input validation
19     assert(all(isfinite(vh)), 'Non-finite hover velocity');
20     assert(all(isfinite(V)), 'Non-finite velocity');
21     assert(all(isfinite(alpha)), 'Non-finite angle');
22
23     % For hover (V=0), vi should equal vh
24     hover_condition = (V < 1e-6);
25
26     vi = zeros(1, length(vh));
27     vi(hover_condition) = vh(hover_condition);
28
29     vi_func = @(w, vh, V, alpha) w - vh.^2 ./ ...
30                                  (sqrt((V .* cos(alpha)).^2) + ...
31                                   (V .* sin(alpha) + w).^2);
32
33     % For non-hover cases, use fzero
```

```
    non_hover = ~hover_condition;
    options = optimset('TolX', 1e-6);

    for i = find(non_hover)
        vi(i) = fzero(@(w) vi_func(w, vh(i), V(i), alpha(i)), ...
        [-50, 50], options);
    end
end
```

Código Fonte 36 – Função `computeInducedVelocity`

A.5.5 `computeFlightAngle.m`

```
function gamma = computeFlightAngle(velX, velY)
    % Computes flight path angle
    %
    % Inputs:
    %   velX: [1,n] = horizontal velocity component
    %   velY: [1,n] = vertical velocity component
    %
    % Outputs:
    %   gamma: [1,n] = flight path angle [rad]

    arguments
        velX double
        velY double
    end

    gamma = pi / 2 * ones(size(velX));

    conditionX = abs(velX) > 1e-4;
    conditionY = abs(velY) > 1e-4;

    gamma(conditionX) = atan2(velY(conditionX), velX(conditionX));

    gamma(~conditionX & conditionY) = sign(velY(~conditionX & ...
                                            conditionY)) * ...
                                            pi / 2;
end
```

Código Fonte 37 – Função `computeFlightAngle`

A.5.6 `evtolParams.m`

```
function params = evtolParams()
    % Returns the EVTOL problem parameters

    % Environment parameters
    params.environment.GRAVITY = 9.78;  % Gravitational acceleration [m/s^2]
    params.environment.AIR_DENSITY = 1.2;  % Air density [kg/m^3]

    % Aircraft parameters
    params.aircraft.mass = 240;  % Vehicle mass [kg]
    params.aircraft.rotorArea = 1.6;  % Rotor disk area [m^2]
    params.aircraft.chi = 1.0;  % Rotor-to-rotor
                                % interference factor [-]

    params.aircraft.fuselage.CD = 1.0;  % Fuselage drag coefficient [-]
    params.aircraft.fuselage.Sf = 2.11;  % Fuselage cross-sectional
                                         % area [m^2]
    params.aircraft.fuselage.St = 1.47;  % Fuselage top area [m^2]

    params.aircraft.wing.S = 4.0;  % Wing area [m^2]
    params.aircraft.wing.e = 0.9;  % Wing span efficiency factor [-]
    params.aircraft.wing.CDp = 0.0437;  % Wing parasite
                                        % drag coefficient [-]
    params.aircraft.wing.AR = 20.0;  % Wing aspect ratio [-]
    params.aircraft.wing.CL0 = 0.28;  % Wing lift coefficient at zero
                                      % AoA [-]
    params.aircraft.wing.CLalpha = 4.00;  % Wing lift curve
                                          % slope [rad^-1]
    params.aircraft.wing.alphaFus = 0 * (pi / 180);  % Wing mounting
                                                     % angle [rad]
    params.aircraft.wing.sigmoid.M = 4.0;  % Sigmoid steepness
                                           % parameter [-]
    params.aircraft.wing.sigmoid.alpha0 = 20 * (pi / 180);  % Stall
                                                            % angle [rad]

```

```
35 end
```

Código Fonte 38 – Função `evtolParams`

A.5.7 `boundaryConstraints.m`

