

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.
134f.

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.

1. Otimização de trajetória. 2. Controle ótimo. 3. Aeronaves de decolagem vertical. 4. Controle.
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. 134f. 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, 14 de novembro de 2024.

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

Agradecimentos

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

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

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

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

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

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

“Agir conforme aquilo que se fala, alinhar discurso e prática, além de ser uma postura ética, é um sinal de autenticidade.”

— MARIO SERGIO CORTELLA

Resumo

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

Abstract

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

Lista de Figuras

FIGURA 2.1 – Representações das <i>splines</i> de controle e de estado	26
FIGURA 3.1 – Métodos da classe <code>TrajectoryProblem</code>	28
FIGURA 3.2 – Métodos auxiliares da classe <code>TrajectoryProblem</code>	29
FIGURA 3.3 – Método para a solução do <code>TrajectoryProblem</code> com a colocação tra- pezoidal	29
FIGURA 3.4 – Formato da <code>structure solution</code>	29
FIGURA 3.5 – Diagrama de forças no movimento simples	36
FIGURA 3.6 – Diagrama de forças no pêndulo invertido	37
FIGURA 3.7 – Diagrama de forças na manobra de mudança de faixa	38
FIGURA 3.8 – Diagrama de forças no braquistócrona	40
FIGURA 3.9 – Diagrama de forças no eVTOL	41
FIGURA 4.1 – Chute inicial para o problema de movimento simples em uma di- mensão.	43
FIGURA 4.2 – Trajetória ótima do problema de movimento simples em uma dimen- são.	43
FIGURA 4.3 – Trajetória ótima do problema de pêndulo invertido.	44
FIGURA 4.4 – Trajetória ótima do problema de manobra de mudança de faixa. . .	45
FIGURA 4.5 – Chute inicial para o problema de braquistócrona.	45
FIGURA 4.6 – Trajetória ótima do problema de braquistócrona.	46
FIGURA 4.7 – Comparação entre a solução obtida e a solução analítica.	47
FIGURA 4.8 – Chute inicial para o problema de trajetória do eVTOL.	48
FIGURA 4.9 – Trajetória ótima do problema de trajetória do eVTOL.	49

FIGURA 4.10 – Comparação entre a solução obtida e a solução de referência.	49
FIGURA 4.11 – Comparação da energia.	50

Lista de Códigos Fonte

1	Modelo de implementação	36
2	Classe TrajectoryProblem	73
3	Função packZ	74
4	Função unpackZ	75
5	Função computeDefects	76
6	Função evaluateConstraints	78
7	Função evaluateObjective	79
8	Função spline2	80
9	Arquivo principal do template	83
10	Função templateDynamics do template	83
11	Função templateParams do template	84
12	Função boundaryConstraints do template	84
13	Função pathConstraints do template	85
14	Função boundaryObjective do template	86
15	Função pathObjective do template	86
16	Arquivo principal do problema movimento simples	89
17	Função simpleMassDynamics do problema movimento simples	89
18	Função boundaryConstraints do problema movimento simples	90
19	Função pathConstraints do problema movimento simples	90
20	Função boundaryObjective do problema movimento simples	91
21	Função pathObjective do problema movimento simples	91
22	Função generateSimpleMassGuess do problema movimento simples	92
23	Arquivo principal do problema do pêndulo invertido	94
24	Função pendulumDynamics do problema do pêndulo invertido	94
25	Função pathObjective do problema do pêndulo invertido	94
26	Arquivo principal do problema da manobra de mudança de faixa	96
27	Função carDynamics do problema da manobra de mudança de faixa	97
28	Função boundaryConstraints do problema da manobra de mudança de faixa	97
29	Função pathObjective do problema da manobra de mudança de faixa	98

30	Arquivo principal do problema da braquistócrona	101
31	Função <code>brachistochroneDynamics</code> do problema da braquistócrona	102
32	Função <code>brachistochroneParams</code> do problema da braquistócrona	102
33	Função <code>boundaryConstraints</code> do problema da braquistócrona	103
34	Função <code>pathConstraints</code> do problema da braquistócrona	104
35	Função <code>boundaryObjective</code> do problema da braquistócrona	104
36	Função <code>pathObjective</code> do problema da braquistócrona	105
37	Função <code>generateBrachistochroneGuess</code> do problema da braquistócrona .	106
38	Função <code>checkConstraints</code> do problema da braquistócrona	109
39	Função <code>compareWithAnalytical</code> do problema da braquistócrona	110
40	Arquivo principal do problema do eVTOL	114
41	Função <code>evtolDynamics</code> do problema do eVTOL	116
42	Função <code>computeLiftDrag</code> do problema do eVTOL	118
43	Função <code>computeInducedVelocity</code> do problema do eVTOL	119
44	Função <code>computeFlightAngle</code> do problema do eVTOL	120
45	Função <code>evtolParams</code> do problema do eVTOL	121
46	Função <code>boundaryConstraints</code> do problema do eVTOL	123
47	Função <code>pathConstraints</code> do problema do eVTOL	124
48	Função <code>boundaryObjective</code> do problema do eVTOL	124
49	Função <code>pathObjective</code> do problema do eVTOL	125
50	Função <code>physicalInitialGuess</code> do problema do eVTOL	127
51	Função <code>checkConstraints</code> do problema do eVTOL	129
52	Função <code>testAeroForces</code> do problema do eVTOL	132
53	Função <code>testEvtolDynamics</code> do problema do eVTOL	134

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	18
1.1	Motivação	18
1.2	Objetivo	18
1.3	Revisão Bibliográfica	19
2	CONTROLE ÓTIMO	21
2.1	Formulação Geral	21
2.2	Formulação por Colocação Direta	24
2.2.1	Colocação Trapezoidal	25
3	METODOLOGIA	27
3.1	Interface de Dados	27
3.1.1	TrajectoryProblem.m	28
3.1.2	packZ.m	33
3.1.3	unpackZ.m	33
3.1.4	computeDefects.m	33
3.1.5	evaluateConstraints.m	34
3.1.6	evaluateObjective.m	34
3.1.7	spline2.m	34
3.2	Problemas Exemplos	35
3.2.1	Modelo de utilização	35
3.2.2	Movimento simples em uma dimensão	36
3.2.3	Pêndulo invertido	37
3.2.4	Manobra de mudança de faixa	38

3.2.5	Braquistócrona	39
3.2.6	Trajectoria de subida de eVTOL	40
4	RESULTADOS	42
4.1	Movimento simples em uma dimensão	42
4.2	Pêndulo invertido	43
4.3	Manobra de mudança de faixa	44
4.4	Braquistócrona	45
4.5	Trajectoria do eVTOL	47
5	CONCLUSÃO	51
	REFERÊNCIAS	52
	ANEXO A – ARQUIVOS DE CÓDIGO FONTE	54
A.1	Arquivos da biblioteca	54
A.1.1	TrajectoryProblem.m	54
A.1.2	packZ.m	73
A.1.3	unpackZ.m	74
A.1.4	computeDefects.m	75
A.1.5	evaluateConstraints.m	76
A.1.6	evaluateObjective.m	78
A.1.7	spline2.m	79
A.2	Arquivos de modelo	80
A.2.1	mainTemplate.m	80
A.2.2	templateDynamics.m	83
A.2.3	templateParams.m	83
A.2.4	boundaryConstraints.m	84
A.2.5	pathConstraints.m	85
A.2.6	boundaryObjective.m	85
A.2.7	pathObjective.m	86
A.3	Arquivos do problema movimento simples	87

A.3.1	mainSimpleMass.m	87
A.3.2	simpleMassDynamics.m	89
A.3.3	boundaryConstraints.m	89
A.3.4	pathConstraints.m	90
A.3.5	boundaryObjective.m	90
A.3.6	pathObjective.m	91
A.3.7	generateSimpleMassGuess.m	91
A.4	Arquivos do problema do pêndulo invertido	92
A.4.1	mainPendulumSwingup.m	92
A.4.2	pendulumDynamics.m	94
A.4.3	pathObjective.m	94
A.5	Arquivos do problema da manobra de mudança de faixa	95
A.5.1	mainLaneChange.m	95
A.5.2	carDynamics.m	96
A.5.3	boundaryConstraints.m	97
A.5.4	pathObjective.m	98
A.6	Arquivos do problema da braquistócrona	98
A.6.1	mainBrachistochrone.m	98
A.6.2	brachistochroneDynamics.m	101
A.6.3	brachistochroneParams.m	102
A.6.4	boundaryConstraints.m	102
A.6.5	pathConstraints.m	103
A.6.6	boundaryObjective.m	104
A.6.7	pathObjective.m	104
A.6.8	generateBrachistochroneGuesses.m	105
A.6.9	checkConstraints.m	107
A.6.10	compareWithAnalytical.m	109
A.7	Arquivos do problema do eVTOL	111
A.7.1	mainEvtol.m	111

A.7.2	evtolDynamics.m	114
A.7.3	computeLiftDrag.m	116
A.7.4	computeInducedVelocity.m	118
A.7.5	computeFlightAngle.m	119
A.7.6	evtolParams.m	120
A.7.7	boundaryConstraints.m	121
A.7.8	pathConstraints.m	123
A.7.9	boundaryObjective.m	124
A.7.10	pathObjective.m	125
A.7.11	physicalInitialGuess.m	125
A.7.12	checkConstraints.m	127
A.7.13	testAeroForces.m	129
A.7.14	testEvtolDynamics.m	132

1 Introdução

1.1 Motivação

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

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

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

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

1.2 Objetivo

Este trabalho tem como objetivo a implementação de uma biblioteca no MATLAB, que permita a solução de problemas de otimização de trajetória utilizando um método de co-

locação direta, mais especificamente a colocação trapezoidal. Inspirando-se nos softwares PSOPT (BECERRA, 2022) e OptimTraj (KELLY, 2022), tal biblioteca deverá simplificar a resolução desse tipo de problema, de modo que o usuário deverá apenas modelizar o problema e entrar com os parâmetros necessários por meio de uma interface de dados. Assim, não haverá necessidade de implementar a lógica matemática por trás da solução numérica para obtenção dos resultados pertinentes.

1.3 Revisão Bibliográfica

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

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

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

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

Com o advento do computador digital, tornou-se viável a solução de sistemas dinâmicos mais complexos e, 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

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

e às seguintes restrições

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

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

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

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

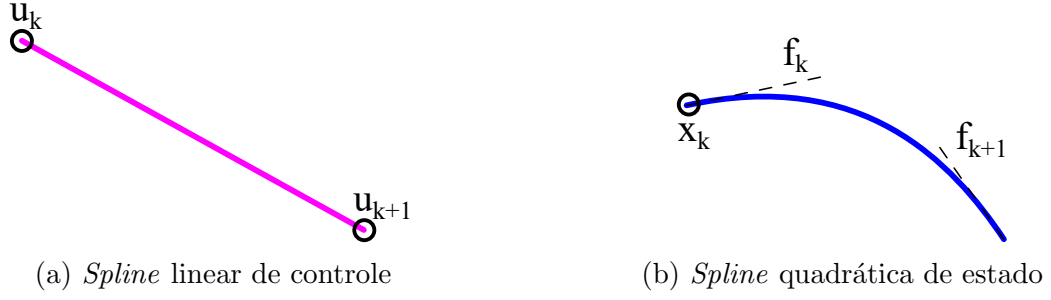
$$\mathbf{u}(t) \approx \mathbf{u}_k + \frac{\tau}{h_k}(\mathbf{u}_{k+1} - \mathbf{u}_k), \quad (2.12)$$

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

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

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 forma simplificada, de forma que o usuário precise apenas modelar o problema, sem a necessidade de implementar a lógica da solução numérica. Para isso, é necessário que a biblioteca forneça uma interface de dados, de modo a permitir que o usuário entre com a modelagem do seu problema. Além disso, deve-se implementar a colocação trapezoidal como método de solução.

Para validação da implementação a ser realizada, foram implementadas as soluções para alguns problemas exemplos. Tais problemas incluem o estudo de otimização de trajetória de um movimento simples em uma dimensão, o problema de pêndulo invertido, o problema de manobra de mudança de faixa e a braquistócrona, que são problemas clássicos na literatura de otimização de trajetórias, além do 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

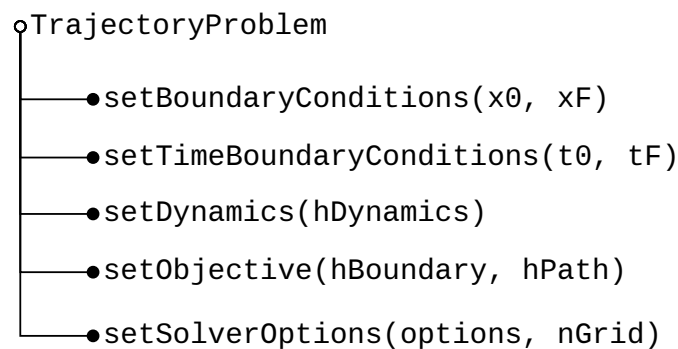
Diferente dos softwares OptimTraj (KELLY, 2022) e PSOPT (BECERRA, 2022), a implementação será realizada utilizando uma abordagem de POO, com uma classe MATLAB principal que conterá métodos para a configuração do problema e cálculo da solução. Além da classe principal, a biblioteca contará com funções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

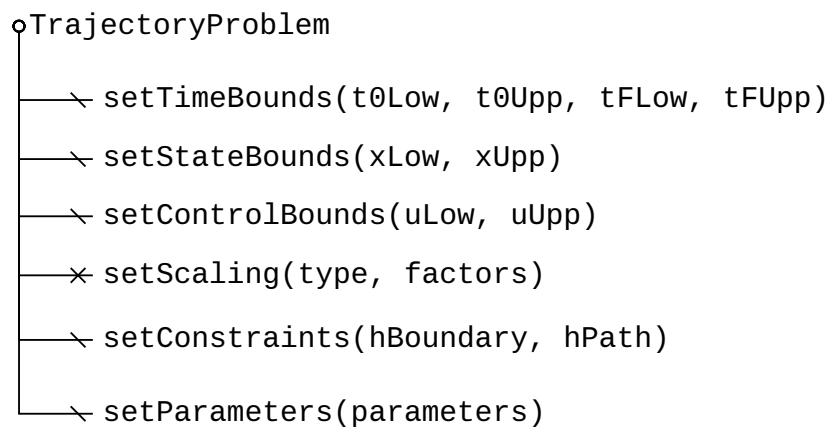
3.1.1 TrajectoryProblem.m

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.



