

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

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

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.
75f.

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. 75f. Trabalho de Conclusão de Curso (Graduação) – Instituto Tecnológico de Aeronáutica, São José dos Campos.

CESSÃO DE DIREITOS

NOME DO AUTOR: Henrique Silva Simplicio

TÍTULO 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, 11 de novembro de 2024.

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

Agradecimentos

“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

Abstract

Lista de Figuras

FIGURA 2.1 – Representações das <i>splines</i> de controle e de estado	22
FIGURA 3.1 – Métodos da classe <code>TrajectoryProblem</code>	24
FIGURA 3.2 – Métodos auxiliares da classe <code>TrajectoryProblem</code>	25
FIGURA 3.3 – Método para a solução do <code>TrajectoryProblem</code> com a colocação tra- pezoidal	25
FIGURA 3.4 – Formato da <code>structure solution</code>	25
FIGURA 3.5 – Diagrama de forças no movimento simples	32
FIGURA 3.6 – Diagrama de forças no braquistócrona	33
FIGURA 3.7 – Diagrama de forças no eVTOL	34
FIGURA 4.1 – Chute inicial para o problema de movimento simples em uma di- mensão.	37
FIGURA 4.2 – Trajetória ótima do problema de movimento simples em uma dimen- são.	37
FIGURA 4.3 – Chute inicial para o problema de braquistócrona.	38
FIGURA 4.4 – Trajetória ótima do problema de braquistócrona.	39
FIGURA 4.5 – Comparação entre a solução obtida e a solução analítica.	39
FIGURA 4.6 – Chute inicial para o problema de trajetória do eVTOL.	40
FIGURA 4.7 – Trajetória ótima do problema de trajetória do eVTOL.	41
FIGURA 4.8 – Comparação entre a solução obtida e a solução de referência.	41
FIGURA 4.9 – Comparação da energia.	42

Lista de Códigos Fonte

1	Modelo de implementação	32
2	Classe <code>TrajectoryProblem</code>	65
3	Função <code>packZ</code>	66
4	Função <code>unpackZ</code>	67
5	Função <code>computeDefects</code>	68
6	Função <code>evaluateConstraints</code>	70
7	Função <code>evaluateObjective</code>	71
8	Função <code>spline2</code>	72
9	Arquivo principal do problema movimento simples	75
10	Função <code>simpleMassDynamics</code>	75

Lista de Abreviaturas e Siglas

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

Sumário

1	INTRODUÇÃO	14
1.1	Motivação	14
1.2	Objetivo	14
1.3	Revisão Bibliográfica	15
2	CONTROLE ÓTIMO	17
2.1	Formulação Geral	17
2.2	Formulação por Colocação Direta	20
2.2.1	Colocação Trapezoidal	21
3	METODOLOGIA	23
3.1	Interface de Dados	23
3.1.1	TrajectoryProblem.m	24
3.1.2	packZ.m	29
3.1.3	unpackZ.m	29
3.1.4	computeDefects.m	29
3.1.5	evaluateConstraints.m	30
3.1.6	evaluateObjective.m	30
3.1.7	spline2.m	30
3.2	Problemas Exemplos	31
3.2.1	Modelo de utilização	31
3.2.2	Movimento simples em uma dimensão	32
3.2.3	Braquistócrona	33
3.2.4	Trajectoria de subida de eVTOL	34

4	RESULTADOS	36
4.1	Movimento simples em uma dimensão	36
4.2	Braquistócrona	37
4.3	Trajatória do eVTOL	39
5	CONCLUSÃO	43
	REFERÊNCIAS	44
	ANEXO A – ARQUIVOS DE CÓDIGO FONTE	46
A.1	Arquivos da biblioteca	46
A.1.1	TrajectoryProblem.m	46
A.1.2	packZ.m	65
A.1.3	unpackZ.m	66
A.1.4	computeDefects.m	67
A.1.5	evaluateConstraints.m	68
A.1.6	evaluateObjective.m	70
A.1.7	spline2.m	71
A.2	Arquivos de modelo	73
A.3	Arquivos do problema movimento simples	73
A.3.1	mainSimpleMass.m	73
A.3.2	simpleMassDynamics.m	75

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 de problemas do tipo, 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, conseqüentemente, 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*:

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

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

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

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

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

às restrições de trajetória

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

às restrições de evento

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

às restrições de ligação de fase

$$\begin{aligned}
\psi_L &\leq \psi[\mathbf{x}^{(1)}(t_0^{(1)}), \mathbf{u}^{(1)}(t_0^{(1)}), \\
&\quad \mathbf{x}^{(1)}(t_f^{(1)}), \mathbf{u}^{(1)}(t_f^{(1)}), \mathbf{p}^{(1)}, t_0^{(1)}, t_f^{(1)}, \\
&\quad \mathbf{x}^{(2)}(t_0^{(2)}), \mathbf{u}^{(2)}(t_0^{(2)}), \\
&\quad \mathbf{x}^{(2)}(t_f^{(2)}), \mathbf{u}^{(2)}(t_f^{(2)}), \mathbf{p}^{(2)}, t_0^{(2)}, t_f^{(2)}, \\
&\quad \vdots \\
&\quad \mathbf{x}^{(N)}(t_0^{(N)}), \mathbf{u}^{(N)}(t_0^{(N)}), \\
&\quad \mathbf{x}^{(N)}(t_f^{(N)}), \mathbf{u}^{(N)}(t_f^{(N)}), \mathbf{p}^{(N)}, t_0^{(N)}, t_f^{(N)}] \leq \psi_U,
\end{aligned} \tag{2.5}$$

às restrições de limite

$$\begin{aligned}
\mathbf{u}_L^{(i)} &\leq \mathbf{u}^{(i)}(t) \leq \mathbf{u}_U^{(i)}, \quad t \in [t_0^{(i)}, t_f^{(i)}], \\
\mathbf{x}_L^{(i)} &\leq \mathbf{x}^{(i)}(t) \leq \mathbf{x}_U^{(i)}, \quad t \in [t_0^{(i)}, t_f^{(i)}], \\
\mathbf{p}_L^{(i)} &\leq \mathbf{p}^{(i)} \leq \mathbf{p}_U^{(i)}, \\
t_{0L}^{(i)} &\leq t_0^{(i)} \leq t_{0U}^{(i)}, \\
t_{fL}^{(i)} &\leq t_f^{(i)} \leq t_{fU}^{(i)},
\end{aligned} \tag{2.6}$$

e às seguintes restrições

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

onde U_Ψ é o domínio da função ψ .

2.2 Formulação por Colocação Direta

Os métodos diretos discretizam o PCO, de forma a convertê-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.

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

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

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

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 $\mathbf{f}_k \equiv \mathbf{f}[\mathbf{x}(t_k), \mathbf{u}(t_k), \mathbf{p}, t_k]$.