```
1  function [c, ceq] = boundaryConstraints(x0, xF, t0, tF, boundaryConditions)
2      % Evaluates boundary constraints for the EVTOL problem
3      %
4      % Inputs:
5      %   x0: [5,1] = initial state vector [sx; sy; vx; vy; E]
6      %   xF: [5,1] = final state vector [sx; sy; vx; vy; E]
7      %   t0: double = initial time
8      %   tF: double = final time
9      %
10     % Outputs:
11     %   c: inequality constraints at boundaries
12     %   ceq: equality constraints at boundaries
13
14     arguments
15         x0 double
16         xF double
17         t0 {mustBeNumeric}
18         tF {mustBeNumeric}
19         boundaryConditions struct
20     end
21
22     % Initial conditions
23     initialPosX = x0(1);
24     initialPosY = x0(2);
25     initialVelX = x0(3);
26     initialVelY = x0(4);
27     initialEnergy = x0(5);
28
29     % Final conditions
30     finalPosX = xF(1);
31     finalPosY = xF(2);
32     finalVelX = xF(3);
33     finalVelY = xF(4);
```

```
34
35      % Equality constraints
36      ceq = [
37              % Initial conditions
38              % x(0) = boundaryConditions.xLow(1)
39              initialPosX - boundaryConditions.xLow(1)
40              % y(0) = boundaryConditions.xLow(2)
41              initialPosY - boundaryConditions.xLow(2)
42              % vx(0) = boundaryConditions.xLow(3)
43              initialVelX - boundaryConditions.xLow(3)
44              % vy(0) = boundaryConditions.xLow(4)
45              initialVelY - boundaryConditions.xLow(4)
46              % E(0) = boundaryConditions.xLow(5)
47              initialEnergy - boundaryConditions.xLow(5)
48
49              % Final conditions
50              % x(T) = boundaryConditions.xF(1)
51              finalPosX - boundaryConditions.xF(1)
52              % y(T) = boundaryConditions.xF(2)
53              finalPosY - boundaryConditions.xF(2)
54              % vx(T) = boundaryConditions.xF(3)
55              finalVelX - boundaryConditions.xF(3)
56              % vy(T) = boundaryConditions.xF(4)
57              finalVelY - boundaryConditions.xF(4)
58
59              % Time bounds
60              t0 - boundaryConditions.t0
61              tF - boundaryConditions.tF
62             ];
63
64      % No inequality constraints
65      c = [];
66  end
```

Código Fonte 39 – Função `boundaryConstraints`

A.5.8 pathConstraints.m

```
function [c, ceq] = pathConstraints(time, state, control, params)
    % Evaluates path constraints for the EVTOL problem
    % Only include constraints that cannot be expressed
    % as simple bounds
    %
    % Example of potential nonlinear constraints:
    % c = [
    %     sqrt(x(3,:).^2 + x(4,:).^2) - v_max;  % velocity magnitude
    %                                           % constraint
    %     sqrt(u(1,:).^2 + u(2,:).^2) - T_max;  % total thrust
    %                                           % constraint
    % ];
    %
    % Inputs:
    %   time: [1,n] = time vector
    %   state: [5,n] = state vector [sx; sy; vx; vy; E]
    %   control: [2,n] = control vector [Tx; Ty]
    %   params: parameter struct
    %
    % Outputs:
    %   c: [m,1] = inequality constraints
    %   ceq: [m,1] = equality constraints

    arguments
        time double
        state double
        control double
        params struct
    end

    % No path constraints
    c = [];
    ceq = [];
end
```

Código Fonte 40 – Função `pathConstraints`

A.5.9 boundaryObjective.m

```
function cost = boundaryObjective(x0, xF, t0, tF, p)
    % Evaluates boundary cost for the EVTOL problem
    %
    % Inputs:
    %   x0: [5,1] = initial state vector [sx; sy; vx; vy; E]
    %   xF: [5,1] = final state vector [sx; sy; vx; vy; E]
    %   t0: double = initial time
    %   tF: double = final time
    %   p: parameter struct
    %
    % Outputs:
    %   cost: scalar cost value