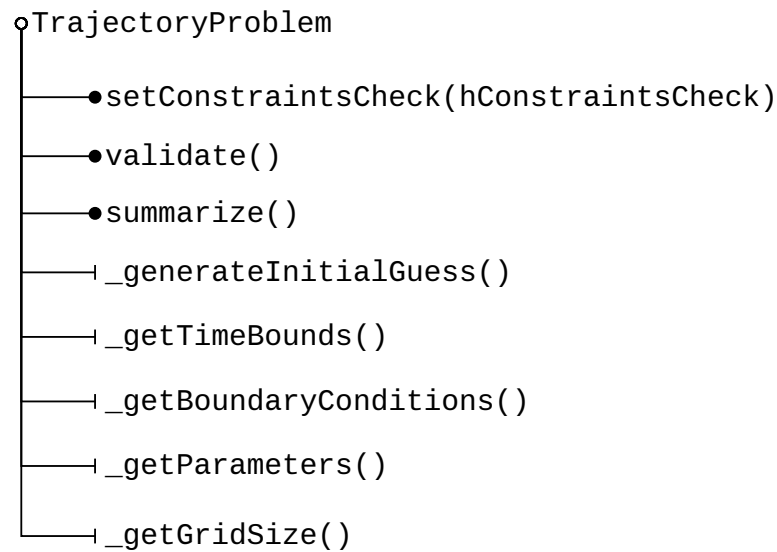
(a) Métodos obrigatórios



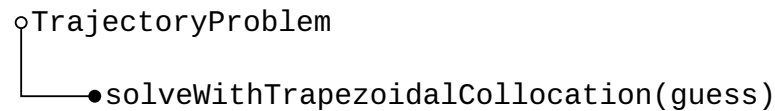
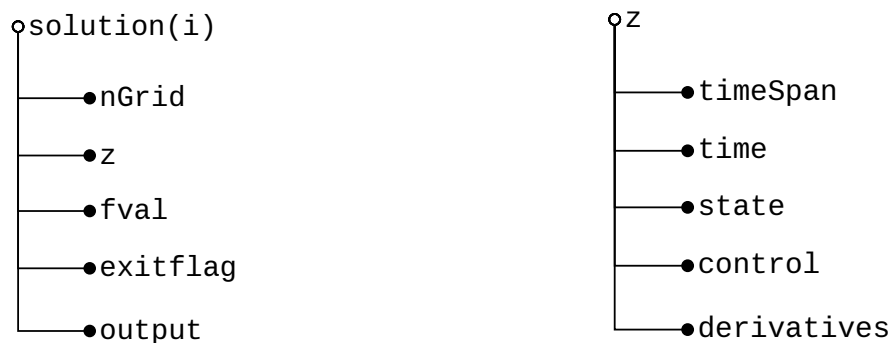
(b) Métodos opcionais

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 trapezoidalFIGURA 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 3.2.5, e um problema de minimização de consumo energético de um veículo multirrotor, na Seção 3.2.6.

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.

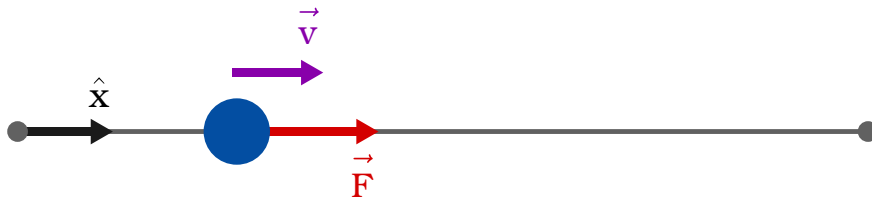


FIGURA 3.5 – Diagrama de forças no movimento simples

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

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

3.2.3 Pêndulo invertido

O problema do pêndulo invertido envolve o controle de um pêndulo do seu ponto de equilíbrio estável (pendendo para baixo) até seu ponto de equilíbrio instável (apontando para cima), minimizando o esforço de controle. Este é um problema clássico em teoria de controle e robótica.

O sistema consiste em um pêndulo de comprimento l e massa m , afetado pela gravidade g e amortecimento b , conforme Figura 3.6.

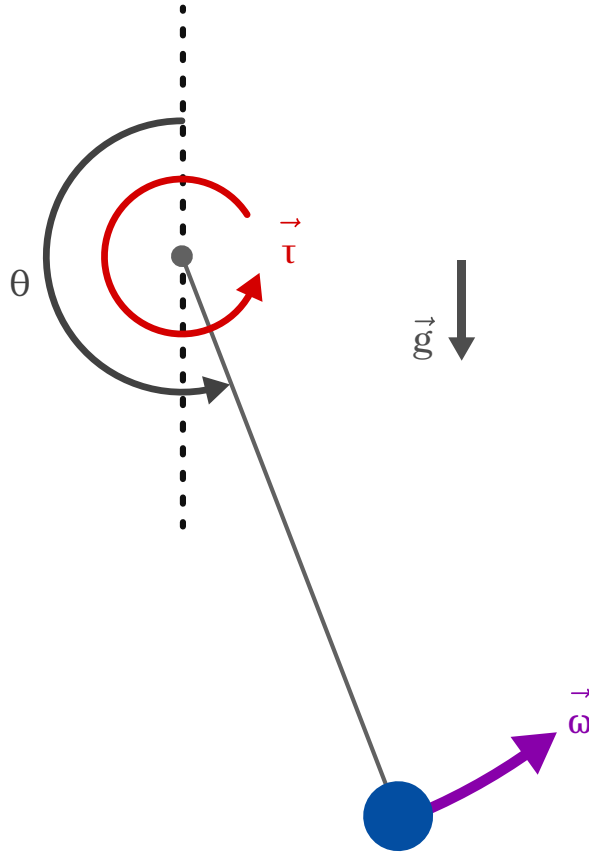


FIGURA 3.6 – Diagrama de forças no pêndulo invertido

Os vetores de estado, controle e dinâmica do sistema são descritos pelas Equações 3.3.

$$\mathbf{x} = \begin{bmatrix} \theta \\ \omega \end{bmatrix}, \quad \mathbf{u} = \begin{bmatrix} \tau \end{bmatrix}, \quad \dot{\mathbf{x}} = \begin{bmatrix} \omega \\ \frac{\tau - mgl \sin(\theta) - b\omega}{ml^2} \end{bmatrix}, \quad (3.3)$$

onde θ é o ângulo em relação à vertical para cima, ω é a velocidade angular e τ é o torque aplicado.

Por fim, o problema consiste em encontrar a trajetória que minimiza o esforço de controle, conforme Equação 3.4.

$$J = \int_0^{t_f} \tau^2 dt \quad (3.4)$$

sujeito às condições de contorno descritas pelas Equações 3.5.

$$\underbrace{\mathbf{x}(0) = \begin{bmatrix} \pi \\ 0 \end{bmatrix}}_{\text{pendendo para baixo em repouso}} \quad \underbrace{\mathbf{x}(t_f) = \begin{bmatrix} 0 \\ 0 \end{bmatrix}}_{\text{apontando para cima em repouso}} \quad (3.5)$$

3.2.4 Manobra de mudança de faixa

O problema de mudança de faixa envolve encontrar uma trajetória ótima para um veículo mudar de faixa enquanto mantém o conforto (minimizando a aceleração) e mantendo-se próximo a uma velocidade longitudinal desejada, conforme ilustrado na Figura 3.7.

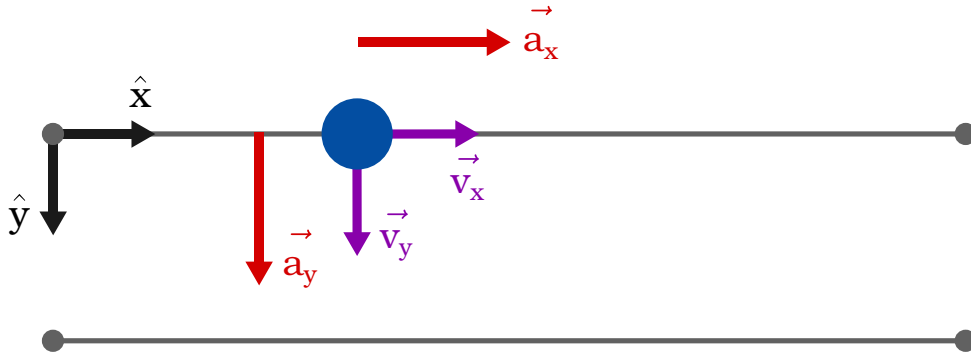


FIGURA 3.7 – Diagrama de forças na manobra de mudança de faixa

O sistema modela um veículo se movendo em um plano 2D, onde consideramos sua posição e velocidade nas direções longitudinal (x) e lateral (y), conforme Equação 3.6.

$$\mathbf{x} = \begin{bmatrix} x \\ y \\ v_x \\ v_y \end{bmatrix}, \quad \mathbf{u} = \begin{bmatrix} a_x \\ a_y \end{bmatrix}, \quad \dot{\mathbf{x}} = \begin{bmatrix} v_x \\ v_y \\ a_x \\ a_y \end{bmatrix}, \quad (3.6)$$

onde x é a posição longitudinal, y é a posição lateral, v_x é a velocidade longitudinal, v_y é a velocidade lateral, a_x é a aceleração longitudinal e a_y é a aceleração lateral.

Por fim, o problema consiste em encontrar a trajetória que minimiza a combinação da minimização da aceleração (conforto) e a manutenção da velocidade alvo, conforme Equação 3.7.

$$J = \int_0^{t_f} [w_1(a_x^2 + a_y^2) + w_2(v_x - v_{alvo})^2] dt \quad (3.7)$$

onde w_1 é o peso para minimização da aceleração e w_2 é o peso para manutenção da velocidade alvo, sendo v_{alvo} a velocidade longitudinal desejada. O problema é sujeito às condições de contorno descritas pelas Equações 3.8.

$$\underbrace{\mathbf{x}(0) = \begin{bmatrix} 0 \\ 0 \\ v_{alvo} \\ 0 \end{bmatrix}}_{\text{início na faixa esquerda}} \quad \underbrace{\mathbf{x}(t_f) = \begin{bmatrix} x_f \\ w_{faixa} \\ v_{alvo} \\ 0 \end{bmatrix}}_{\text{fim na faixa direita}} \quad (3.8)$$

onde w_{faixa} é a largura da faixa.

3.2.5 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 3.8. 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 3.9.

Por fim, o problema consiste em encontrar a trajetória que minimiza o tempo de descida entre dois pontos, conforme Equação 3.10.

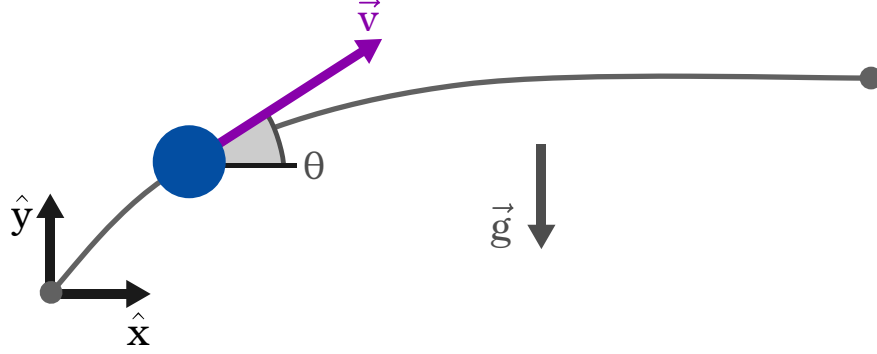


FIGURA 3.8 – Diagrama de forças no braquistócrona

$$\mathbf{x} = \begin{bmatrix} s_x \\ s_y \\ v \end{bmatrix}, \quad \mathbf{u} = [\theta], \quad \dot{\mathbf{x}} = \begin{bmatrix} v \cos(\theta) \\ v \sin(\theta) \\ -g \sin(\theta) \end{bmatrix}, \quad (3.9)$$

$$J = t_f. \quad (3.10)$$

3.2.6 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 3.11.

$$\mathbf{x} = \begin{bmatrix} s_x \\ s_y \\ v_x \\ v_y \\ E \end{bmatrix}, \quad \mathbf{u} = \begin{bmatrix} T_x \\ T_y \end{bmatrix}, \quad (3.11)$$

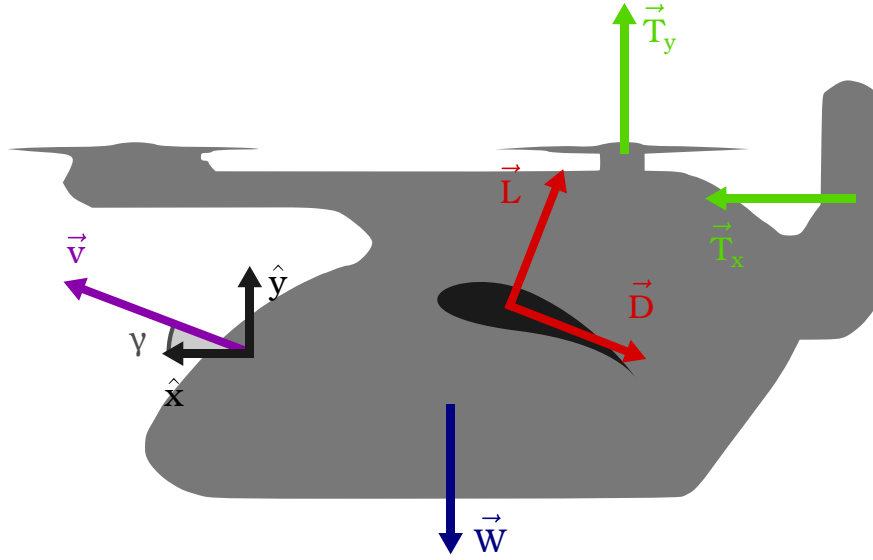


FIGURA 3.9 – Diagrama de forças no eVTOL

Conforme a Figura 3.9, a dinâmica do sistema pode ser descrita pelas Equações 3.12.

$$\begin{aligned} m\ddot{p}_x &= F_x = T_x - D(\mathbf{v}) \cos(\gamma) - L(\mathbf{v}) \sin(\gamma) \\ m\ddot{p}_y &= F_y = T_y - D(\mathbf{v}) \sin(\gamma) + L(\mathbf{v}) \cos(\gamma) - mg, \end{aligned} \quad (3.12)$$

de modo que as restrições diferenciais escrevem-se

$$\begin{aligned} \dot{s}_x &= v_x \\ \dot{s}_y &= v_y & \dot{E} &= P_{total} \\ \dot{v}_x &= \frac{1}{m} F_x, & &= (2P_{ind,rotor,x}(1 + \chi) + T_x|\mathbf{v}| \cos(-\gamma)) \\ \dot{v}_y &= \frac{1}{m} F_y, & &+ (4P_{ind,rotor,y}(1 + \chi) + T_y|\mathbf{v}| \sin(-\gamma)). \end{aligned} \quad (3.13)$$

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

$$J = E_f. \quad (3.14)$$

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 3.2.5; e, por fim, os resultados do problema de trajetória do eVTOL na Seção 3.2.6. 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.