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

$$\begin{aligned}
\mathbf{c}_L = \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \\ \vdots \\ \mathbf{0} \\ \mathbf{g}_L \\ \mathbf{g}_L \\ \vdots \\ \mathbf{g}_L \\ \mathbf{e}_L \\ \mathbf{e}_L \\ \psi_L \\ \psi_L \end{bmatrix} \quad \mathbf{c}(\mathbf{x}) = \begin{bmatrix} \zeta_1 \\ \zeta_2 \\ \vdots \\ \zeta_{M-1} \\ \mathbf{g}_1 \\ \mathbf{g}_2 \\ \vdots \\ \mathbf{g}_M \\ \mathbf{e}_0 \\ \mathbf{e}_f \\ \psi_0 \\ \psi_f \end{bmatrix} \quad \mathbf{c}_U = \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \\ \vdots \\ \mathbf{0} \\ \mathbf{g}_U \\ \mathbf{g}_U \\ \vdots \\ \mathbf{g}_U \\ \mathbf{e}_U \\ \mathbf{e}_U \\ \psi_U \\ \psi_U \end{bmatrix} \quad (2.11)
\end{aligned}$$

As primeiras $n_x n_s$ restrições exigem que os vetores ζ 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

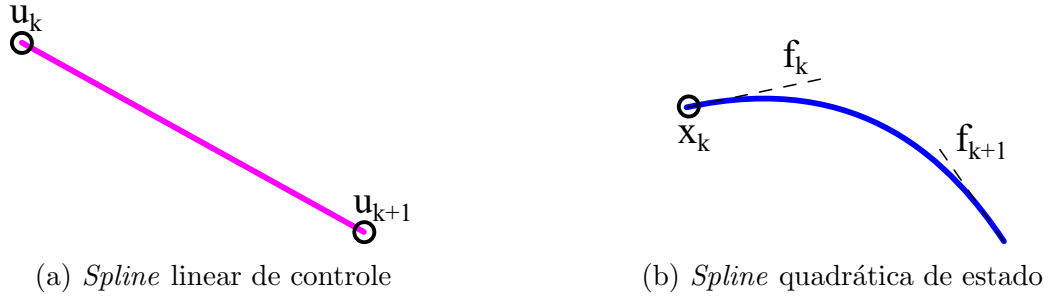
$$\mathbf{u}(t) \approx \mathbf{u}_k + \frac{\tau}{h_k}(\mathbf{u}_{k+1} - \mathbf{u}_k), \quad (2.12)$$

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

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

que, ao ser integrada, resulta em

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

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

Desse modo, as variáveis são definidas como

$$\mathbf{z}^\top = (\mathbf{x}_1, \mathbf{u}_1, \dots, \mathbf{x}_M, \mathbf{u}_M), \quad (2.15)$$

e as restrições

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

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

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

3 Metodologia

Relembrando, o objetivo deste trabalho é criar uma biblioteca no MATLAB que permita a resolução de problemas de otimização de trajetória de maneira facilitada, 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

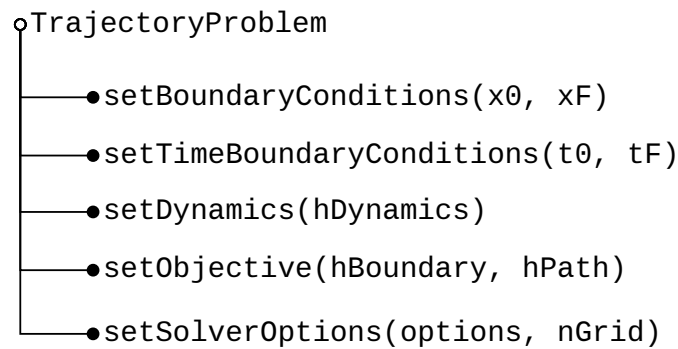
Diferente dos softwares OptimTraj (KELLY, 2022) e PSOPT (BECERRA, 2022), a implementação será realizada utilizando uma abordagem de POO, com uma classe MATLAB principal que conterá métodos para a configuração do problema e cálculo da solução. Além da classe principal, a biblioteca contará com funções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

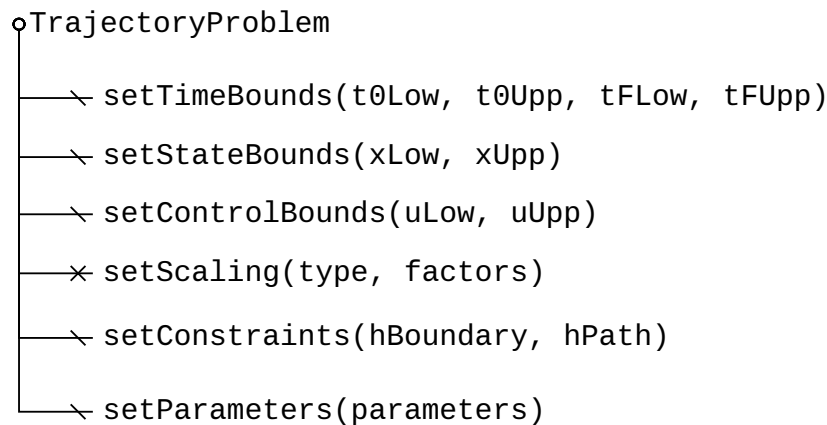
3.1.1 TrajectoryProblem.m

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.

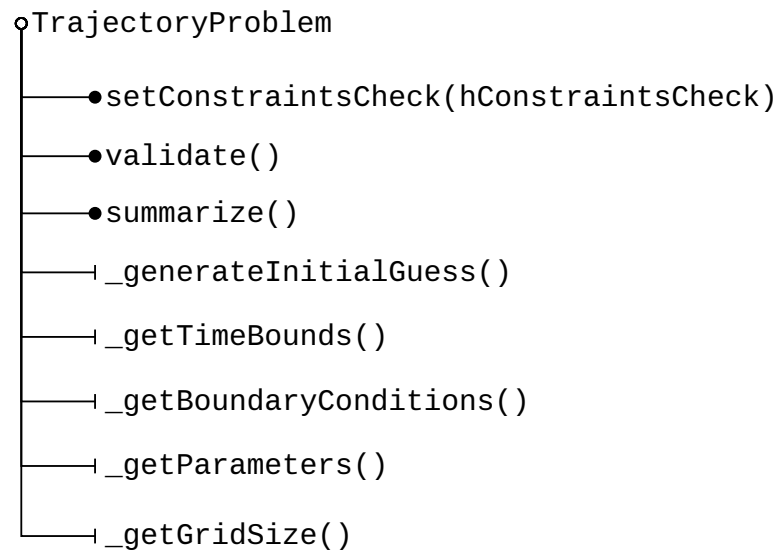


FIGURA 3.2 – Métodos auxiliares da classe TrajectoryProblem

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.