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

    % Extract final energy from state
    finalEnergy = xF(5);

    % Minimize final energy
    cost = finalEnergy;
end
```

Código Fonte 41 – Função `boundaryObjective`

A.5.10 pathObjective.m

```
function L = pathObjective(time, state, control, params)
    % Evaluates running cost for the EVTOL problem
    %
    % Inputs:
    %   time: [1,n] = time vector
```

```
    %   state: [5,n] = state vector [x; y; vx; vy; E]
    %   control: [2,n] = control vector [Tx; Ty]
    %   params: parameter struct
    %
    % Outputs:
    %   L: [1,n] = instantaneous cost value

    arguments
        time double
        state double
        control double
        params struct
    end

    % No running cost (only terminal cost)
    L = zeros(1, length(time));
end
```

Código Fonte 42 – Função `pathObjective`

A.5.11 physicalInitialGuess.m

```
function zGuess = physicalInitialGuess(problem, plotFlag)
    % Generate a physics-based initial guess for the EVTOL problem
    if nargin < 2
        plotFlag = false;
    end

    % Extract problem parameters
    params = problem.getParameters();
    mass = params.aircraft.mass;
    g = params.environment.GRAVITY;

    % Extract time bounds, boundary conditions, and grid size
    [t0, tF] = problem.getTimeBounds();
    [x0, xF] = problem.getBoundaryConditions();
    nGrid = problem.getGridSize();

    % Generate time vector and relative time
```

```
    time = linspace(t0, tF, nGrid(1));
    tRel = time - t0;  % Shift time to start at 0
    duration = tF - t0;

    % Calculate trajectory using relative time
    posX = x0(1) + ...
           x0(3) * tRel + ...
           (3 * (xF(1) - x0(1)) / duration^2 - ...
           2 * x0(3) / duration - xF(3) / duration) * tRel.^2 + ...
           (-2 * (xF(1) - x0(1)) / duration^3 + ...
           (x0(3) + xF(3)) / duration^2) * tRel.^3;
    posY = x0(2) + ...
           x0(4) * tRel + ...
           (3 * (xF(2) - x0(2)) / duration^2 - ...
           2 * x0(4) / duration - xF(4) / duration) * tRel.^2 + ...
           (-2 * (xF(2) - x0(2)) / duration^3 + ...
           (x0(4) + xF(4)) / duration^2) * tRel.^3;
    velX = gradient(posX, time);  % Note: gradient still uses
                                  % original time step
    velY = gradient(posY, time);

    % Add validation
    assert(all(isfinite(velX)), 'Non-finite values in velX');
    assert(all(isfinite(velY)), 'Non-finite values in velY');

    % Calculate minimum thrust needed to maintain trajectory
    [lift, drag] = computeLiftDrag(velX, velY, params);
    gamma = computeFlightAngle(velX, velY);

    % Solve for required thrust
    thrustX = drag .* cos(gamma) + lift .* sin(gamma);
    thrustY = drag .* sin(gamma) - lift .* cos(gamma) + mass .* g;

    % Calculate power at this point
    state = [posX; posY; velX; velY; zeros(1, length(time))];
    control = [thrustX; thrustY];
    dx = evtolDynamics(time, state, control, params);
    dE = dx(5, :);

    % Integrate power to get energy
```

```
    energy = x0(5) + cumtrapz(time, dE);

    state(5, :) = energy;

    % Combine into guess
    zGuess = [state; control];

    % Plot if requested
    if plotFlag
        guess = struct();
        guess.time = time;
        guess.state = state;
        guess.control = control;
        plotResults('Initial Guess', guess);
    end
end
```

Código Fonte 43 – Função `physicalInitialGuess`

A.5.12 checkConstraints.m

```
% Auxiliary function to check constraints
function violations = checkConstraints(time, state, control, params)
    % Initialize violations structure
    violations = struct();
    violations.state = struct();
    violations.control = struct();
    violations.physical = struct();