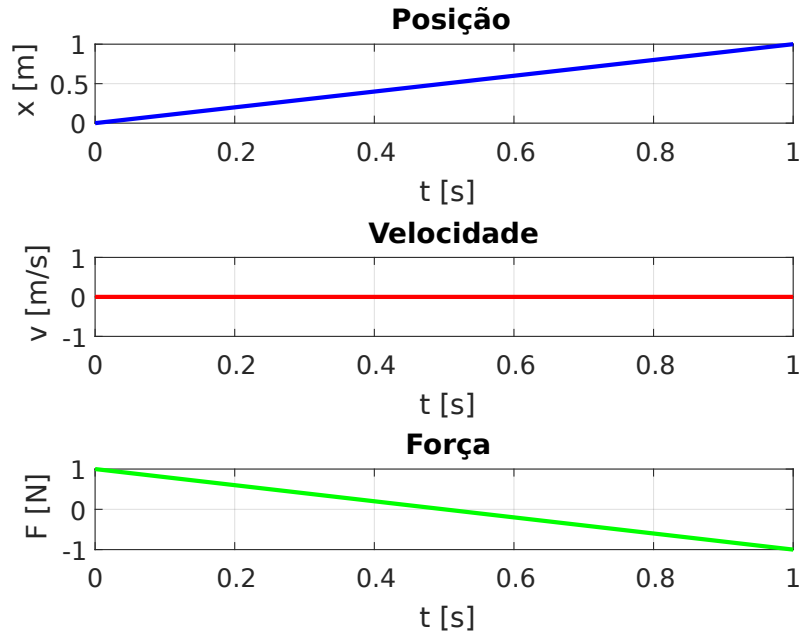


FIGURA 4.1 – Chute inicial para o problema de movimento simples em uma dimensão.

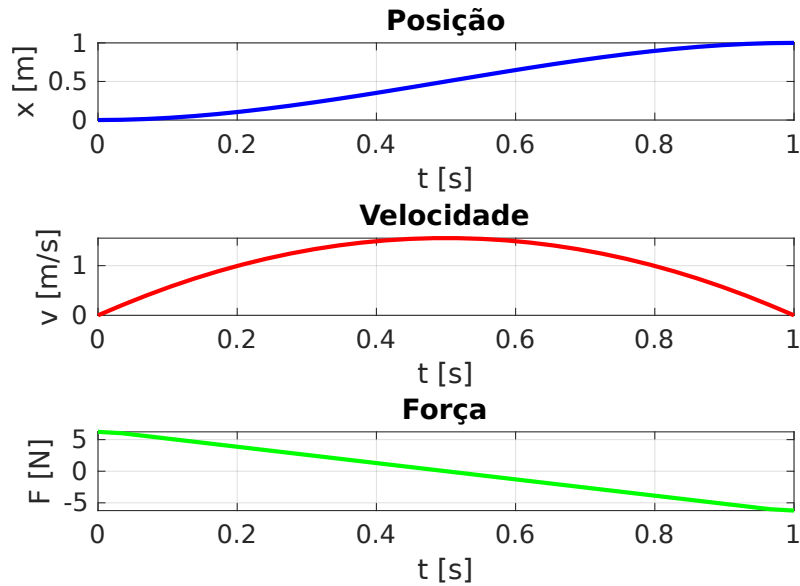


FIGURA 4.2 – Trajetória ótima do problema de movimento simples em uma dimensão.

4.2 Pêndulo invertido

Para a resolução deste problema, utilizou-se $m = 1$ kg, $l = 0,5$ m, $b = 0,1$ N s m⁻¹ e $g = 9,81$ m s⁻² como parâmetros. Não foi necessário utilizar um chute inicial elaborado, uma vez que o solver encontrou a solução ótima sem dificuldades.

Como opções do solver, utilizou-se uma sequência de refinamento da malha, com $n = [30, 60]$ pontos de colocação. Além disso, apenas aumentou-se o número máximo de avaliações da função por iteração para que o solver convergisse para a solução ótima.

O resultado é apresentado na Figura 4.3.

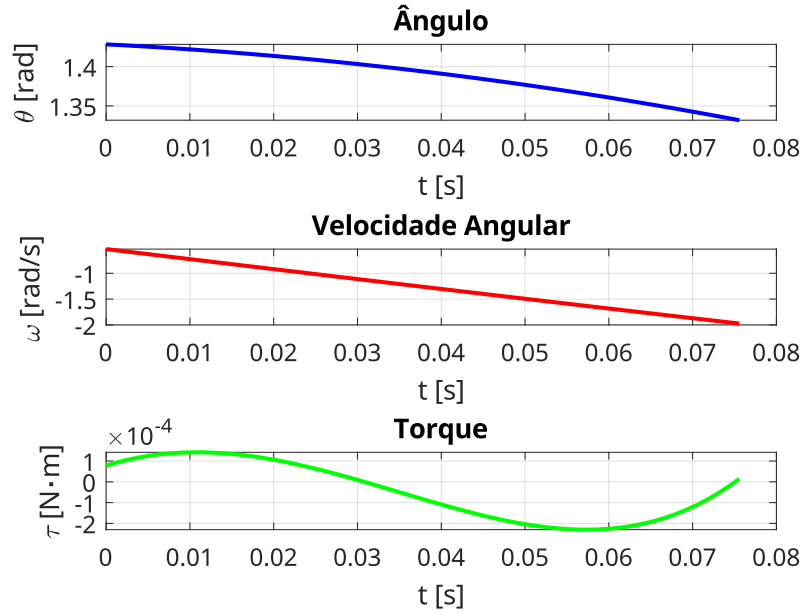


FIGURA 4.3 – Trajetória ótima do problema de pêndulo invertido.

4.3 Manobra de mudança de faixa

Como parâmetros para a resolução deste problema, utilizou-se $a_{max} = 2 \text{ m s}^{-2}$, $v_{alvo} = 15 \text{ m s}^{-1}$, $w_{faixa} = 3,5 \text{ m}$ e $x_f = 75 \text{ m}$ como parâmetros. Também não foi necessário utilizar um chute inicial elaborado, uma vez que o solver encontrou a solução ótima sem dificuldades.

Como opções do solver, utilizou-se uma sequência de refinamento da malha, com $n = [20, 40, 80]$ pontos de colocação. Além disso, assim como no problema do pêndulo invertido, apenas ajustou-se o número máximo de avaliações da função por iteração para que o solver convergisse para a solução ótima. O resultado é apresentado na Figura 4.4.

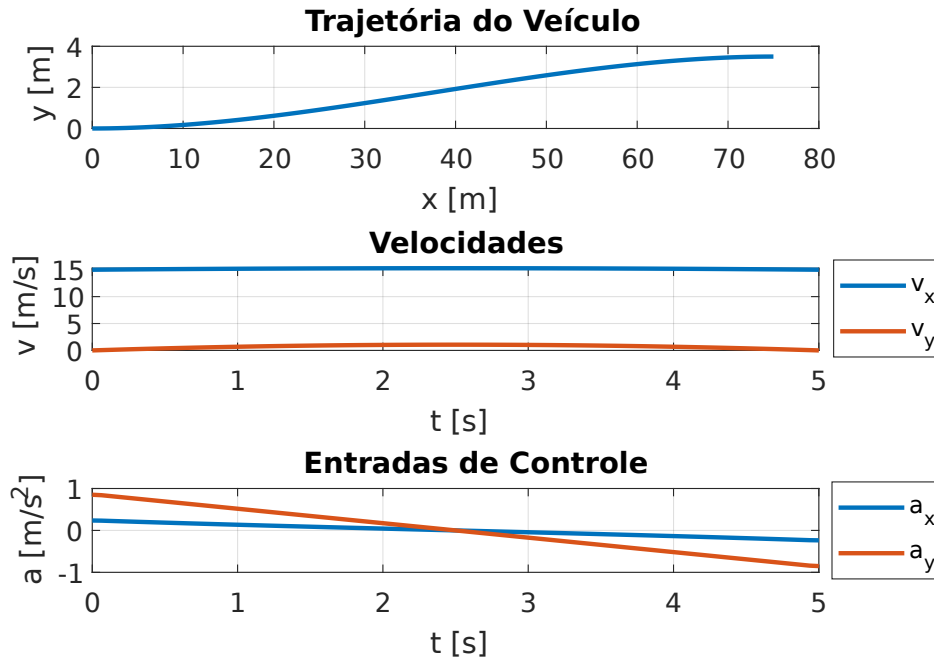


FIGURA 4.4 – Trajetória ótima do problema de manobra de mudança de faixa.

4.4 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 4.5.

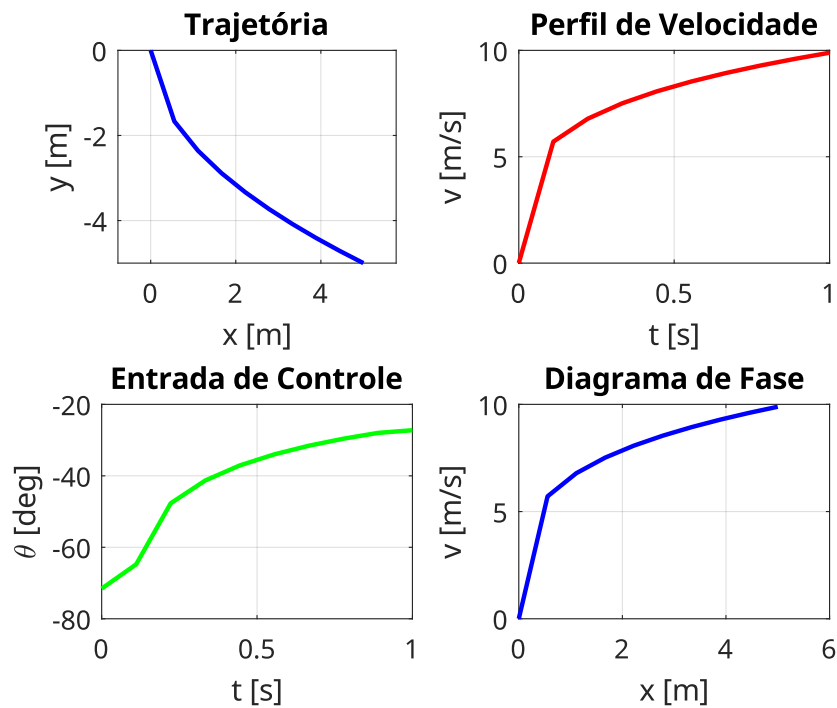


FIGURA 4.5 – Chute inicial para o problema de braquistócrona.

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 4.6.

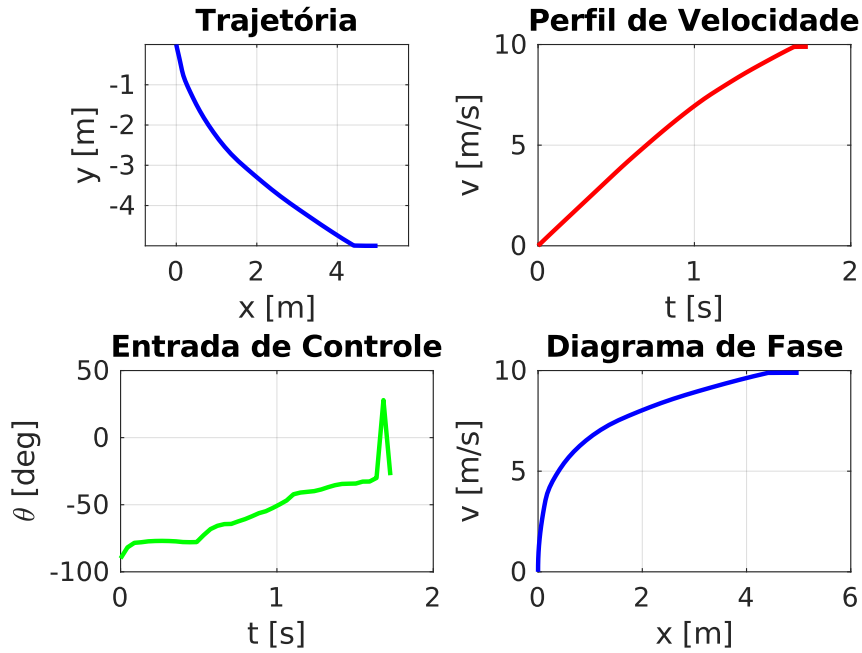


FIGURA 4.6 – Trajetória ótima do problema de braquistócrona.

Na Figura 4.7 é apresentada a comparação entre a solução obtida e a solução analítica do problema, a qual é uma cicloide.

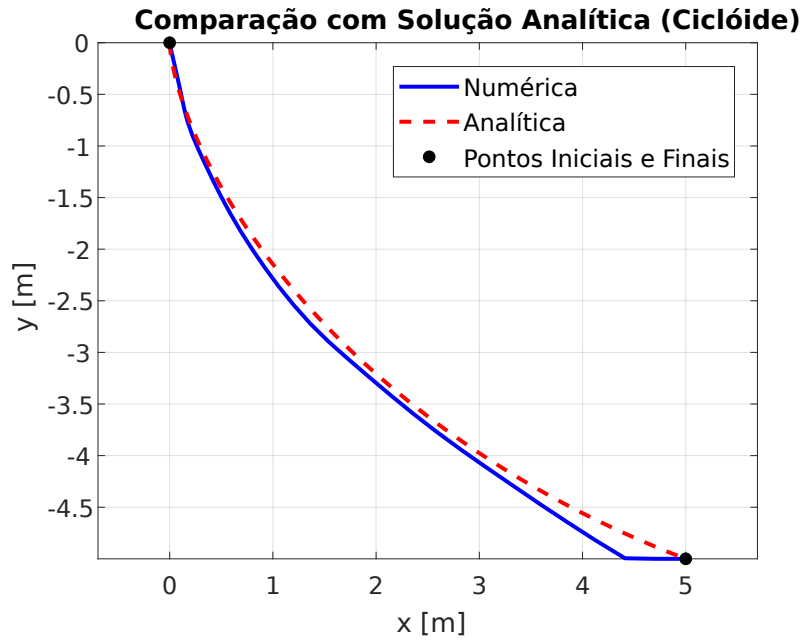


FIGURA 4.7 – Comparação entre a solução obtida e a solução analítica.

4.5 Trajetória do eVTOL

O problema considerado é o caso base do trabalho de referência, utilizando-se os mesmos parâmetros nele apresentados. Testou-se também a utilização do mesmo chute inicial, mas sem sucesso. Desse modo, construiu-se um chute inicial fisicamente plausível utilizando-se um polinômio de terceiro grau para a trajetória e respeitando as relações entre as variáveis de estado e controle, como mostrado na Figura 4.8.