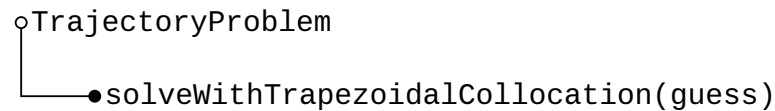
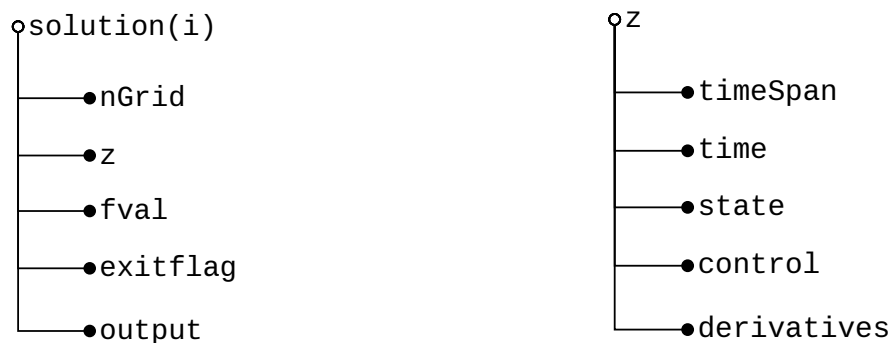


FIGURA 3.3 – Método para a solução do TrajectoryProblem com a colocação trapezoidal