    % Check state bounds
    if any(state(1, :) > 1000 | state(1, :) < 0)
        violations.state.sx = struct( ...
                                    'max', max(state(1, :)), ...
                                    'min', min(state(1, :)), ...
                                    'bounds', [0, 1000], ...
                                    'numViolations', ...
                                    sum(state(1, :) > 1000 | ...
                                        state(1, :) < 0));
    end
```

```
    if any(state(2, :) > 110 | state(2, :) < -10)
        violations.state.sy = struct( ...
                                    'max', max(state(2, :)), ...
                                    'min', min(state(2, :)), ...
                                    'bounds', [-10, 110], ...
                                    'numViolations', ...
                                    sum(state(2, :) > 110 | ...
                                        state(2, :) < -10));
    end
    if any(state(3, :) > 35 | state(3, :) < 0)
        violations.state.vx = struct( ...
                                    'max', max(state(3, :)), ...
                                    'min', min(state(3, :)), ...
                                    'bounds', [0, 35], ...
                                    'numViolations', ...
                                    sum(state(3, :) > 35 | ...
                                        state(3, :) < 0));
    end
    if any(state(4, :) > 6 | state(4, :) < -5)
        violations.state.vy = struct( ...
                                    'max', max(state(4, :)), ...
                                    'min', min(state(4, :)), ...
                                    'bounds', [-5, 6], ...
                                    'numViolations', ...
                                    sum(state(4, :) > 6 | ...
                                        state(4, :) < -5));
    end

    % Check control bounds
    if any(control(1, :) > 1800 | control(1, :) < 0)
        violations.control.Tx = struct( ...
                                      'max', max(control(1, :)), ...
                                      'min', min(control(1, :)), ...
                                      'bounds', [0, 1800], ...
                                      'numViolations', ...
                                      sum(control(1, :) > 1800 | ...
                                          control(1, :) < 0));
    end
    if any(control(2, :) > 2600 | control(2, :) < 0)
        violations.control.Ty = struct( ...
```

```
                                        'max', max(control(2, :)), ...
                                        'min', min(control(2, :)), ...
                                        'bounds', [0, 2600], ...
                                        'numViolations', ...
                                        sum(control(2, :) > 2600 | ...
                                            control(2, :) < 0));
    end

    % Check physical consistency
    V = sqrt(state(3, :).^2 + state(4, :).^2);
    gamma = atan2(state(4, :), state(3, :));

    violations.physical.velocity = struct( ...
                                          'max', max(V), ...
                                          'min', min(V), ...
                                          'mean', mean(V));

    violations.physical.flightPathAngle = struct( ...
                                                 'max', ...
                                                 max(abs(gamma)) * 180 / pi, ...
                                                 'min', ...
                                                 min(gamma) * 180 / pi, ...
                                                 'mean', ...
                                                 mean(gamma) * 180 / pi);
end
```

Código Fonte 44 – Função `checkConstraints`

A.5.13 `testAeroForces.m`

```
function testAeroForces()
    % Test function to validate lift and drag calculations
    % across different angles of attack

    params = evtolParams();

    % Extract parameters
    AIR_DENSITY = params.environment.AIR_DENSITY;
    wingS = params.aircraft.wing.S;
```

```
    wingCL0 = params.aircraft.wing.CL0;
    wingCLalpha = params.aircraft.wing.CLalpha;
    wingAlphaFus = params.aircraft.wing.alphaFus;
    wingM = params.aircraft.wing.sigmoid.M;
    wingAlpha0 = params.aircraft.wing.sigmoid.alpha0;

    % Test velocities
    V = 25; % m/s
    alphaRange = (-30:1:30) * (pi / 180); % Convert degrees to radians

    % Initialize arrays
    lift = zeros(size(alphaRange));
    drag = zeros(size(alphaRange));

    % Calculate forces for each angle of attack
    for i = 1:length(alphaRange)
        alpha = alphaRange(i);
        velX = V * cos(alpha);
        velY = V * sin(alpha);
        [lift(i), drag(i)] = computeLiftDrag(velX, velY, params);
    end