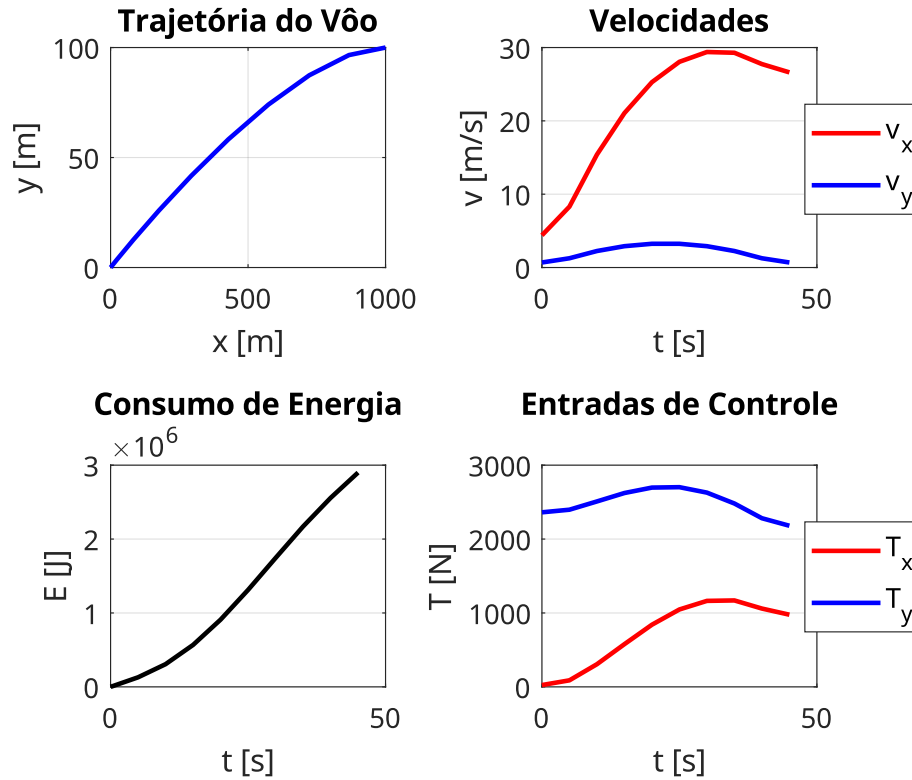


FIGURA 4.8 – Chute inicial para o problema de trajetória do eVTOL.

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 4.9.

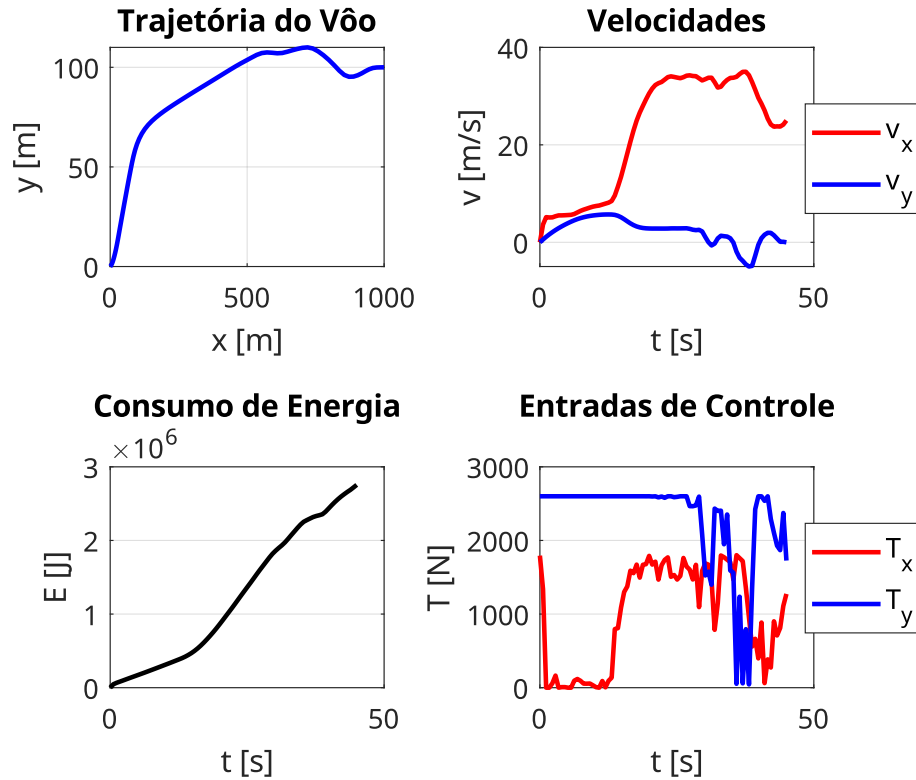


FIGURA 4.9 – Trajetória ótima do problema de trajetória do eVTOL.

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 $E = 2,75484$ MJ para a solução obtida comparado a $E = 2,94342$ MJ da referência. A Figura 4.10 mostra a comparação entre a solução obtida e a solução de referência.

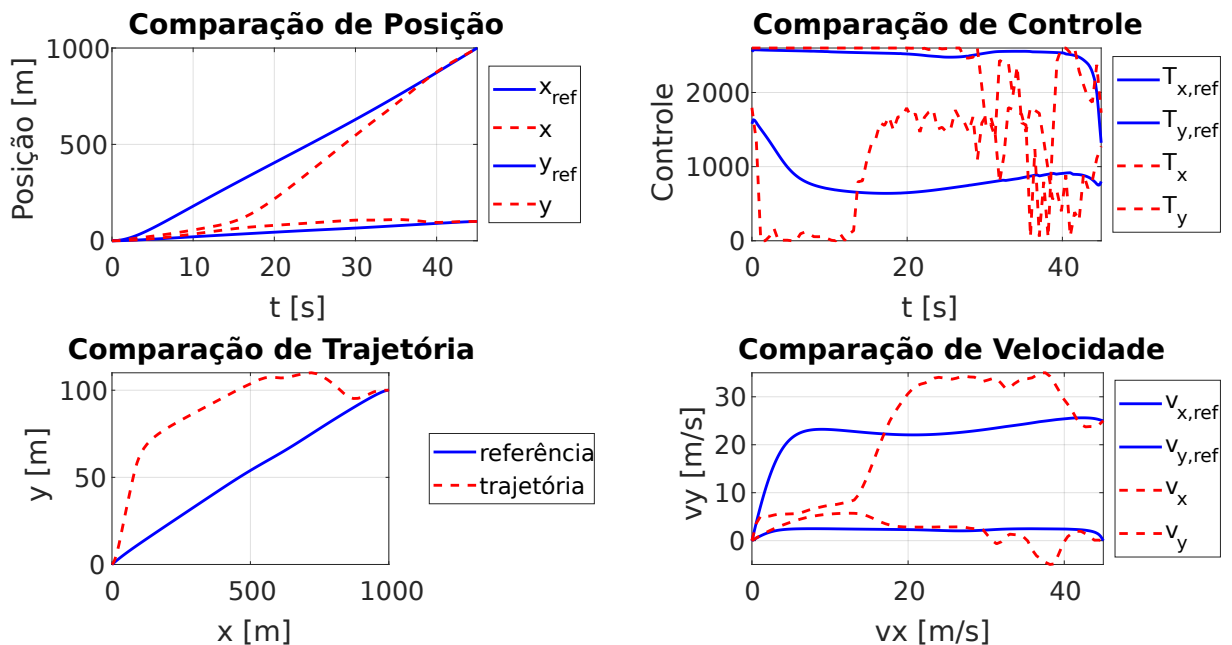


FIGURA 4.10 – Comparação entre a solução obtida e a solução de referência.

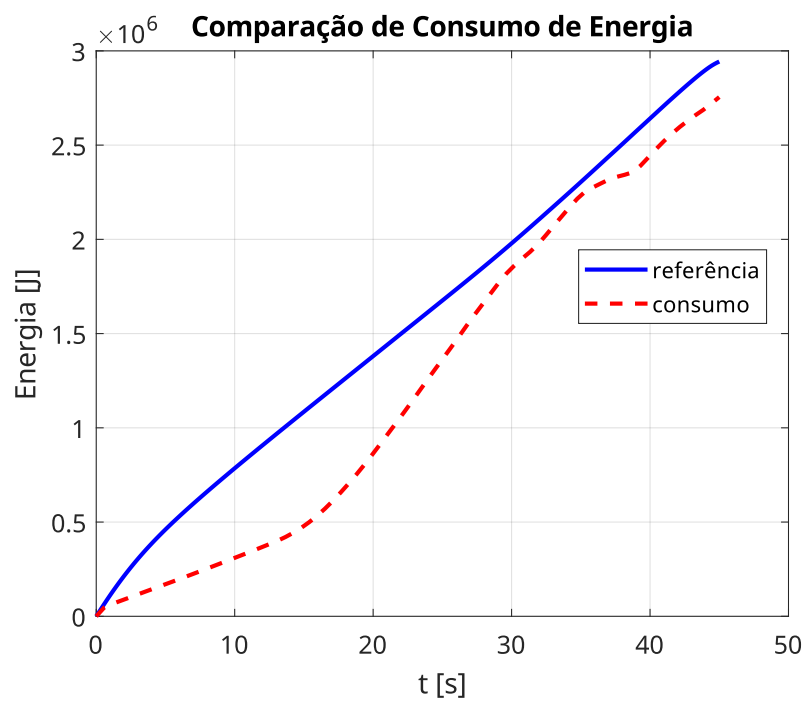


FIGURA 4.11 – Comparação da energia.

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. 19

BECERRA, V. **PSOPT Optimal Control Solver: User Manual**. 2022. Disponível em: <<https://github.com/PSOPT/psopt/releases/tag/5.02>>. 18, 19, 20, 22, 27

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>. 19

BELLMAN, R. **Dynamic Programming**. [S.l.]: Princeton University Press, 2010. Google-Books-ID: 92aYDwAAQBAJ. ISBN 978-0-691-14668-3. 19

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>>. 20

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. 19, 20, 21, 24, 51

BRYSON, A. E. **Applied Optimal Control: Optimization, Estimation and Control**. [S.l.]: Routledge, 2018. Google-Books-ID: LFUPEAAAQBAJ. ISBN 978-1-351-46592-2. 19

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. 27, 40

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>>. 20, 22, 24, 51

KELLY, M. P. **OptimTraj: Trajectory Optimization for Matlab**. 2022. Disponível em: <<https://github.com/MatthewPeterKelly/OptimTraj>>. 18, 19, 20, 27

KIRK, D. E. **Optimal Control Theory: An Introduction**. [S.l.]: Courier Corporation, 2004. Google-Books-ID: fCh2SAtWIdwC. ISBN 978-0-486-43484-1. 19

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>>. 19

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>>. 19

LEWIS, F. L. *et al.* **Optimal Control**. [S.l.]: John Wiley & Sons, 2012. Google-Books-ID: U3Gtlot_hYEC. ISBN 978-1-118-12272-3. 19, 20

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>>. 20

PONTRYAGIN, L. S. **Mathematical Theory of Optimal Processes**. [S.l.]: CRC Press, 1987. Google-Books-ID: kwzq0F4cBVAC. ISBN 978-2-88124-077-5. 19

SIMPLICIO, H. **hsimplicio/tg-ita**. 2024. Disponível em: <<https://github.com/hsimplicio/tg-ita>>. 27, 54

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>>. 19

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

```

Código Fonte 2 – Classe TrajectoryProblem

A.1.2 packZ.m

```

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

```

```

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.2.1 mainTemplate.m

```

1      % Main script for template trajectory optimization problem
2      clear; clc; close all;
3

```

```
4  % Add path to the TrajectoryProblem class
5  addpath('..');
6
7  %% Create problem instance
8  % TODO: Set the number of states and controls for your problem
9  nx = 2; % Number of states
10 nu = 1; % Number of controls
11 problem = TrajectoryProblem(nx, nu);
12
13 %% Set time bounds
14 t0Low = 0;
15 t0Upp = 0;
16 tFLow = 0;
17 tFUpp = 1;
18 problem.setTimeBounds(t0Low, t0Upp, tFLow, tFUpp);
19
20 %% Set time boundary conditions
21 t0 = 0;
22 tF = 1;
23 problem.setTimeBoundaryConditions(t0, tF);
24
25 %% Set state bounds
26 % TODO: Define your state bounds
27 xLow = [-inf; -inf]; % Lower bounds for each state
28 xUpp = [inf; inf]; % Upper bounds for each state
29 problem.setStateBounds(xLow, xUpp);
30
31 %% Set control bounds
32 % TODO: Define your control bounds
33 uLow = [-inf]; % Lower bounds for each control
34 uUpp = [inf]; % Upper bounds for each control
35 problem.setControlBounds(uLow, uUpp);
36
37 %% Set parameters
38 % TODO: Create and set your problem parameters
39 params = templateParams();
40 problem.setParameters(params);
41
42 %% Set boundary conditions
43 % TODO: Define your boundary conditions
```

```
44 x0 = [0; 0]; % Initial state
45 xF = [1; 0]; % Final state
46 problem.setBoundaryConditions(x0, xF);
47
48 %% Set functions
49 % TODO: Set your dynamics function
50 problem.setDynamics(@templateDynamics);
51
52 % TODO: Set your objective function
53 problem.setObjective(@boundaryObjective, @pathObjective);
54
55 % TODO: Set your constraints function
56 problem.setConstraints(@boundaryConstraints, @pathConstraints);
57
58 %% Set solver options
59 % TODO: Adjust grid size and solver options as needed
60 nGrid = [50, 100, 200]; % Number of grid points for each iteration
61 options = optimoptions('fmincon');
62 options.Display = 'iter';
63 options.MaxFunEvals = 1e5;
64 problem.setSolverOptions(options, nGrid);
65
66 %% Optional: Set constraint checking function
67 % TODO: Set your constraint checking function
68 problem.setConstraintsCheck(@checkConstraints);
69
70 %% Optional: Set variable names
71 % TODO: Set meaningful names for your states and controls
72 problem.setVariableNames({'x1', 'x2'}, {'u1'});
73
74 %% Validate problem definition
75 problem.validate();
76
77 %% Solve the problem
78 solution = problem.solveWithTrapezoidalCollocation();
79
80 %% Save the solution
81 mkdir('results');
82 save('results/solution.mat', 'solution');
83
```

```

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

```

Código Fonte 9 – Arquivo principal do template

A.2.2 templateDynamics.m

```

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

```

Código Fonte 10 – Função templateDynamics do template

A.2.3 templateParams.m

```

1  function params = templateParams()
2      % Returns the problem parameters