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.

```
1  clear; clc; close all;
2
3  % Adicione o caminho para a classe TrajectoryProblem
4  addpath('..');
5
6  % Crie uma instância do problema
7  nx = 2; % Número de estados
8  nu = 1; % Número de controles
9  problem = TrajectoryProblem(nx, nu);
10
11 % Defina as propriedades básicas do problema
12 problem.setTimeBoundaryConditions(0, 1); % Intervalo de tempo [0,1]
13 problem.setBoundaryConditions(x0, xF); % Condições iniciais e finais
14 problem.setStateBounds(xLow, xUpp); % Limites de estado
15 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.

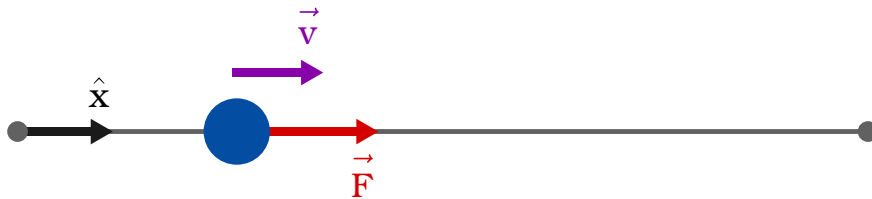


FIGURA 3.5 – Diagrama de forças no movimento simples

$$\mathbf{x} = \begin{bmatrix} s_x \\ v_x \end{bmatrix}, \quad \mathbf{u} = [F], \quad \dot{\mathbf{x}} = \begin{bmatrix} v \\ F/m \end{bmatrix}, \quad (3.1)$$

$$J = \int_{t_0}^{t_f} [F^2] dt. \quad (3.2)$$

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.

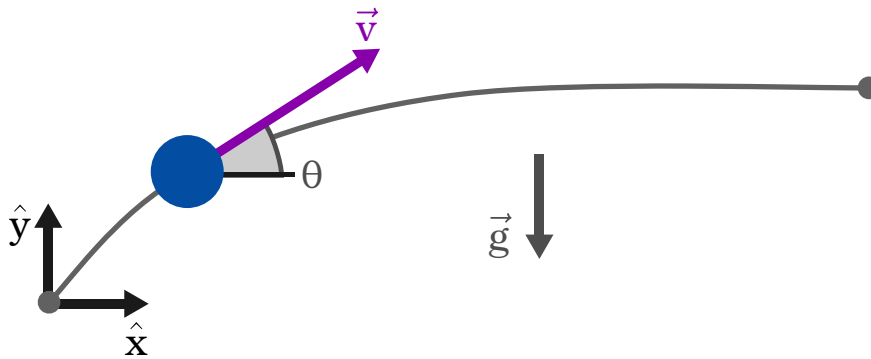


FIGURA 3.6 – Diagrama de forças no braquistócrona

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

$$J = t_f. \quad (3.4)$$

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.

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

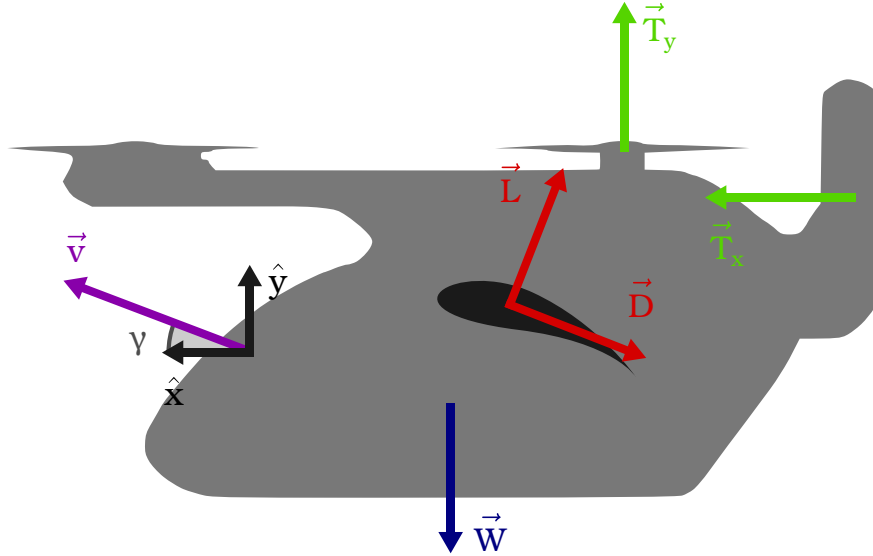


FIGURA 3.7 – Diagrama de forças no eVTOL

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

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

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 $m = 1$ kg, $s_{x,0} = 0$ m, $s_{x,f} = 1$ m, $t_0 = 0$ s e $t_f = 1$ s. Como chute inicial, interpolou-se linearmente os valores iniciais e finais das variáveis de estado e entre os valores 1 N e -1 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.

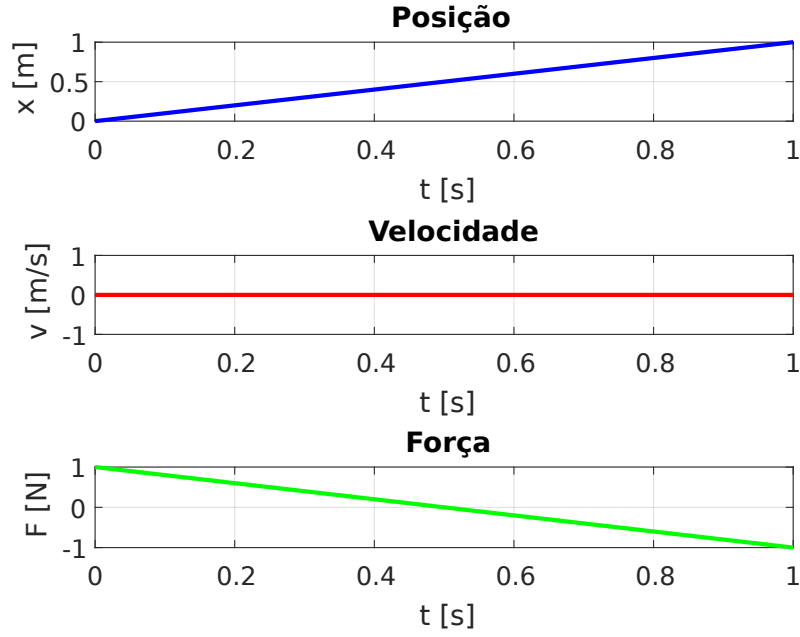


FIGURA 4.1 – Chute inicial para o problema de movimento simples em uma dimensão.

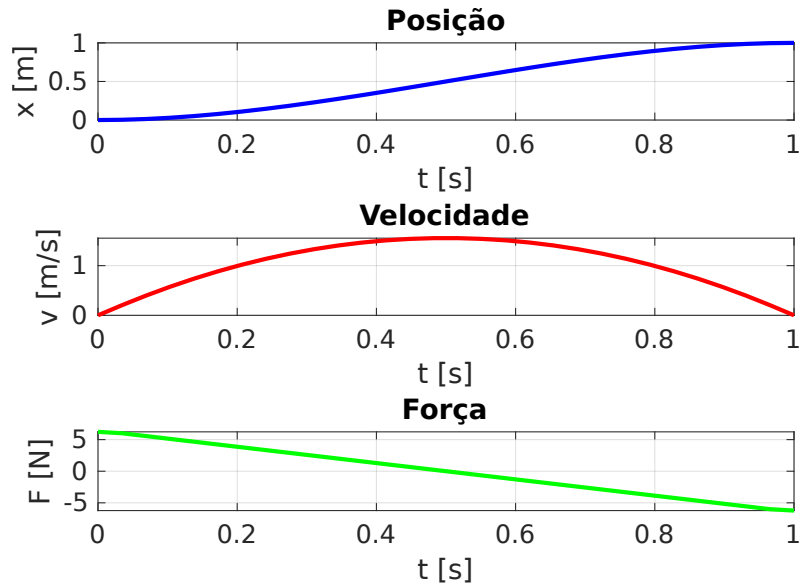


FIGURA 4.2 – Trajetória ótima do problema de movimento simples em uma dimensão.

4.2 Braquistócrona