    % Calculate theoretical values for comparison
    % Linear region lift coefficient (before stall)
    linearCL = wingCL0 + wingCLalpha * (alphaRange + wingAlphaFus);
    linearLift = 0.5 * AIR_DENSITY * V^2 * wingS .* linearCL;

    % Plot results
    figure('Name', 'Aerodynamic Forces Validation');

    % Plot Lift
    subplot(2, 2, 1);
    plot(alphaRange * 180 / pi, lift, 'b-', 'LineWidth', 2, ...
         'DisplayName', 'Computed Lift');
    hold on;
    plot(alphaRange * 180 / pi, linearLift, 'r--', 'LineWidth', 1.5, ...
         'DisplayName', 'Linear Theory');
    grid on;
    xlabel('Angle of Attack (degrees)');
    ylabel('Lift Force (N)');
```

```
50      title('Lift vs Angle of Attack');
51      legend('show');
52
53      % Plot Drag
54      subplot(2, 2, 2);
55      plot(alphaRange * 180 / pi, drag, 'b-', 'LineWidth', 2);
56      grid on;
57      xlabel('Angle of Attack (degrees)');
58      ylabel('Drag Force (N)');
59      title('Drag vs Angle of Attack');
60
61      % Plot L/D ratio
62      subplot(2, 2, 3);
63      plot(alphaRange * 180 / pi, lift ./ drag, 'b-', 'LineWidth', 2);
64      grid on;
65      xlabel('Angle of Attack (degrees)');
66      ylabel('L/D Ratio');
67      title('Lift-to-Drag Ratio');
68
69      % Print key values
70      [maxLD, maxLDIdx] = max(lift ./ drag);
71      alphaMaxLD = alphaRange(maxLDIdx) * 180 / pi;
72
73      [maxL, maxLIdx] = max(lift);
74      alphaMaxL = alphaRange(maxLIdx) * 180 / pi;
75
76      fprintf('\nKey Performance Parameters:\n');
77      fprintf('Maximum L/D Ratio: %.2f at %.1f degrees\n', ...
78              maxLD, alphaMaxLD);
79      fprintf('Maximum Lift: %.2f N at %.1f degrees\n', maxL, alphaMaxL);
80      fprintf('Stall Angle: %.1f degrees (theoretical)\n', ...
81              wingAlpha0 * 180 / pi);
82
83      % Verify transition between linear and flat plate models
84      subplot(2, 2, 4);
85      straightCL = wingCL0 + wingCLalpha * (alphaRange + wingAlphaFus);
86      M = wingM;
87      alpha0 = wingAlpha0;
88
89      sigMinus = exp(-M * (alphaRange - alpha0));
```

```
    sigPlus = exp(M * (alphaRange + alpha0));
    sigma = (1 + sigMinus + sigPlus) ./ ...
            ((1 + sigMinus) .* (1 + sigPlus));

    flatPlateCL = 2 * sign(alphaRange) .* ...
                  sin(alphaRange).^2 .* cos(alphaRange);
    CLCombined = sigma .* flatPlateCL + (1 - sigma) .* straightCL;

    plot(alphaRange * 180 / pi, straightCL, 'r--', 'LineWidth', 1.5, ...
         'DisplayName', 'Linear Model');
    hold on;
    plot(alphaRange * 180 / pi, flatPlateCL, 'g--', 'LineWidth', 1.5, ...
         'DisplayName', 'Flat Plate');
    plot(alphaRange * 180 / pi, CLCombined, 'b-', 'LineWidth', 2, ...
         'DisplayName', 'Combined Model');
    grid on;
    xlabel('Angle of Attack (degrees)');
    ylabel('Lift Coefficient');
    title('Lift Coefficient Models');
    legend('show');
end
```

Código Fonte 45 – Função `testAeroForces`

A.5.14 `testEvtolDynamics.m`