```

```
3      % TODO: Define your problem parameters here
4
5      % Example parameters
6      params.param1 = 1.0;
7      params.param2 = 2.0;
8  end
```

Código Fonte 11 – Função `templateParams` do `template`

A.2.4 `boundaryConstraints.m`

```
1  function [c, ceq] = boundaryConstraints(x0, xF, t0, tF, boundaryConditions)
2      % Evaluates boundary constraints
3      % Inputs:
4      %   x0: [nx,1] initial state vector
5      %   xF: [nx,1] final state vector
6      %   t0: initial time
7      %   tF: final time
8      %   boundaryConditions: struct with boundary conditions
9      % Outputs:
10     %   c: inequality constraints at boundaries
11     %   ceq: equality constraints at boundaries
12
13     arguments
14         x0 double
15         xF double
16         t0 double
17         tF double
18         boundaryConditions struct
19     end
20
21     % TODO: Implement your boundary constraints
22     c = [];
23     ceq = [];
24 end
```

Código Fonte 12 – Função `boundaryConstraints` do `template`

A.2.5 pathConstraints.m

```

1  function [c, ceq] = pathConstraints(time, state, control, params)
2      % Evaluates path constraints
3      % Inputs:
4      %   time: [1,n] time vector
5      %   state: [nx,n] state matrix
6      %   control: [nu,n] control matrix
7      %   params: parameter struct
8      % Outputs:
9      %   c: inequality constraints
10     %   ceq: equality constraints
11
12     arguments
13         time double
14         state double
15         control double
16         params struct
17     end
18
19     % TODO: Implement your path constraints
20     c = [];
21     ceq = [];
22 end

```

Código Fonte 13 – Função pathConstraints do template

A.2.6 boundaryObjective.m

```

1  function cost = boundaryObjective(x0, xF, t0, tF, params)
2      % Evaluates boundary cost (Mayer term)
3      % Inputs:
4      %   x0: [nx,1] initial state vector
5      %   xF: [nx,1] final state vector
6      %   t0: initial time
7      %   tF: final time
8      %   params: parameter struct
9      % Outputs:
10     %   cost: scalar cost value

```

```

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

```

Código Fonte 14 – Função boundaryObjective do template

A.2.7 pathObjective.m

```

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

```

Código Fonte 15 – Função pathObjective do template

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 16 – Arquivo principal do problema movimento simples

A.3.2 simpleMassDynamics.m

```

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

```

Código Fonte 17 – Função simpleMassDynamics do problema movimento simples

A.3.3 boundaryConstraints.m

```

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

```

```

5         t0 double
6         tF double
7         boundaryConditions struct
8     end
9
10    % Initial and final conditions
11    ceq = [
12        x0 - boundaryConditions.x0    % Initial state
13        xF - boundaryConditions.xF    % Final state
14    ];
15
16    % No inequality constraints
17    c = [];
18 end

```

Código Fonte 18 – Função boundaryConstraints do problema movimento simples

A.3.4 pathConstraints.m

```

1    function [c, ceq] = pathConstraints(time, state, control, params)
2        arguments
3            time double
4            state double
5            control double
6            params struct
7        end
8
9        % No path constraints needed
10       c = [];    % Inequality constraints
11       ceq = []; % Equality constraints
12 end

```

Código Fonte 19 – Função pathConstraints do problema movimento simples

A.3.5 boundaryObjective.m

```

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

```

```

3         x0 double
4         xF double
5         t0 double
6         tF double
7         params struct
8     end
9
10    % No boundary cost
11    cost = 0;
12 end

```

Código Fonte 20 – Função `boundaryObjective` do problema movimento simples

A.3.6 `pathObjective.m`

```

1 function L = pathObjective(time, state, control, params)
2     arguments
3         time double
4         state double
5         control double
6         params struct
7     end
8
9     % Minimize control effort
10    L = control(1, :).^2;
11 end

```

Código Fonte 21 – Função `pathObjective` do problema movimento simples

A.3.7 `generateSimpleMassGuess.m`

```

1 function zGuess = generateSimpleMassGuess(problem)
2     % Extract problem parameters
3     [x0, xF] = problem.getBoundaryConditions();
4     nGrid = problem.getGridSize();
5
6     % Linear interpolation for states
7     state = [

```

```

8         linspace(x0(1), xF(1), nGrid(1))    % Position
9         linspace(x0(2), xF(2), nGrid(1))    % Velocity
10    ];
11
12    % Linear interpolation for control
13    control = linspace(1, -1, nGrid(1));
14
15    % Combine states and control
16    zGuess = [state; control];
17 end

```

Código Fonte 22 – Função generateSimpleMassGuess do problema movimento simples

A.4 Arquivos do problema do pêndulo invertido

A.4.1 mainPendulumSwingup.m

```

1  % Pendulum swing-up trajectory optimization
2  clear; clc; close all;
3
4  % Add path to the TrajectoryProblem class
5  addpath('..');
6
7  %% Create problem instance
8  nx = 2; % States: [theta; omega] (angle, angular velocity)
9  nu = 1; % Control: [tau] (torque)
10 problem = TrajectoryProblem(nx, nu);
11
12 %% Set time bounds
13 problem.setTimeBounds(0, 0, 0, 3); % Fixed time interval [0, 3]
14 problem.setTimeBoundaryConditions(0, 3);
15
16 %% Set state bounds
17 % Angle bounds:  $\pm 4\pi$  (allow multiple rotations)
18 % Angular velocity bounds:  $\pm 8\pi$  rad/s
19 xLow = [-4*pi; -8*pi];
20 xUpp = [4*pi; 8*pi];
21 problem.setStateBounds(xLow, xUpp);

```

```
22
23 %% Set control bounds
24 uLow = [-2]; % Maximum negative torque
25 uUpp = [2]; % Maximum positive torque
26 problem.setControlBounds(uLow, uUpp);
27
28 %% Set parameters
29 params.MASS = 1.0; % Mass [kg]
30 params.LENGTH = 0.5; % Length [m]
31 params.GRAVITY = 9.81; % Gravity [m/s^2]
32 params.DAMPING = 0.1; % Damping coefficient
33 problem.setParameters(params);
34
35 %% Set boundary conditions
36 x0 = [pi; 0]; % Start hanging down (theta = pi) at rest
37 xF = [0; 0]; % End pointing up (theta = 0) at rest
38 problem.setBoundaryConditions(x0, xF);
39
40 %% Set functions
41 problem.setDynamics(@pendulumDynamics);
42 problem.setObjective([], @pathObjective); % Minimize control effort
43 problem.setConstraints([], []); % No additional constraints needed
44
45 %% Set solver options
46 nGrid = [30, 60]; % Start with coarse grid, then refine
47 options = optimoptions('fmincon');
48 options.Display = 'iter';
49 options.MaxFunctionEvaluations = 1e5;
50 problem.setSolverOptions(options, nGrid);
51
52 %% Set variable names
53 problem.setVariableNames({'theta', 'omega'}, {'tau'});
54
55 %% Solve the problem
56 solution = problem.solveWithTrapezoidalCollocation();
57
58 %% Save solution
59 mkdir('results');
60 save('results/solution.mat', 'solution');
```

```
62  %% Plot results
63  plotResults('Pendulum Swing-up', solution(end).z, true);
```

Código Fonte 23 – Arquivo principal do problema do pêndulo invertido

A.4.2 pendulumDynamics.m

```
1  function dx = pendulumDynamics(time, state, control, params)
2      % Extract states and control
3      theta = state(1,:); % Angle
4      omega = state(2,:); % Angular velocity
5      tau = control(1,:); % Input torque
6
7      % Extract parameters
8      m = params.MASS;
9      l = params.LENGTH;
10     g = params.GRAVITY;
11     b = params.DAMPING;
12
13     % Compute derivatives
14     dtheta = omega;
15     domega = (tau - m*g*l*sin(theta) - b*omega) / (m*l^2);
16
17     dx = [dtheta; domega];
18 end
```

Código Fonte 24 – Função pendulumDynamics do problema do pêndulo invertido

A.4.3 pathObjective.m

```
1  function L = pathObjective(time, state, control, params)
2      % Minimize control effort (quadratic cost)
3      L = control(1,:).^2;
4  end
```

Código Fonte 25 – Função pathObjective do problema do pêndulo invertido

A.5 Arquivos do problema da manobra de mudança de faixa

A.5.1 mainLaneChange.m

```

1  % Car lane-change trajectory optimization
2  clear; clc; close all;
3
4  % Add path to the TrajectoryProblem class
5  addpath('..');
6
7  %% Create problem instance
8  nx = 4; % States: [x; y; vx; vy] (position x,y and velocities)
9  nu = 2; % Control: [ax; ay] (acceleration in x,y)
10 problem = TrajectoryProblem(nx, nu);
11
12 %% Set time bounds
13 problem.setTimeBounds(0, 0, 0, 5); % Fixed time interval
14                                     % [0, 5] seconds
15 problem.setTimeBoundaryConditions(0, 5);
16
17 %% Parameters
18 params.MAX_ACCEL = 2.0; % Maximum acceleration [m/s^2]
19 params.TARGET_VX = 15.0; % Target forward velocity [m/s]
20 params.LANE_WIDTH = 3.5; % Standard lane width [m]
21 problem.setParameters(params);
22
23 %% Set state bounds
24 xLow = [0; -1; 10; -5]; % x position, y position,
25                          % x velocity, y velocity
26 xUpp = [100; 5; 20; 5];
27 problem.setStateBounds(xLow, xUpp);
28
29 %% Set control bounds
30 uLow = [-params.MAX_ACCEL; -params.MAX_ACCEL];
31 uUpp = [params.MAX_ACCEL; params.MAX_ACCEL];
32 problem.setControlBounds(uLow, uUpp);
33
34 %% Set boundary conditions
35 x0 = [0; 0; params.TARGET_VX; 0]; % Start in left lane

```



```

36 xF = [75; params.LANE_WIDTH; params.TARGET_VX; 0]; % End in right lane
37 problem.setBoundaryConditions(x0, xF);
38
39 %% Set functions
40 problem.setDynamics(@carDynamics);
41 problem.setObjective([], @pathObjective); % Minimize acceleration
42                                         % and deviation from
43                                         % target speed
44 problem.setConstraints(@boundaryConstraints, []);
45
46 %% Set solver options
47 nGrid = [20, 40, 80]; % Start with coarse grid, then refine
48 options = optimoptions('fmincon');
49 options.Display = 'iter';
50 options.MaxFunEvals = 1e5;
51 problem.setSolverOptions(options, nGrid);
52
53 %% Solve the problem
54 solution = problem.solveWithTrapezoidalCollocation();
55
56 %% Save solution
57 mkdir('results');
58 save('results/laneChangeSolution.mat', 'solution');
59
60 %% Plot results
61 plotResults('Lane Change Maneuver', solution(end).z, true);

```

Código Fonte 26 – Arquivo principal do problema da manobra de mudança de faixa

A.5.2 carDynamics.m

```

1 function dx = carDynamics(time, state, control, params)
2     % Simple double integrator dynamics
3     % States: [x; y; vx; vy]
4     % Controls: [ax; ay]
5
6     % Extract velocities and accelerations
7     vx = state(3,:);
8     vy = state(4,:);

```

```

9      ax = control(1,:);
10     ay = control(2,:);
11
12     % Simple kinematics
13     dx = [
14         vx;          % dx/dt = vx
15         vy;          % dy/dt = vy
16         ax;          % dvx/dt = ax
17         ay;          % dvy/dt = ay
18     ];
19     end

```

Código Fonte 27 – Função carDynamics do problema da manobra de mudança de faixa

A.5.3 boundaryConstraints.m

```

1  function [c, ceq] = boundaryConstraints(x0, xF, t0, tF, boundaryConditions)
2      arguments
3          x0 double
4          xF double
5          t0 double
6          tF double
7          boundaryConditions struct
8      end
9
10     % Initial and final conditions
11     ceq = [
12         x0 - boundaryConditions.x0; % Initial state
13         xF - boundaryConditions.xF; % Final state
14     ];
15
16     % No inequality constraints
17     c = [];
18     end

```

Código Fonte 28 – Função boundaryConstraints do problema da manobra de mudança de faixa

A.5.4 pathObjective.m

```
1  function L = pathObjective(time, state, control, params)
2      % Minimize acceleration and deviation from target velocity
3      ax = control(1,:);
4      ay = control(2,:);
5      vx = state(3,:);
6
7      % Weights for different objectives
8      w_accel = 1.0;    % Weight for acceleration minimization
9      w_speed = 0.1;    % Weight for speed tracking
10
11     % Compute costs
12     accel_cost = ax.^2 + ay.^2; % Minimize acceleration (comfort)
13     speed_cost = (vx - params.TARGET_VX).^2; % Stay close to
14                                           % target speed
15
16     L = w_accel * accel_cost + w_speed * speed_cost;
17 end
```

Código Fonte 29 – Função pathObjective do problema da manobra de mudança de faixa

A.6 Arquivos do problema da braquistócrona

A.6.1 mainBrachistochrone.m

```
1  % Brachistochrone problem:
2  % Find the path of fastest descent between two points
3  clear;
4  clc;
5  close all;
6
7  % Add path to the TrajectoryProblem class
8  addpath('..');
9
10 %% Create problem instance
11 nx = 3; % States: [x; y; v] (horizontal position,
12          %                      vertical position,
```

```

13         %                                velocity)
14     nu = 1; % Control: [theta] (angle of the path)
15     problem = TrajectoryProblem(nx, nu);
16
17     %% Set time bounds
18     t0Low = 0;
19     t0Upp = 0;
20     tFlow = 0;
21     tFUpp = 2 + 1e-6;
22     problem.setTimeBounds(t0Low, t0Upp, tFlow, tFUpp);
23
24     %% Set time boundary conditions
25     t0 = 0;
26     tF = 1; % Initial guess for final time (will be optimized)
27     problem.setTimeBoundaryConditions(t0, tF);
28
29     %% Set state bounds
30     xLow = [0; -5; 0]; % Lower bounds: x>=0, y>=-5, v>=0
31     xUpp = [5; 0; 15]; % Upper bounds: x<=5, y<=0, v<=15
32     problem.setStateBounds(xLow, xUpp);
33
34     %% Set control bounds
35     uLow = [-pi / 2]; % Minimum angle
36     uUpp = [pi / 2]; % Maximum angle
37     problem.setControlBounds(uLow, uUpp);
38
39     %% Set parameters
40     params = brachistochroneParams();
41     problem.setParameters(params);
42
43     %% Set boundary conditions
44     x0 = [0; 0; 0]; % Start at origin with zero velocity
45     xF = [5; -5; xUpp(3)]; % End at (5,-5)
46     % with free final velocity up to vMax
47     problem.setBoundaryConditions(x0, xF);
48
49     %% Set scaling
50     % xScale = abs(xF);
51     % uScale = 1; % Angle scale
52     % tScale = sqrt(2*abs(xF(2))/params.GRAVITY); % Natural time scale