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

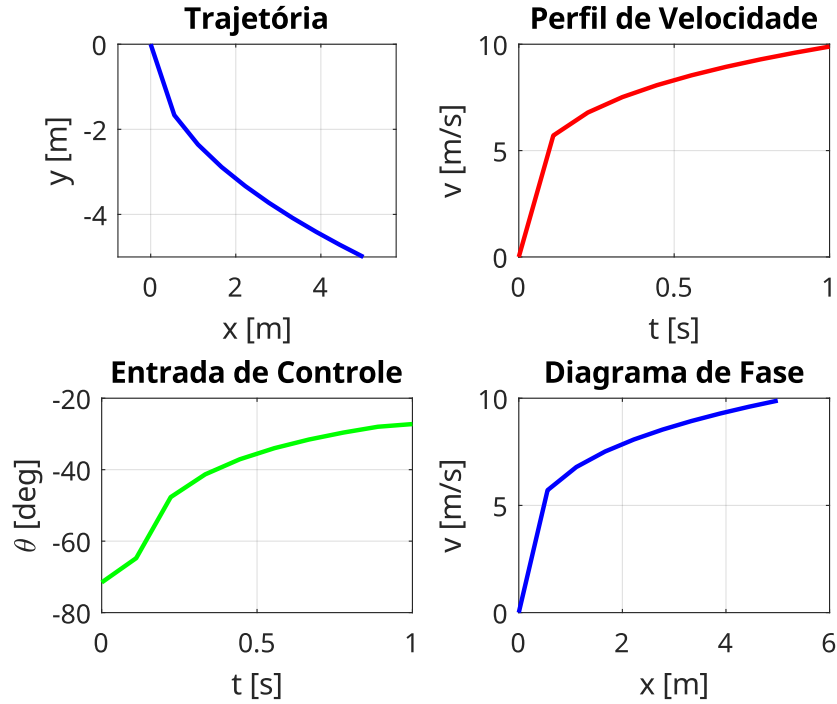


FIGURA 4.3 – Chute inicial para o problema de braquistócrona.

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.

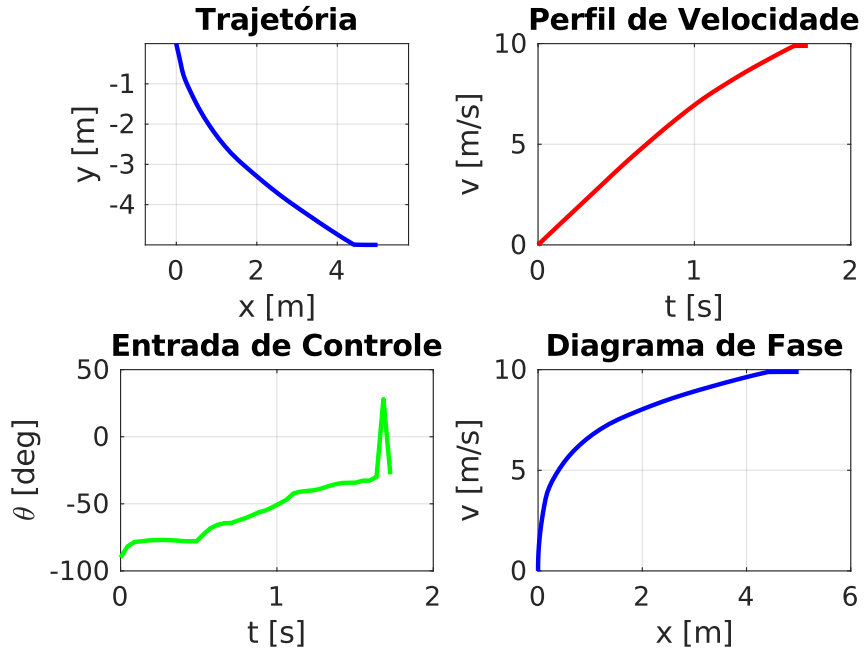


FIGURA 4.4 – Trajetória ótima do problema de braquistócrona.

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

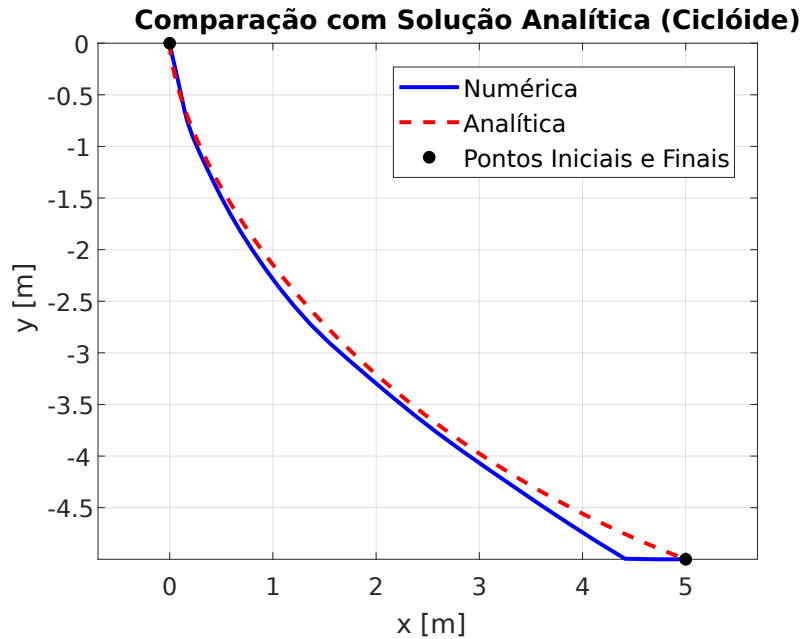


FIGURA 4.5 – Comparação entre a solução obtida e a solução analítica.

4.3 Trajetória do eVTOL

O problema considerado é o caso base do trabalho de referência, utilizando-se os mesmos parâmetros nele apresentados. Também, testou-se a utilização do mesmo chute inicial,

mas sem sucesso. Desse modo, construiu-se um chute inicial que fosse fisicamente plausível utilizando-se um polinômio de terceiro grau para a trajetória e respeitando as relações entre as variáveis de estado e controle, como mostrado na Figura 4.6.

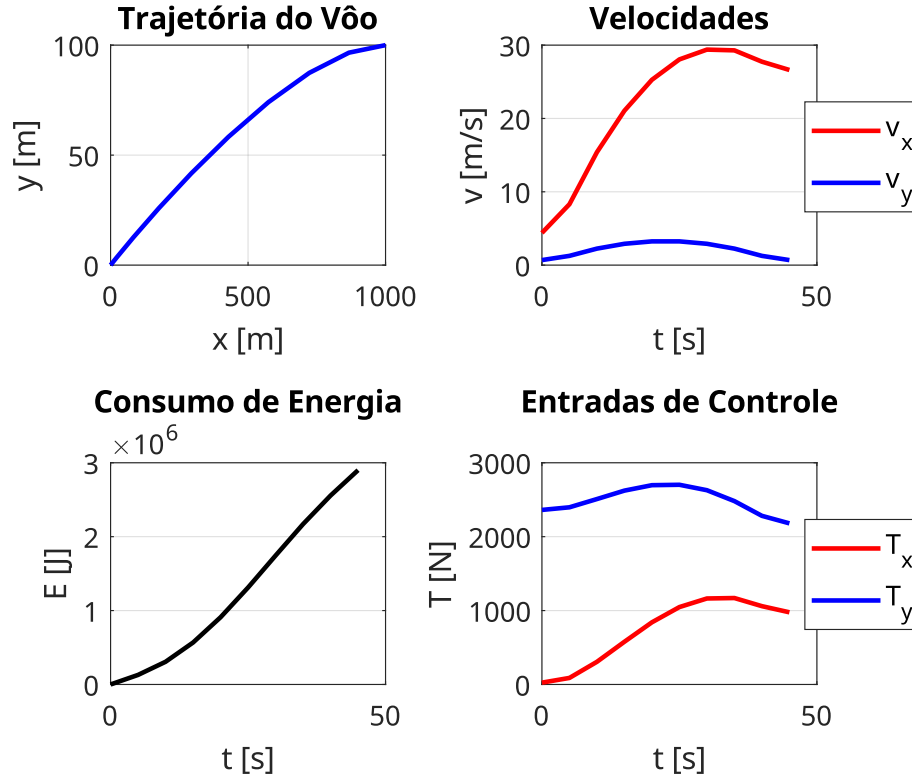


FIGURA 4.6 – Chute inicial para o problema de trajetória do eVTOL.

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

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

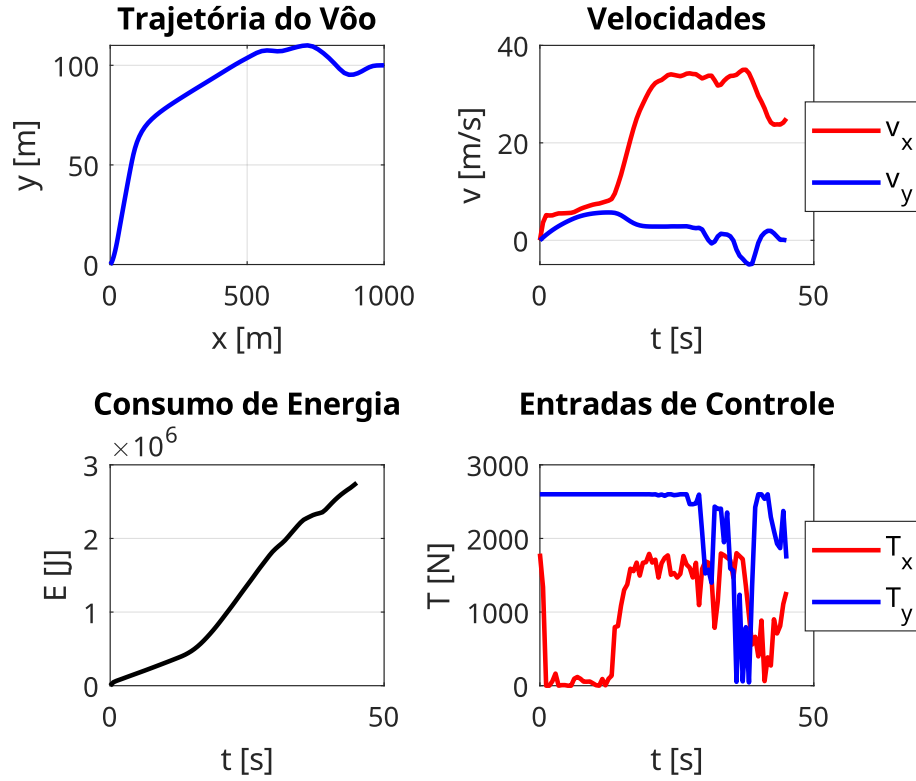


FIGURA 4.7 – Trajetória ótima do problema de trajetória do eVTOL.

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

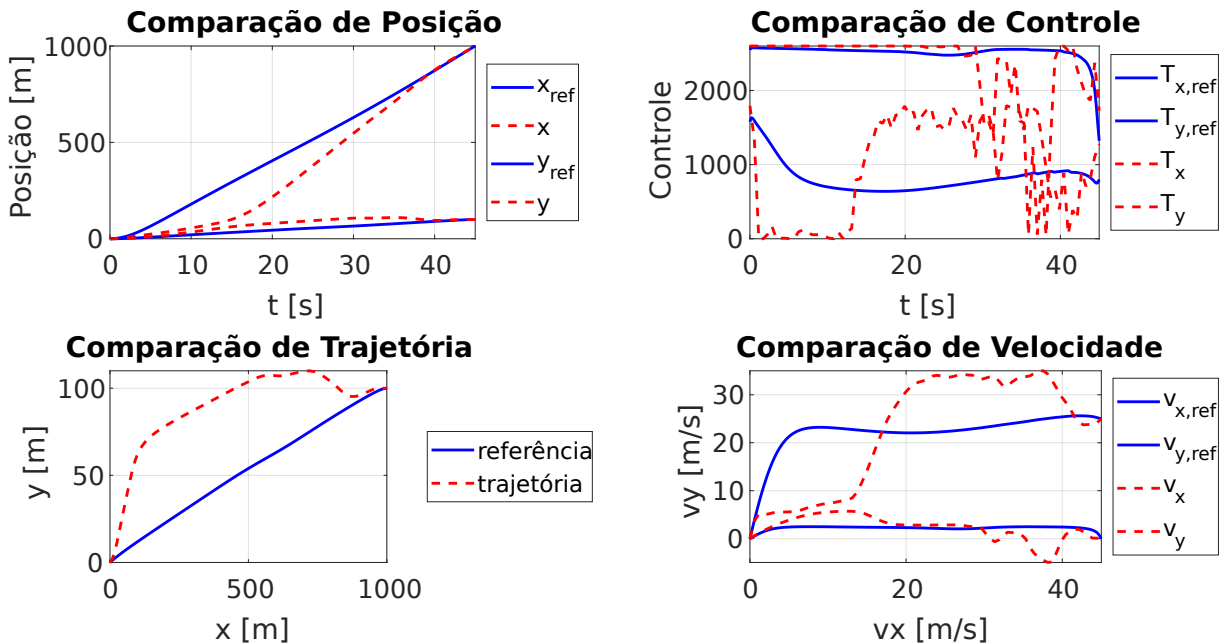


FIGURA 4.8 – Comparação entre a solução obtida e a solução de referência.

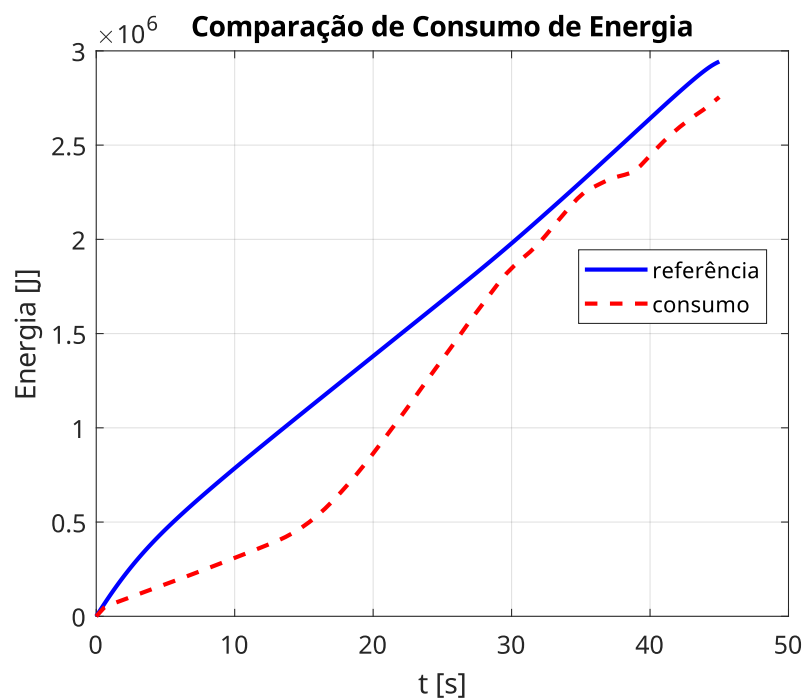


FIGURA 4.9 – Comparação da energia.

5 Conclusão

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

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

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

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

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

Referências

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Anexo A - Arquivos de Código Fonte

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

A.1 Arquivos da biblioteca

A.1.1 TrajectoryProblem.m