```
% Test function to verify dynamics implementation
function testEvtolDynamics()
    % Setup test parameters matching your current configuration
    params = evtolParams();
    GRAVITY = params.environment.GRAVITY;
    AIR_DENSITY = params.environment.AIR_DENSITY;
    mass = params.aircraft.mass;
    rotorArea = params.aircraft.rotorArea;
    chi = params.aircraft.chi;

    % Test Case 1: Hover condition
    hoverState = [0; 0; 0; 0; 0];
    hoverControl = [0; mass * GRAVITY];
```

```
14      disp('Hover Test:');
15      dxHover = evtolDynamics(hoverState, hoverControl, params);
16      disp(['    Vertical acceleration: ' num2str(dxHover(4))]);
17      disp(['    Power consumption: ' num2str(dxHover(5))]);
18      % Debug output
19      disp('Hover calculations:');
20      disp(['    Total thrust: ' num2str(hoverControl(2)) ' N']);
21      disp(['    Thrust per rotor: ' num2str(hoverControl(2) / 4) ' N']);
22      disp(['    Hover induced velocity: ' ...
23            num2str(sqrt(hoverControl(2) / (8 * AIR_DENSITY * rotorArea))) ...
24            ' m/s']);
25      disp(['    Total power (computed): ' num2str(dxHover(5)) ' W']);
26
27      disp(' ');
28
29      % Test Case 2: Forward flight
30      xForward = [0; 0; 25; 0; 0];
31      % Calculate required thrust for steady forward flight
32      [lift, drag] = computeLiftDrag(xForward(3), xForward(4), params);
33      uForward = [drag; mass * GRAVITY - lift];
34      disp('Forward Flight Test:');
35      dxForward = evtolDynamics(xForward, uForward, params);
36      disp(['    Lift force: ' num2str(lift) ' N']);
37      disp(['    Drag force: ' num2str(drag) ' N']);
38      disp(['    Required thrust - horizontal: ' ...
39            num2str(uForward(1)) ' N']);
40      disp(['    Required thrust - vertical: ' ...
41            num2str(uForward(2)) ' N']);
42      disp(['    Horizontal acceleration: ' ...
43            num2str(dxForward(3))]);
44      disp(['    Vertical acceleration: ' ...
45            num2str(dxForward(4))]);
46
47      disp(' ');
48
49      % Test Case 3: Climbing flight
50      xClimb = [0; 0; 20; 5; 0];
51      gamma = atan2(5, 20);
52      [lift, drag] = computeLiftDrag(xClimb(3), xClimb(4), params);
53      requiredThrust = [drag * cos(gamma) + lift * sin(gamma); ...
```

```
                        drag * sin(gamma) - lift * cos(gamma) + ...
                        mass * GRAVITY];
    climbControl = requiredThrust;
    disp('Climbing Flight Test:');
    dxClimb = evtolDynamics(xClimb, climbControl, params);
    disp(['    Lift force: ' num2str(lift) ' N']);
    disp(['    Drag force: ' num2str(drag) ' N']);
    disp(['    Required thrust - horizontal: ' ...
          num2str(climbControl(1)) ' N']);
    disp(['    Required thrust - vertical: ' ...
          num2str(climbControl(2)) ' N']);
    disp(['    Horizontal acceleration: ' ...
          num2str(dxClimb(3))]);
    disp(['    Vertical acceleration: ' ...
          num2str(dxClimb(4))]);

    disp(' ');

    % Verify energy model consistency
    % For hover: P = T*v_induced
    hoverThrust = hoverControl(2) / 4;  % Per rotor
    hoverInducedVel = sqrt(hoverThrust / ...
                           (2 * AIR_DENSITY * rotorArea));
    hoverPowerTheoretical = 4 * hoverThrust * hoverInducedVel * ...
                            (1 + chi);
    disp('Energy Model Test:');
    disp(['    Hover power (computed): ' ...
          num2str(dxHover(5)) ' W']);
    disp(['    Hover power (theoretical): ' ...
          num2str(hoverPowerTheoretical) ' W']);
end
```

Código Fonte 46 – Função `testEvtolDynamics`

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	123

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. 123 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**