```

```

53
54 % problem.setScaling('state', xScale); % Scale states to be O(1)
55 % problem.setScaling('control', uScale); % Scale control to be O(1)
56 % problem.setScaling('time', tScale); % Scale time to be O(1)
57
58 %% Set functions
59 problem.setDynamics(@brachistochroneDynamics);
60 problem.setObjective(@boundaryObjective, @pathObjective);
61 problem.setConstraints(@boundaryConstraints, @pathConstraints);
62
63 %% Set solver options
64 nGrid = [10, 20, 40]; % Number of grid points for each iteration
65 options = optimoptions('fmincon');
66 options.Display = 'iter';
67 options.MaxFunEvals = 1e5;
68 options.Algorithm = 'sqp';
69 options.EnableFeasibilityMode = true;
70 options.SubproblemAlgorithm = 'cg';
71 options.FiniteDifferenceType = 'central'; % More accurate gradients
72 options.FiniteDifferenceStepSize = 1e-6; % Smaller step size
73 options.OptimalityTolerance = 1e-6; % Tighter tolerance
74 options.ConstraintTolerance = 1e-6; % Tighter tolerance
75 options.StepTolerance = 1e-10; % Smaller steps
76 options.MaxIterations = 1000; % Increase if needed
77 options.ScaleProblem = true; % Add scaling
78 options.HessianApproximation = 'bfgs'; % Use BFGS approximation
79
80 problem.setSolverOptions(options, nGrid);
81
82 %% Optional: Set constraint checking function
83 problem.setConstraintsCheck(@checkConstraints);
84
85 %% Set variable names
86 problem.setVariableNames({'x', 'y', 'v'}, {'theta'});
87
88 %% Solve the problem
89 zGuess = generateBrachistochroneGuess(problem);
90 solution = problem.solveWithTrapezoidalCollocation(zGuess);
91
92 %% Display solution

```

```

93     disp('=== Solution Analysis ===');
94     disp('Time span:');
95     disp(solution(end).z.timeSpan);
96
97     disp('Initial state:');
98     disp(solution(end).z.state(:, 1)');
99     disp('Target initial state:');
100    disp(x0');
101
102    disp('Final state:');
103    disp(solution(end).z.state(:, end)');
104    disp('Target final state:');
105    disp(xF');
106
107    % Check energy conservation
108    E = 0.5 * solution(end).z.state(3, :).^2 + ...
109    params.GRAVITY * solution(end).z.state(2, :);
110    disp('Energy variation:');
111    disp(['Max: ', num2str(max(E) - min(E))]);
112
113    % Save the solution
114    mkdir('results');
115    save('results/solution.mat', 'solution');
116
117    %% Plot results
118    plotResults('Brachistochrone Solution', solution(end).z, true);
119
120    %% Compare with analytical solution
121    compareWithAnalytical(solution(end).z, true);

```

Código Fonte 30 – Arquivo principal do problema da braquistócrona

A.6.2 brachistochroneDynamics.m

```

1     function dx = brachistochroneDynamics(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(3, :);          % Velocity
11    theta = control(1, :);    % Path angle
12    g = params.GRAVITY;      % Gravity
13
14    % System dynamics
15    dx = [
16        v .* cos(theta)      % dx/dt = v*cos(theta)
17        v .* sin(theta)      % dy/dt = v*sin(theta)
18        -g .* sin(theta)     % dv/dt = -g*sin(theta)
19    ];
20 end

```

Código Fonte 31 – Função brachistochroneDynamics do problema da braquistócrona

A.6.3 brachistochroneParams.m

```

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

```

Código Fonte 32 – Função brachistochroneParams do problema da braquistócrona

A.6.4 boundaryConstraints.m

```

1 function [c, ceq] = boundaryConstraints(x0, xF, t0, tF, boundaryConditions)
2     arguments
3         x0 double
4         xF double
5         t0 double
6         tF double
7         boundaryConditions struct
8     end
9

```

```

10      % Initial conditions
11      ceq = [
12          x0(1) - boundaryConditions.x0(1)    % Initial x position
13          x0(2) - boundaryConditions.x0(2)    % Initial y position
14          x0(3) - boundaryConditions.x0(3)    % Initial velocity
15          xF(1) - boundaryConditions.xF(1)    % Final x position
16          xF(2) - boundaryConditions.xF(2)    % Final y position
17      ];
18
19      % Debug output
20      % disp('Current boundary constraint values:');
21      % disp(['Initial x: ', num2str(ceq(1))]);
22      % disp(['Initial y: ', num2str(ceq(2))]);
23      % disp(['Initial v: ', num2str(ceq(3))]);
24      % disp(['Final x: ', num2str(ceq(4))]);
25      % disp(['Final y: ', num2str(ceq(5))]);
26
27      % No inequality constraints
28      c = [];
29  end

```

Código Fonte 33 – Função boundaryConstraints do problema da braquistócrona

A.6.5 pathConstraints.m

```

1  function [c, ceq] = pathConstraints(time, state, control, params)
2      arguments
3          time double
4          state double
5          control double
6          params struct
7      end
8
9      g = params.GRAVITY;
10
11     % Extract states
12     sx = state(1, :);    % Horizontal position
13     sy = state(2, :);    % Vertical position
14     v = state(3, :);    % Velocity

```



```

15
16     % Path constraints
17     c = [];
18     % Energy conservation (optional)
19     E = 0.5 * v.^2 + g * sy;
20     E0 = g * sy(1); % Initial potential energy
21     ceq = (E - E0)'; % Energy should be conserved
22
23     % Debug output
24     % disp('Energy conservation error:');
25     % disp(['Max error: ', num2str(max(abs(ceq)))]);
26     % disp(['Mean error: ', num2str(mean(abs(ceq)))]);
27 end

```

Código Fonte 34 – Função pathConstraints do problema da braquistócrona

A.6.6 boundaryObjective.m

```

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

```

Código Fonte 35 – Função boundaryObjective do problema da braquistócrona

A.6.7 pathObjective.m

```

1  function L = pathObjective(time, state, control, params)
2      arguments
3          time double

```

```

4         state double
5         control double
6         params struct
7     end
8
9     % No running cost
10    L = zeros(size(time));
11 end

```

Código Fonte 36 – Função pathObjective do problema da braquistócrona

A.6.8 generateBrachistochroneGuesses.m

```

1  function zGuess = generateBrachistochroneGuess(problem, plotFlag)
2      % Generate an initial guess for the brachistochrone problem
3      %
4      % The guess is a parabolic path that connects the initial and final
5      % points. The velocity is estimated using energy conservation.
6      %
7      % Inputs:
8      %   problem - A TrajectoryProblem object
9      %   plotFlag - A boolean flag to plot the guess
10     %
11     % Outputs:
12     %   zGuess - A guess for the solution
13
14     if nargin < 2
15         plotFlag = false;
16     end
17
18     % Extract problem parameters
19     params = problem.getParameters();
20     g = params.GRAVITY;
21
22     % Extract time bounds, boundary conditions, and grid size
23     [t0, tF] = problem.getTimeBounds();
24     [x0, xF] = problem.getBoundaryConditions();
25     nGrid = problem.getGridSize();
26

```

```
27     time = linspace(t0, tF, nGrid(1));
28
29     % Initial and final positions
30     sx0 = x0(1);
31     sy0 = x0(2);
32     sxf = xF(1);
33     syf = xF(2);
34
35     % Generate parabolic path as initial guess
36     sx = linspace(sx0, sxf, nGrid(1));
37     %
38     %  $x - sx0 = a * (y - sy0)^2$ 
39     %
40     a = (sxf - sx0) / (syf - sy0)^2;
41     sy = sy0 + sign(syf - sy0) * sqrt((sx - sx0) / a);
42
43     % Estimate velocity (using energy conservation)
44     v = sqrt(2 * g * (sy0 - sy));
45
46     % Estimate control angle
47     dsx = gradient(sx, time);
48     dsy = gradient(sy, time);
49     theta = atan2(dsy, dsx);
50
51     state = [sx; sy; v];
52     control = theta;
53     zGuess = [state; control];
54
55     % Plot if requested
56     if plotFlag
57         guess = struct();
58         guess.time = time;
59         guess.state = state;
60         guess.control = control;
61         plotResults('Initial Guess', guess);
62         compareWithAnalytical(guess);
63     end
64 end
```

Código Fonte 37 – Função `generateBrachistochroneGuess` do problema da braquistócrona

A.6.9 `checkConstraints.m`

```

1  function violations = checkConstraints(time, state, control, params)
2      violations = struct();
3      violations.state = struct();
4      violations.state.x = struct();
5      violations.state.y = struct();
6      violations.state.v = struct();
7      violations.control = struct();
8      violations.control.theta = struct();
9      violations.energy = struct();
10     violations.velocity = struct();
11
12     % Check state bounds
13     if any(state(1, :) < 0) || any(state(1, :) > 10)
14         violations.state.x = struct( ...
15             'max', max(state(1, :)), ...
16             'min', min(state(1, :)), ...
17             'bounds', [0, 10]);
18     end
19
20     if any(state(2, :) < -10) || any(state(2, :) > 0)
21         violations.state.y = struct( ...
22             'max', max(state(2, :)), ...
23             'min', min(state(2, :)), ...
24             'bounds', [-10, 0]);
25     end
26
27     if any(state(3, :) < 0) || any(state(3, :) > 20)
28         violations.state.v = struct( ...
29             'max', max(state(3, :)), ...
30             'min', min(state(3, :)), ...
31             'bounds', [0, 20]);
32     end
33
34     % Check control bounds

```

```

35     if any(control < -pi / 2) || any(control > pi / 2)
36         violations.control.theta = struct( ...
37             'max', max(control), ...
38             'min', min(control), ...
39             'bounds', [-pi / 2, pi / 2]);
40     end
41
42     % Check energy conservation
43     E = 0.5 * state(3, :).^2 + params.GRAVITY * state(2, :);
44     dE = gradient(E, time);
45
46     if any(abs(dE) > 1e-6)
47         violations.energy = struct( ...
48             'maxChange', max(abs(dE)), ...
49             'meanChange', mean(abs(dE)));
50     end
51
52     % Check velocity consistency
53     vFromEnergy = sqrt(2 * params.GRAVITY * (state(2, 1) - state(2, :)));
54     velocityError = abs(state(3, :) - vFromEnergy);
55
56     if any(velocityError > 1e-6)
57         violations.velocity = struct( ...
58             'maxError', max(velocityError), ...
59             'meanError', mean(velocityError));
60     end
61
62     % Debug: print violations if any
63     % if ~isempty(fieldnames(violations))
64     %     disp('Constraint Violations Found:');
65     %     if ~isempty(fieldnames(violations.energy))
66     %         disp('Energy:');
67     %         disp(violations.energy);
68     %     end
69     %     if ~isempty(fieldnames(violations.velocity))
70     %         disp('Velocity:');
71     %         disp(violations.velocity);
72     %     end
73     %     if ~isempty(fieldnames(violations.state))
74     %         disp('State:');

```

```

75      %      if ~isempty(fieldnames(violations.state.x))
76      %          disp('sx:');
77      %          disp(violations.state.x);
78      %      end
79      %      if ~isempty(fieldnames(violations.state.y))
80      %          disp('sy:');
81      %          disp(violations.state.y);
82      %      end
83      %      if ~isempty(fieldnames(violations.state.v))
84      %          disp('v:');
85      %          disp(violations.state.v);
86      %      end
87      %  end
88      %  if ~isempty(fieldnames(violations.control))
89      %      disp('Control:');
90      %      if ~isempty(fieldnames(violations.control.theta))
91      %          disp('theta:');
92      %          disp(violations.control.theta);
93      %      end
94      %  end
95      % end
96  end

```

Código Fonte 38 – Função checkConstraints do problema da braquistócrona

A.6.10 compareWithAnalytical.m

```

1  function compareWithAnalytical(z, plotFlag)
2      % Compare numerical solution with analytical cycloid solution
3
4      if nargin < 2
5          plotFlag = false;
6      end
7
8      % Extract final conditions
9      xf = z.state(1, end);
10     yf = z.state(2, end);
11
12     % Calculate cycloid parameters

```

```

13
14     % Calculate final angle
15     f = @(theta_f) xf / yf - ...
16     (theta_f - sin(theta_f)) / (1 - cos(theta_f));
17     theta_f = fsolve(f, pi / 2, ...
18                     optimoptions('fsolve', 'Display', 'off'));
19
20     % Calculate radius
21     R = yf / (1 - cos(theta_f));
22
23     % Generate analytical solution points
24     theta = linspace(0, theta_f, 200);
25     x_analytical = R * (theta - sin(theta));
26     y_analytical = R * (1 - cos(theta));
27
28     % Plot comparison
29     figure('Name', 'Comparison with Analytical Solution');
30     plot(z.state(1, :), z.state(2, :), 'b-', ...
31          'LineWidth', 2, 'DisplayName', 'Numérica');
32     hold on;
33     plot(x_analytical, y_analytical, 'r--', ...
34          'LineWidth', 2, 'DisplayName', 'Analítica');
35     plot([0 xf], [0 yf], 'k.', ...
36          'MarkerSize', 20, 'DisplayName', 'Pontos Iniciais e Finais');
37     grid on;
38     xlabel('x [m]');
39     ylabel('y [m]');
40     title('Comparação com Solução Analítica (Ciclóide)');
41     legend('Location', 'best');
42     axis equal;
43
44     if plotFlag
45         mkdir('figures');
46         savefig('figures/Brachistochrone Comparison.fig');
47         saveas(gcf, 'figures/Brachistochrone Comparison.svg');
48         saveas(gcf, 'figures/Brachistochrone Comparison.png');
49     end
50 end