```
1  classdef TrajectoryProblem < handle
2      % Class to define a trajectory optimization problem
3
4      properties (SetAccess = private)
5          % Problem dimensions
6          nx % Number of states
7          nu % Number of controls
8
9          % Time boundary conditions
10         t0 {mustBeNumeric} % Initial time
11         tF {mustBeNumeric} % Final time
12         timeSpan = {} % Cell array of time spans
13
14         % Boundary conditions
15         x0 % Initial state
16         xF % Final state
17
18         % Boundary conditions struct
19         boundaryConditions
20
21         % Time bounds
```

```

22         tOLow {mustBeNumeric} % Lower bound for initial time
23         tOUpp {mustBeNumeric} % Upper bound for initial time
24         tFlow {mustBeNumeric} % Lower bound for final time
25         tFUpp {mustBeNumeric} % Upper bound for final time
26
27         % State and control bounds
28         xLow % Lower state bounds
29         xUpp % Upper state bounds
30         uLow % Lower control bounds
31         uUpp % Upper control bounds
32
33         % Function handles
34         dynamics % System dynamics
35         constraints % Constraint function
36         objective % Objective function
37         constraintsCheck % Function handle for problem-specific
38         % constraint checking
39
40         % Constraint functions
41         boundaryConstraints % Boundary constraints function
42         pathConstraints % Path constraints function
43
44         % Objective functions
45         boundaryObjective % Mayer term
46         pathObjective % Lagrange term
47
48         % Parameters
49         parameters % Problem parameters
50
51         % Solver properties
52         nGrid = [] % Array of grid points for each iteration
53         solverOptions = {} % Cell array of options for each
54         % iteration
55
56         % Scaling factors
57         stateScaling
58         controlScaling
59         timeScaling
60         scaling % Struct containing stateScaling, controlScaling,
61         % 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), ...

```

```

302         zeros(obj.nx, 1), ...
303         obj.t0, obj.tF, ...
304         obj.boundaryConditions);
305     if ~isempty(c)
306         validateattributes(c, {'numeric'}, ...
307             {'vector'});
308     end
309     if ~isempty(ceq)
310         validateattributes(ceq, {'numeric'}, ...
311             {'vector'});
312     end
313     catch ME
314         error('Invalid boundary constraints function: %s', ...
315             ME.message);
316     end
317
318     % Scale boundary conditions
319     bndConditions = struct();
320     if isfield(obj.boundaryConditions, 't0')
321         bndConditions.t0 = obj.boundaryConditions.t0 / ...
322             obj.timeScaling;
323     end
324     if isfield(obj.boundaryConditions, 'tF')
325         bndConditions.tF = obj.boundaryConditions.tF / ...
326             obj.timeScaling;
327     end
328     if isfield(obj.boundaryConditions, 'x0')
329         bndConditions.x0 = obj.boundaryConditions.x0 / ...
330             obj.stateScaling;
331     end
332     if isfield(obj.boundaryConditions, 'xF')
333         bndConditions.xF = obj.boundaryConditions.xF / ...
334             obj.stateScaling;
335     end
336     obj.boundaryConstraints = @(x0, xF, t0, tF) ...
337         hBoundaryConstraints(x0, xF, ...
338             t0, tF, ...
339             bndConditions);
340 else
341     obj.boundaryConstraints = [];

```

```

342         end
343
344         % Validate path constraints if provided
345         if ~isempty(hPathConstraints)
346             validateattributes(hPathConstraints, ...
347                               {'function_handle'}, {});
348
349             try
350                 time_test = [obj.t0, obj.tF];
351                 [c, ceq] = hPathConstraints(time_test, ...
352                                             zeros(obj.nx, 2), ...
353                                             zeros(obj.nu, 2), ...
354                                             obj.parameters);
355
356                 if ~isempty(c)
357                     validateattributes(c, {'numeric'}, ...
358                                       {'vector'});
359
360                 end
361                 if ~isempty(ceq)
362                     validateattributes(ceq, {'numeric'}, ...
363                                       {'vector'});
364
365                 end
366             catch ME
367                 error('Invalid path constraints function: %s', ...
368                       ME.message);
369             end
370
371             obj.pathConstraints = @(time, state, control) ...
372                                   hPathConstraints(time, state, ...
373                                                     control, ...
374                                                     obj.parameters);
375
376         else
377             obj.pathConstraints = [];
378         end
379
380         % Create combined constraints function
381         obj.constraints = @(z, packInfo) ...
382                           evaluateConstraints(z, packInfo, ...
383                                               obj.dynamics, ...
384                                               @computeDefects, ...
385                                               obj.pathConstraints, ...
386                                               obj.boundaryConstraints);
387     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, ...

```