```

A.7 Arquivos do problema do eVTOL

A.7.1 mainEvtol.m

```
1  %% Main script for EVTOL trajectory optimization
2  clear;
3  clc;
4  close all;
5
6  % Add path to the TrajectoryProblem class
7  addpath('..');
8
9  %% Create problem instance
10 problem = TrajectoryProblem(5, 2); % 5 states, 2 controls
11
12 %% Set time bounds
13 t0Low = 0;
14 t0Upp = 0;
15 tFLow = 0;
16 tFUpp = 45;
17 problem.setTimeBounds(t0Low, t0Upp, tFLow, tFUpp);
18
19 %% Set time boundary conditions
20 t0 = 0;
21 tF = 45;
22 problem.setTimeBoundaryConditions(t0, tF);
23
24 %% Set state bounds
25 xLow = [0; -10; 0; -5; 0];
26 xUpp = [1000; 110; 35; 6; 1e8];
27 problem.setStateBounds(xLow, xUpp);
28
29 %% Set control bounds
30 uLow = [0; 0];
31 uUpp = [1800; 2600];
32 problem.setControlBounds(uLow, uUpp);
33
34 %% Set scaling
35 % problem.setScaling('time', tF - t0);
```



```

36 % problem.setScaling('state', xUpp - xLow);
37 % problem.setScaling('control', uUpp - uLow);
38
39 %% Set parameters
40 problem.setParameters(evtolParams());
41
42 %% Set boundary conditions
43 x0 = [0; 0; 0; 0; 0];
44 xF = [1000; 100; 25; 0; 1e8]; % Free final energy
45 problem.setBoundaryConditions(x0, xF);
46
47 %% Set functions
48 % Set dynamics
49 problem.setDynamics(@evtolDynamics);
50
51 % Set objective
52 hBoundaryObjective = @boundaryObjective; % Mayer Term
53 hPathObjective = []; % integrand of Lagrange Term
54 problem.setObjective(hBoundaryObjective, hPathObjective);
55
56 % Set constraints
57 hBoundaryConstraint = @boundaryConstraints; % Boundary Constraints
58 hPathConstraint = []; % Path Constraints
59 problem.setConstraints(hBoundaryConstraint, hPathConstraint);
60
61 %% Set solver options
62 nGrid = [10, 20, 40, 80];
63 options = optimoptions('fmincon');
64 options.Display = 'iter';
65 options.MaxFunEvals = 1e5;
66 options.Algorithm = 'sqp';
67 options.EnableFeasibilityMode = true;
68 options.SubproblemAlgorithm = 'cg';
69 options.FiniteDifferenceType = 'central'; % More accurate gradients
70 options.FiniteDifferenceStepSize = 1e-6; % Smaller step size
71 options.OptimalityTolerance = 1e-6; % Tighter tolerance
72 options.ConstraintTolerance = 1e-6; % Tighter tolerance
73 options.StepTolerance = 1e-10; % Smaller steps
74 options.MaxIterations = 1000; % Increase if needed
75 options.ScaleProblem = true; % Add scaling

```

```
76 options.HessianApproximation = 'bfgs';      % Use BFGS approximation
77
78 problem.setSolverOptions(options, nGrid);
79
80 % Optional: Set constraint checking function
81 problem.setConstraintsCheck(@checkConstraints);
82
83 % Optional: Set variable names
84 problem.setVariableNames({'sx', 'sy', 'vx', 'vy', 'E'}, {'Tx', 'Ty'});
85
86 % Validate problem definition
87 problem.validate();
88
89 % Optional: Get initial guess
90 zGuess = physicalInitialGuess(problem);
91
92 % Solve the problem
93 solution = problem.solveWithTrapezoidalCollocation(zGuess);
94
95 % Check constraints
96 violations = solution(end).violations;
97 % Display constraint violations
98 if ~isempty(fieldnames(violations.state))
99     disp('State Violations:');
100     disp(violations.state);
101 end
102
103 if ~isempty(fieldnames(violations.control))
104     disp('Control Violations:');
105     disp(violations.control);
106 end
107
108 if ~isempty(fieldnames(violations.physical))
109     disp('Physical Consistency:');
110     disp(violations.physical.velocity);
111     disp(violations.physical.flightPathAngle);
112 end
113
114 disp(solution(end).output);
115
```

```

116 % Save the solution
117 mkdir('results');
118 save(['results/solution-base-t', num2str(tF), '.mat'], 'solution');
119
120 % Plot results
121 plotResults('Final Solution', solution(end).z, true);

```

Código Fonte 40 – Arquivo principal do problema do eVTOL

A.7.2 evtolDynamics.m

```

1  function stateDerivatives = evtolDynamics(time, state, control, params)
2      % Define the dynamics for the EVTOL problem
3      %
4      % Inputs:
5      %   time: [1,n] time vector
6      %   state: [5,n] = [posX; posY; velX; velY; energy] = state matrix
7      %   control: [2,n] = [thrustX; thrustY] = control matrix
8      %   params: parameter struct
9      %
10     % Outputs:
11     %   stateDerivatives: [5,n] = [velX; velY; accelX; accelY; powerRate] = deriv
12     %
13     arguments
14         time double
15         state double
16         control double
17         params struct
18     end
19
20     % Get the parameters
21     GRAVITY = params.environment.GRAVITY;
22     AIR_DENSITY = params.environment.AIR_DENSITY;
23     mass = params.aircraft.mass;
24     rotorArea = params.aircraft.rotorArea;
25     chi = params.aircraft.chi;
26
27     % Assign state variables
28     posX = state(1, :);

```

```

29     posY = state(2, :);
30     velX = state(3, :);
31     velY = state(4, :);
32     energy = state(5, :);
33
34     % Assign control variables
35     thrustX = control(1, :);
36     thrustY = control(2, :);
37
38     % Compute aerodynamic parameters
39     V = sqrt(velX.^2 + velY.^2);
40     gamma = computeFlightAngle(velX, velY);
41     angleToVertical = pi / 2 - gamma; % Angle between
42                                     % velocity and vertical
43     angleToHorizontal = gamma; % Angle between
44                               % velocity and horizontal
45
46     % Compute the aerodynamic forces
47     [lift, drag] = computeLiftDrag(velX, velY, params);
48
49     % Compute the thrust forces
50     rotorThrustX = thrustX / 2;
51     rotorThrustY = thrustY / 4;
52
53     % Induced velocity calculations
54     hoverVelocityX = sqrt(rotorThrustX / ...
55                           (2 * AIR_DENSITY * rotorArea));
56     hoverVelocityY = sqrt(rotorThrustY / ...
57                           (2 * AIR_DENSITY * rotorArea));
58
59     inducedVelocityX = computeInducedVelocity(hoverVelocityX, ...
60                                               V, angleToHorizontal);
61     inducedVelocityY = computeInducedVelocity(hoverVelocityY, ...
62                                               V, angleToVertical);
63
64     % Power calculations
65     rotorPowerX = rotorThrustX .* inducedVelocityX;
66     rotorPowerY = rotorThrustY .* inducedVelocityY;
67     armPowerX = 2.0 * rotorPowerX * (1 + chi);
68     armPowerY = 4.0 * rotorPowerY * (1 + chi);

```

```

69     inducedPower = armPowerX + armPowerY;
70
71     forwardPower = thrustX .* V .* sin(angleToHorizontal) + ...
72                   thrustY .* V .* sin(angleToVertical);
73
74     % Compute the derivatives
75     stateDerivatives = [
76         velX % posX derivative
77         velY % posY derivative
78         (thrustX - drag .* cos(gamma) - ...
79          lift .* sin(gamma)) / mass % velX derivative
80         (thrustY - drag .* sin(gamma) + ...
81          lift .* cos(gamma) - mass * GRAVITY) ...
82         / mass % velY derivative
83         inducedPower + forwardPower % energy derivative
84     ];
85 end

```

Código Fonte 41 – Função `evtolDynamics` do problema do eVTOL

A.7.3 `computeLiftDrag.m`

```

1  function [lift, drag] = computeLiftDrag(velX, velY, params)
2      % Computes aerodynamic forces for the EVTOL
3      %
4      % Inputs:
5      %   velX: [1,n] = horizontal velocity component
6      %   velY: [1,n] = vertical velocity component
7      %   params: parameter struct
8      %
9      % Outputs:
10     %   lift: [1,n] = lift force
11     %   drag: [1,n] = drag force
12
13     arguments
14         velX double
15         velY double
16         params struct
17 end

```

```

18
19 % Extract parameters
20 rho = params.environment.AIR_DENSITY;
21
22 CDFus = params.aircraft.fuselage.CD;
23 SfFus = params.aircraft.fuselage.Sf;
24 StFus = params.aircraft.fuselage.St;
25
26 SWing = params.aircraft.wing.S;
27 eWing = params.aircraft.wing.e;
28 CDpWing = params.aircraft.wing.CDp;
29 ARWing = params.aircraft.wing.AR;
30 CLOWing = params.aircraft.wing.CLO;
31 CLalphaWing = params.aircraft.wing.CLalpha;
32 alphaFus = params.aircraft.wing.alphaFus;
33
34 % Variables related to the wing lift coefficient
35 M = params.aircraft.wing.sigmoid.M;
36 alpha0 = params.aircraft.wing.sigmoid.alpha0;
37
38 gamma = computeFlightAngle(velX, velY);
39 alpha = -gamma;
40 alphaWing = alpha + alphaFus; % Wing angle of attack
41
42 % Compute Wing Lift Coefficient
43 straightCL = CLOWing + CLalphaWing * alphaWing;
44
45 % Apply sigmoid function to merge linear and flat plate models
46 sigMinus = exp(-M * (alphaWing - alpha0));
47 sigPlus = exp(M * (alphaWing + alpha0));
48 sigma = (1 + sigMinus + sigPlus) ./ ...
49         ((1 + sigMinus) .* (1 + sigPlus));
50
51 CD = CDpWing + straightCL.^2 / (pi * eWing * ARWing);
52 CL = sigma .* (2 * sign(alphaWing) .* ...
53             sin(alphaWing).^2 .* cos(alphaWing)) + ...
54         (1 - sigma) .* straightCL;
55
56 V = sqrt(velX.^2 + velY.^2);
57

```

```

58     LWing = 0.5 * rho * V.^2 * SWing .* CL;
59     DWing = 0.5 * rho * V.^2 * SWing .* CD;
60
61     % Fuselage drag
62     DfFus = 0.5 * rho * velX.^2 * CDFus * SfFus;
63     DtFus = 0.5 * rho * velY.^2 * CDFus * StFus;
64
65     % Final computation
66     lift = LWing;
67     drag = sqrt(DfFus.^2 + DtFus.^2) + DWing;
68 end

```

Código Fonte 42 – Função computeLiftDrag do problema do eVTOL

A.7.4 computeInducedVelocity.m

```

1  function vi = computeInducedVelocity(vh, V, alpha)
2      % Computes induced velocity using momentum theory
3      %
4      % Inputs:
5      %   vh: [1,n] hover induced velocity
6      %   V: [1,n] free stream velocity
7      %   alpha: [1,n] angle between free stream and rotor disk normal
8      %
9      % Outputs:
10     %   vi: [1,n] induced velocity
11
12     arguments
13         vh double
14         V double
15         alpha double
16     end
17
18     % Input validation
19     assert(all(isfinite(vh)), 'Non-finite hover velocity');
20     assert(all(isfinite(V)), 'Non-finite velocity');
21     assert(all(isfinite(alpha)), 'Non-finite angle');
22
23     % For hover (V=0), vi should equal vh

```

```

24     hover_condition = (V < 1e-6);
25
26     vi = zeros(1, length(vh));
27     vi(hover_condition) = vh(hover_condition);
28
29     vi_func = @(w, vh, V, alpha) w - vh.^2 ./ ...
30         (sqrt((V .* cos(alpha)).^2) + ...
31         (V .* sin(alpha) + w).^2);
32
33     % For non-hover cases, use fzero
34     non_hover = ~hover_condition;
35     options = optimset('TolX', 1e-6);
36
37     for i = find(non_hover)
38         vi(i) = fzero(@(w) vi_func(w, vh(i), V(i), alpha(i)), ...
39             [-50, 50], options);
40     end
41 end

```

Código Fonte 43 – Função computeInducedVelocity do problema do eVTOL

A.7.5 computeFlightAngle.m

```

1  function gamma = computeFlightAngle(velX, velY)
2      % Computes flight path angle
3      %
4      % Inputs:
5      %   velX: [1,n] = horizontal velocity component
6      %   velY: [1,n] = vertical velocity component
7      %
8      % Outputs:
9      %   gamma: [1,n] = flight path angle [rad]
10
11     arguments
12         velX double
13         velY double
14     end
15
16     gamma = pi / 2 * ones(size(velX));

```



```

17
18     conditionX = abs(velX) > 1e-4;
19     conditionY = abs(velY) > 1e-4;
20
21     gamma(conditionX) = atan2(velY(conditionX), velX(conditionX));
22
23     gamma(~conditionX & conditionY) = sign(velY(~conditionX & ...
24                                         conditionY)) * ...
25                                         pi / 2;
26 end

```

Código Fonte 44 – Função computeFlightAngle do problema do eVTOL

A.7.6 evtolParams.m

```

1  function params = evtolParams()
2      % Returns the EVTOL problem parameters
3
4      % Environment parameters
5      params.environment.GRAVITY = 9.78; % Gravitational acceleration [m/s^2]
6      params.environment.AIR_DENSITY = 1.2; % Air density [kg/m^3]
7
8      % Aircraft parameters
9      params.aircraft.mass = 240; % Vehicle mass [kg]
10     params.aircraft.rotorArea = 1.6; % Rotor disk area [m^2]
11     params.aircraft.chi = 1.0; % Rotor-to-rotor
12                                % interference factor [-]
13
14     params.aircraft.fuselage.CD = 1.0; % Fuselage drag coefficient [-]
15     params.aircraft.fuselage.Sf = 2.11; % Fuselage cross-sectional
16                                         % area [m^2]
17     params.aircraft.fuselage.St = 1.47; % Fuselage top area [m^2]
18
19     params.aircraft.wing.S = 4.0; % Wing area [m^2]
20     params.aircraft.wing.e = 0.9; % Wing span efficiency factor [-]
21     params.aircraft.wing.CDp = 0.0437; % Wing parasite
22                                         % drag coefficient [-]
23     params.aircraft.wing.AR = 20.0; % Wing aspect ratio [-]
24     params.aircraft.wing.CLO = 0.28; % Wing lift coefficient at zero

```

```

25                                     % AoA [-]
26     params.aircraft.wing.CLalpha = 4.00; % Wing lift curve
27                                     % slope [rad^-1]
28     params.aircraft.wing.alphaFus = 0 * (pi / 180); % Wing mounting
29                                     % angle [rad]
30     params.aircraft.wing.sigmoid.M = 4.0; % Sigmoid steepness
31                                     % parameter [-]
32     params.aircraft.wing.sigmoid.alpha0 = 20 * (pi / 180); % Stall
33                                     % angle [rad]
34
35     end

```

Código Fonte 45 – Função evtolParams do problema do eVTOL

A.7.7 boundaryConstraints.m

```

1     function [c, ceq] = boundaryConstraints(x0, xF, t0, tF, boundaryConditions)
2         % Evaluates boundary constraints for the EVTOL problem
3         %
4         % Inputs:
5         %   x0: [5,1] = initial state vector [sx; sy; vx; vy; E]
6         %   xF: [5,1] = final state vector [sx; sy; vx; vy; E]
7         %   t0: double = initial time
8         %   tF: double = final time
9         %
10        % Outputs:
11        %   c: inequality constraints at boundaries
12        %   ceq: equality constraints at boundaries
13
14        arguments
15            x0 double
16            xF double
17            t0 {mustBeNumeric}
18            tF {mustBeNumeric}
19            boundaryConditions struct
20        end
21
22        % Initial conditions
23        initialPosX = x0(1);