```

422                                     {'optim.options.Fmincon'}, ...
423                                     {}), options);
424
425         obj.solverOptions = options;
426     end
427
428     obj.nGrid = nGrid;
429 end
430
431 function setConstraintsCheck(obj, hConstraintsCheck)
432     % Set function handle for problem-specific
433     % constraint checking
434     validateattributes(hConstraintsCheck, ...
435                       {'function_handle'}, {});
436
437     % Test the function with dummy inputs
438     time_test = linspace(obj.t0, obj.tF, 2);
439     state_test = zeros(obj.nx, 2);
440     control_test = zeros(obj.nu, 2);
441     try
442         violations = hConstraintsCheck(time_test, state_test, ...
443                                       control_test, ...
444                                       obj.parameters);
445         validateattributes(violations, {'struct'}, {});
446     catch ME
447         error('Invalid constraints check function: %s', ...
448             ME.message);
449     end
450
451     obj.constraintsCheck = hConstraintsCheck;
452 end
453
454 function valid = validate(obj)
455     % Validate that all required properties are set
456
457     % Check each requirement individually
458     hasDynamics = ~isempty(obj.dynamics);
459     hasObjective = ~isempty(obj.objective);
460     hasFiniteBounds = ~any(isinf([obj.xLow; obj.xUpp; ...
461                                   obj.uLow; obj.uUpp]));
462     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', ...
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

```

```

502         info = struct();
503         info.numStates = obj.nx;
504         info.numControls = obj.nu;
505         info.timespan = [obj.t0, obj.tF];
506         info.stateBounds = [obj.xLow, obj.xUpp];
507         info.controlBounds = [obj.uLow, obj.uUpp];
508         if ~isempty(obj.stateNames)
509             info.stateNames = obj.stateNames;
510         end
511         if ~isempty(obj.controlNames)
512             info.controlNames = obj.controlNames;
513         end
514     end
515
516     function [state, control] = generateInitialGuess(obj)
517         % Generate initial guess with reasonable values for
518         % free final states
519
520         % Validate boundary conditions exist
521         if isempty(obj.x0) || isempty(obj.xF)
522             error(['TrajectoryProblem:NoBoundaryConditions: ', ...
523                 'Either pass initial guess or set', ...
524                 ' boundary conditions to generate a', ...
525                 ' linear interpolation initial guess.']);
526         end
527
528         timeGrid = linspace(obj.t0, obj.tF, obj.nGrid(1));
529
530         % Initialize state array
531         state = zeros(obj.nx, obj.nGrid(1));
532
533         % For each state variable
534         for i = 1:obj.nx
535             if abs(obj.xF(i)) > 1e6 % Detect "free" final states
536                 % Use a reasonable final value instead
537                 % of the large number
538                 if obj.xF(i) > 0
539                     % Use half of upper bound
540                     finalVal = max(obj.x0(i), ...
541                                     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

```



```

702         solution(i).nGrid = obj.nGrid(i);
703         solution(i).z = struct();
704         solution(i).z.timeSpan = obj.timeSpan{i + 1};
705         solution(i).z.time = timeGrid;
706         solution(i).z.state = state;
707         solution(i).z.control = control;
708         solution(i).z.derivatives = ...
709             obj.dynamics(timeGrid, state, control);
710         solution(i).fval = fval;
711         solution(i).exitflag = exitflag;
712         solution(i).output = output;
713
714         % Check constraints if a check function is provided
715         if ~isempty(obj.constraintsCheck)
716             solution(i).violations = ...
717                 obj.constraintsCheck(timeGrid, ...
718                     solution(i).z.state, ...
719                     solution(i).z.control, ...
720                     obj.parameters);
721         end
722     end
723 end
724
725 function [t0, tF] = getTimeBounds(obj)
726     % Get time bounds
727     t0 = obj.t0;
728     tF = obj.tF;
729 end
730
731 function [x0, xF] = getBoundaryConditions(obj)
732     % Get boundary conditions
733     x0 = obj.x0;
734     xF = obj.xF;
735 end
736
737 function parameters = getParameters(obj)
738     % Get problem parameters
739     parameters = obj.parameters;
740 end
741

```

```

742         function nGrid = gridSize(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 = [

```

```

28         tScaled(:)
29         stateScaled(:)
30         controlScaled(:)
31     ];
32 end

```

Código Fonte 3 – Função packZ

A.1.3 unpackZ.m

```

1  function [time, state, control] = unpackZ(z, packInfo, scalingFlag)
2      % Unpack optimization vector into components
3      % Inputs:
4      %   z: [nz,1] = decision variables
5      %   packInfo: struct = information about the problem
6      %   scalingFlag: logical = flag to indicate if scaling
7      %                   should be applied
8      % Outputs:
9      %   time: [1,nGrid] = time vector
10     %   state: [nx,nGrid] = state variables
11     %   control: [nu,nGrid] = control variables
12
13     % Unpack packInfo
14     nx = packInfo.nx;
15     nu = packInfo.nu;
16     nGrid = packInfo.nGrid;
17
18     % Calculate expected sizes
19     nStates = nx * nGrid;
20     nControls = nu * nGrid;
21     expectedLength = 2 + nStates + nControls; % accounts for time bounds
22
23     % Validate z length
24     assert(length(z) == expectedLength, ...
25         'z length mismatch. Expected %d elements but got %d', ...
26         expectedLength, length(z));
27
28     % Unpack optimization vector into components
29     tSpan = z(1:2);

```

```

30     z_rest = z(3:end);
31
32     % Reshape state and control trajectories
33     nStates = nx * nGrid;
34     state = reshape(z_rest(1:nStates), [nx, nGrid]);
35     control = reshape(z_rest(nStates + 1:end), [nu, nGrid]);
36
37     % Apply scaling up if requested
38     if nargin >= 3 && scalingFlag
39         tSpan = tSpan * packInfo.scaling.timeScaling;
40         state = state .* repmat(packInfo.scaling.stateScaling, ...
41                                1, nGrid);
42         control = control .* repmat(packInfo.scaling.controlScaling, ...
43                                    1, nGrid);
44     end
45
46     % Create time vector
47     time = linspace(tSpan(1), tSpan(2), nGrid);
48 end

```

Código Fonte 4 – Função unpackZ

A.1.4 computeDefects.m

```

1  function defects = computeDefects(timeStep, state, stateDerivatives)
2      % This function computes the defects for
3      % direct transcription using the Trapezoidal Rule.
4      %
5      % Inputs:
6      %   timeStep: scalar = time step
7      %   state: [nStates, nGrid] = matrix of states
8      %   stateDerivatives: [nStates, nGrid-1] = matrix of
9      %                       state derivatives
10     %
11     % Outputs:
12     %   defects: [nStates, nGrid-1] = matrix of defects
13
14     nGrid = size(state, 2);
15

```

```

16     indexLower = 1:(nGrid - 1);
17     indexUpper = 2:nGrid;
18
19     stateLower = state(:, indexLower);
20     stateUpper = state(:, indexUpper);
21
22     derivativesLower = stateDerivatives(:, indexLower);
23     derivativesUpper = stateDerivatives(:, indexUpper);
24
25     % Apply the Trapezoidal Rule
26     defects = stateUpper - stateLower - ...
27             0.5 * timeStep * (derivativesLower + derivativesUpper);
28 end

```

Código Fonte 5 – Função computeDefects

A.1.5 evaluateConstraints.m

```

1  function [c, ceq] = evaluateConstraints(z, packInfo, ...
2                                     dynamics, ...
3                                     defectConstraints, ...
4                                     pathConstraints, ...
5                                     boundaryConstraints)
6
7      % Define the collocation constraints
8      % Inputs:
9      %   z: [nz,1] = decision variables
10     %   packInfo: struct = information about the problem
11     %   dynamics: function = dynamics function
12     %   defectConstraints: function = defect constraints function
13     %   pathConstraints: function = path constraints function
14     %   boundaryConstraints: function = boundary constraints function
15     % Outputs:
16     %   c = [m,1] = inequality constraints
17     %   ceq = [m,1] = equality constraints
18
19     % Check for NaN values in z
20     if any(isnan(z))
21         error('NaN values detected in decision variables');
22     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
62         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), ...

```

```

25         time(end));
26     J = J + phi;
27 end
28
29 % Add Lagrange term if provided
30 if ~isempty(pathObjective)
31     % Trapezoidal integration of the path objective
32     integrand = pathObjective(time, state, control);
33     L = trapz(time, integrand);
34     J = J + L;
35 end
36 end

```

Código Fonte 7 – Função evaluateObjective

A.1.7 spline2.m

```

1  function [yInterp, dyInterp] = spline2(tOld, yOld, dyOld, tNew)
2      % Second-order interpolation using the formula:
3      %  $x(t) \sim x_k + f_k \tau + (\tau^2/2h_k)(f_{k+1} - f_k)$ 
4      % where tau is local time and h_k is the time step
5      %
6      % Inputs:
7      %   tOld: old time vector [1, nOld]
8      %   yOld: old state matrix [nStates, nOld]
9      %   dyOld: old derivative matrix [nStates, nOld]
10     %   tNew: new time vector [1, nNew]
11     %
12     % Outputs:
13     %   yInterp: interpolated state matrix [nStates, nNew]
14     %   dyInterp: interpolated derivative matrix [nStates, nNew]
15
16     nStates = size(yOld, 1);
17     yInterp = zeros(nStates, length(tNew));
18     dyInterp = zeros(nStates, length(tNew));
19
20     for i = 1:length(tNew)
21         t = tNew(i);
22

```



```

23      % Find the interval containing t
24      idx = find(tOld(1:end - 1) <= t & t <= tOld(2:end), 1);
25
26      if isempty(idx)
27          if t < tOld(1)
28              idx = 1;
29          else
30              idx = length(tOld) - 1;
31          end
32      end
33
34      % Get local time step
35      hk = tOld(idx + 1) - tOld(idx);
36
37      % Local time tau
38      tau = t - tOld(idx);
39
40      % Get values and derivatives (now matrices)
41      xk = yOld(:, idx);
42      fk = dyOld(:, idx);
43      fk1 = dyOld(:, idx + 1);
44
45      % Apply the formula (element-wise operations)
46      yInterp(:, i) = xk + fk * tau + ...
47                      (tau^2 / (2 * hk)) * (fk1 - fk);
48      dyInterp(:, i) = fk + (tau / hk) * (fk1 - fk);
49  end
50 end

```

Código Fonte 8 – Função spline2

A.2 Arquivos de modelo

A.3 Arquivos do problema movimento simples

A.3.1 mainSimpleMass.m

```
1  % Simple mass moving in 1D
2  clear;
3  clc;
4  close all;
5
6  % Add path to the TrajectoryProblem class
7  addpath('..');
8
9  %% Create problem instance
10 nx = 2; % States: [x; v] (position, velocity)
11 nu = 1; % Control: [F] (force)
12 problem = TrajectoryProblem(nx, nu);
13
14 %% Set time bounds
15 t0Low = 0;
16 t0Upp = 0;
17 tFLow = 0;
18 tFUpp = 1;
19 problem.setTimeBounds(t0Low, t0Upp, tFLow, tFUpp);
20
21 %% Set time boundary conditions
22 t0 = 0;
23 tF = 1;
24 problem.setTimeBoundaryConditions(t0, tF);
25
26 %% Set state bounds
27 xLow = [-10; -5]; % Position and velocity lower bounds
28 xUpp = [10; 5]; % Position and velocity upper bounds
29 problem.setStateBounds(xLow, xUpp);
30
31 %% Set control bounds
32 uLow = [-50]; % Maximum force in negative direction
33 uUpp = [50]; % Maximum force in positive direction
```

```

34 problem.setControlBounds(uLow, uUpp);
35
36 %% Set parameters
37 params.MASS = 1.0; % Mass of the object
38 problem.setParameters(params);
39
40 %% Set boundary conditions
41 x0 = [0; 0]; % Start at origin with zero velocity
42 xF = [1; 0]; % End at x=1 with zero velocity
43 problem.setBoundaryConditions(x0, xF);
44
45 %% Set scaling
46 % problem.setScaling('state', [1; 1]); % States are already O(1)
47 % problem.setScaling('control', 1); % Control is already O(1)
48 % problem.setScaling('time', 2); % Time is already O(1)
49
50 %% Set functions
51 problem.setDynamics(@simpleMassDynamics);
52 problem.setObjective(@boundaryObjective, @pathObjective);
53 problem.setConstraints(@boundaryConstraints, @pathConstraints);
54
55 %% Set solver options
56 nGrid = [30]; % Start with very few grid points
57 options = optimoptions('fmincon');
58 options.Display = 'iter';
59 options.MaxFunEvals = 1e5;
60 % options.Algorithm = 'sqp';
61 % options.EnableFeasibilityMode = true;
62 % options.SubproblemAlgorithm = 'cg';
63 % options.OptimalityTolerance = 1e-8;
64 % options.ConstraintTolerance = 1e-8;
65 % options.StepTolerance = 1e-8;
66
67 problem.setSolverOptions(options, nGrid);
68
69 %% Generate initial guess
70 zGuess = generateSimpleMassGuess(problem);
71
72 %% Solve the problem
73 solution = problem.solveWithTrapezoidalCollocation(zGuess);

```

```

74
75  %% 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 9 – 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 10 – Função simpleMassDynamics

FOLHA DE REGISTRO DO DOCUMENTO			
1. CLASSIFICAÇÃO/TIPO TC	2. DATA 11 de novembro de 2024	3. DOCUMENTO Nº DCTA/ITA/DM-111/2024	4. Nº DE PÁGINAS 75
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: 1; 2; 3			
10. APRESENTAÇÃO: Trabalho de Graduação, ITA, São José dos Campos, 2024. 75 páginas.		(X) Nacional () Internacional	
11. RESUMO:			
12. GRAU DE SIGILO: (X) OSTENSIVO () RESERVADO () SECRETO			