```

```

24     initialPosY = x0(2);
25     initialVelX = x0(3);
26     initialVelY = x0(4);
27     initialEnergy = x0(5);
28
29     % Final conditions
30     finalPosX = xF(1);
31     finalPosY = xF(2);
32     finalVelX = xF(3);
33     finalVelY = xF(4);
34
35     % Equality constraints
36     ceq = [
37         % Initial conditions
38         % x(0) = boundaryConditions.xLow(1)
39         initialPosX - boundaryConditions.xLow(1)
40         % y(0) = boundaryConditions.xLow(2)
41         initialPosY - boundaryConditions.xLow(2)
42         % vx(0) = boundaryConditions.xLow(3)
43         initialVelX - boundaryConditions.xLow(3)
44         % vy(0) = boundaryConditions.xLow(4)
45         initialVelY - boundaryConditions.xLow(4)
46         % E(0) = boundaryConditions.xLow(5)
47         initialEnergy - boundaryConditions.xLow(5)
48
49         % Final conditions
50         % x(T) = boundaryConditions.xF(1)
51         finalPosX - boundaryConditions.xF(1)
52         % y(T) = boundaryConditions.xF(2)
53         finalPosY - boundaryConditions.xF(2)
54         % vx(T) = boundaryConditions.xF(3)
55         finalVelX - boundaryConditions.xF(3)
56         % vy(T) = boundaryConditions.xF(4)
57         finalVelY - boundaryConditions.xF(4)
58
59         % Time bounds
60         t0 - boundaryConditions.t0
61         tF - boundaryConditions.tF
62     ];
63

```

```

64     % No inequality constraints
65     c = [];
66 end

```

Código Fonte 46 – Função `boundaryConstraints` do problema do eVTOL

A.7.8 `pathConstraints.m`

```

1  function [c, ceq] = pathConstraints(time, state, control, params)
2      % Evaluates path constraints for the EVTOL problem
3      % Only include constraints that cannot be expressed
4      % as simple bounds
5      %
6      % Example of potential nonlinear constraints:
7      % c = [
8      %     sqrt(x(3,:).^2 + x(4,:).^2) - v_max; % velocity magnitude
9      %                                           % constraint
10     sqrt(u(1,:).^2 + u(2,:).^2) - T_max; % total thrust
11     %                                           % constraint
12     % ];
13     %
14     % Inputs:
15     %   time: [1,n] = time vector
16     %   state: [5,n] = state vector [sx; sy; vx; vy; E]
17     %   control: [2,n] = control vector [Tx; Ty]
18     %   params: parameter struct
19     %
20     % Outputs:
21     %   c: [m,1] = inequality constraints
22     %   ceq: [m,1] = equality constraints
23
24     arguments
25         time double
26         state double
27         control double
28         params struct
29     end
30
31     % No path constraints

```

```
32     c = [];  
33     ceq = [];  
34     end
```

Código Fonte 47 – Função pathConstraints do problema do eVTOL

A.7.9 boundaryObjective.m

```
1  function cost = boundaryObjective(x0, xF, t0, tF, p)  
2      % Evaluates boundary cost for the EVTOL problem  
3      %  
4      % Inputs:  
5      %   x0: [5,1] = initial state vector [sx; sy; vx; vy; E]  
6      %   xF: [5,1] = final state vector [sx; sy; vx; vy; E]  
7      %   t0: double = initial time  
8      %   tF: double = final time  
9      %   p: parameter struct  
10     %  
11     % Outputs:  
12     %   cost: scalar cost value  
13  
14     arguments  
15         x0 double  
16         xF double  
17         t0 double  
18         tF double  
19         p struct  
20     end  
21  
22     % Extract final energy from state  
23     finalEnergy = xF(5);  
24  
25     % Minimize final energy  
26     cost = finalEnergy;  
27     end
```

Código Fonte 48 – Função boundaryObjective do problema do eVTOL

A.7.10 pathObjective.m

```

1  function L = pathObjective(time, state, control, params)
2      % Evaluates running cost for the EVTOL problem
3      %
4      % Inputs:
5      %   time: [1,n] = time vector
6      %   state: [5,n] = state vector [x; y; vx; vy; E]
7      %   control: [2,n] = control vector [Tx; Ty]
8      %   params: parameter struct
9      %
10     % Outputs:
11     %   L: [1,n] = instantaneous cost value
12
13     arguments
14         time double
15         state double
16         control double
17         params struct
18     end
19
20     % No running cost (only terminal cost)
21     L = zeros(1, length(time));
22 end

```

Código Fonte 49 – Função pathObjective do problema do eVTOL

A.7.11 physicalInitialGuess.m

```

1  function zGuess = physicalInitialGuess(problem, plotFlag)
2      % Generate a physics-based initial guess for the EVTOL problem
3      if nargin < 2
4          plotFlag = false;
5      end
6
7      % Extract problem parameters
8      params = problem.getParameters();
9      mass = params.aircraft.mass;
10     g = params.environment.GRAVITY;

```

```

11
12     % Extract time bounds, boundary conditions, and grid size
13     [t0, tF] = problem.getTimeBounds();
14     [x0, xF] = problem.getBoundaryConditions();
15     nGrid = problem.getGridSize();
16
17     % Generate time vector and relative time
18     time = linspace(t0, tF, nGrid(1));
19     tRel = time - t0; % Shift time to start at 0
20     duration = tF - t0;
21
22     % Calculate trajectory using relative time
23     posX = x0(1) + ...
24             x0(3) * tRel + ...
25             (3 * (xF(1) - x0(1)) / duration^2 - ...
26             2 * x0(3) / duration - xF(3) / duration) * tRel.^2 + ...
27             (-2 * (xF(1) - x0(1)) / duration^3 + ...
28             (x0(3) + xF(3)) / duration^2) * tRel.^3;
29     posY = x0(2) + ...
30             x0(4) * tRel + ...
31             (3 * (xF(2) - x0(2)) / duration^2 - ...
32             2 * x0(4) / duration - xF(4) / duration) * tRel.^2 + ...
33             (-2 * (xF(2) - x0(2)) / duration^3 + ...
34             (x0(4) + xF(4)) / duration^2) * tRel.^3;
35     velX = gradient(posX, time); % Note: gradient still uses
36                                   % original time step
37     velY = gradient(posY, time);
38
39     % Add validation
40     assert(all(isfinite(velX)), 'Non-finite values in velX');
41     assert(all(isfinite(velY)), 'Non-finite values in velY');
42
43     % Calculate minimum thrust needed to maintain trajectory
44     [lift, drag] = computeLiftDrag(velX, velY, params);
45     gamma = computeFlightAngle(velX, velY);
46
47     % Solve for required thrust
48     thrustX = drag .* cos(gamma) + lift .* sin(gamma);
49     thrustY = drag .* sin(gamma) - lift .* cos(gamma) + mass .* g;
50

```

```

51     % Calculate power at this point
52     state = [posX; posY; velX; velY; zeros(1, length(time))];
53     control = [thrustX; thrustY];
54     dx = evtolDynamics(time, state, control, params);
55     dE = dx(5, :);
56
57     % Integrate power to get energy
58     energy = x0(5) + cumtrapz(time, dE);
59
60     state(5, :) = energy;
61
62     % Combine into guess
63     zGuess = [state; control];
64
65     % Plot if requested
66     if plotFlag
67         guess = struct();
68         guess.time = time;
69         guess.state = state;
70         guess.control = control;
71         plotResults('Initial Guess', guess);
72     end
73 end

```

Código Fonte 50 – Função physicalInitialGuess do problema do eVTOL

A.7.12 checkConstraints.m

```

1     % Auxiliary function to check constraints
2     function violations = checkConstraints(time, state, control, params)
3         % Initialize violations structure
4         violations = struct();
5         violations.state = struct();
6         violations.control = struct();
7         violations.physical = struct();
8
9         % Check state bounds
10        if any(state(1, :) > 1000 | state(1, :) < 0)
11            violations.state.sx = struct( ...

```



```

12         'max', max(state(1, :)), ...
13         'min', min(state(1, :)), ...
14         'bounds', [0, 1000], ...
15         'numViolations', ...
16         sum(state(1, :) > 1000 | ...
17             state(1, :) < 0));
18     end
19     if any(state(2, :) > 110 | state(2, :) < -10)
20         violations.state.sy = struct( ...
21             'max', max(state(2, :)), ...
22             'min', min(state(2, :)), ...
23             'bounds', [-10, 110], ...
24             'numViolations', ...
25             sum(state(2, :) > 110 | ...
26                 state(2, :) < -10));
27     end
28     if any(state(3, :) > 35 | state(3, :) < 0)
29         violations.state.vx = struct( ...
30             'max', max(state(3, :)), ...
31             'min', min(state(3, :)), ...
32             'bounds', [0, 35], ...
33             'numViolations', ...
34             sum(state(3, :) > 35 | ...
35                 state(3, :) < 0));
36     end
37     if any(state(4, :) > 6 | state(4, :) < -5)
38         violations.state.vy = struct( ...
39             'max', max(state(4, :)), ...
40             'min', min(state(4, :)), ...
41             'bounds', [-5, 6], ...
42             'numViolations', ...
43             sum(state(4, :) > 6 | ...
44                 state(4, :) < -5));
45     end
46
47     % Check control bounds
48     if any(control(1, :) > 1800 | control(1, :) < 0)
49         violations.control.Tx = struct( ...
50             'max', max(control(1, :)), ...
51             'min', min(control(1, :)), ...

```

```

52         'bounds', [0, 1800], ...
53         'numViolations', ...
54         sum(control(1, :) > 1800 | ...
55             control(1, :) < 0));
56     end
57     if any(control(2, :) > 2600 | control(2, :) < 0)
58         violations.control.Ty = struct( ...
59             'max', max(control(2, :)), ...
60             'min', min(control(2, :)), ...
61             'bounds', [0, 2600], ...
62             'numViolations', ...
63             sum(control(2, :) > 2600 | ...
64                 control(2, :) < 0));
65     end
66
67     % Check physical consistency
68     V = sqrt(state(3, :).^2 + state(4, :).^2);
69     gamma = atan2(state(4, :), state(3, :));
70
71     violations.physical.velocity = struct( ...
72         'max', max(V), ...
73         'min', min(V), ...
74         'mean', mean(V));
75
76     violations.physical.flightPathAngle = struct( ...
77         'max', ...
78         max(abs(gamma)) * 180 / pi, ...
79         'min', ...
80         min(gamma) * 180 / pi, ...
81         'mean', ...
82         mean(gamma) * 180 / pi);
83 end

```

Código Fonte 51 – Função checkConstraints do problema do eVTOL

A.7.13 testAeroForces.m

```

1 function testAeroForces()
2     % Test function to validate lift and drag calculations

```

```

3      % across different angles of attack
4
5      params = evtolParams();
6
7      % Extract parameters
8      AIR_DENSITY = params.environment.AIR_DENSITY;
9      wingS = params.aircraft.wing.S;
10     wingCLO = params.aircraft.wing.CLO;
11     wingCLalpha = params.aircraft.wing.CLalpha;
12     wingAlphaFus = params.aircraft.wing.alphaFus;
13     wingM = params.aircraft.wing.sigmoid.M;
14     wingAlpha0 = params.aircraft.wing.sigmoid.alpha0;
15
16     % Test velocities
17     V = 25; % m/s
18     alphaRange = (-30:1:30) * (pi / 180); % Convert degrees to radians
19
20     % Initialize arrays
21     lift = zeros(size(alphaRange));
22     drag = zeros(size(alphaRange));
23
24     % Calculate forces for each angle of attack
25     for i = 1:length(alphaRange)
26         alpha = alphaRange(i);
27         velX = V * cos(alpha);
28         velY = V * sin(alpha);
29         [lift(i), drag(i)] = computeLiftDrag(velX, velY, params);
30     end
31
32     % Calculate theoretical values for comparison
33     % Linear region lift coefficient (before stall)
34     linearCL = wingCLO + wingCLalpha * (alphaRange + wingAlphaFus);
35     linearLift = 0.5 * AIR_DENSITY * V^2 * wingS .* linearCL;
36
37     % Plot results
38     figure('Name', 'Aerodynamic Forces Validation');
39
40     % Plot Lift
41     subplot(2, 2, 1);
42     plot(alphaRange * 180 / pi, lift, 'b-', 'LineWidth', 2, ...

```

```

43         'DisplayName', 'Computed Lift');
44     hold on;
45     plot(alphaRange * 180 / pi, linearLift, 'r--', 'LineWidth', 1.5, ...
46         'DisplayName', 'Linear Theory');
47     grid on;
48     xlabel('Angle of Attack (degrees)');
49     ylabel('Lift Force (N)');
50     title('Lift vs Angle of Attack');
51     legend('show');
52
53     % Plot Drag
54     subplot(2, 2, 2);
55     plot(alphaRange * 180 / pi, drag, 'b-', 'LineWidth', 2);
56     grid on;
57     xlabel('Angle of Attack (degrees)');
58     ylabel('Drag Force (N)');
59     title('Drag vs Angle of Attack');
60
61     % Plot L/D ratio
62     subplot(2, 2, 3);
63     plot(alphaRange * 180 / pi, lift ./ drag, 'b-', 'LineWidth', 2);
64     grid on;
65     xlabel('Angle of Attack (degrees)');
66     ylabel('L/D Ratio');
67     title('Lift-to-Drag Ratio');
68
69     % Print key values
70     [maxLD, maxLDIdx] = max(lift ./ drag);
71     alphaMaxLD = alphaRange(maxLDIdx) * 180 / pi;
72
73     [maxL, maxLIdx] = max(lift);
74     alphaMaxL = alphaRange(maxLIdx) * 180 / pi;
75
76     fprintf('\nKey Performance Parameters:\n');
77     fprintf('Maximum L/D Ratio: %.2f at %.1f degrees\n', ...
78         maxLD, alphaMaxLD);
79     fprintf('Maximum Lift: %.2f N at %.1f degrees\n', maxL, alphaMaxL);
80     fprintf('Stall Angle: %.1f degrees (theoretical)\n', ...
81         wingAlpha0 * 180 / pi);
82

```

```

83     % Verify transition between linear and flat plate models
84     subplot(2, 2, 4);
85     straightCL = wingCL0 + wingCLalpha * (alphaRange + wingAlphaFus);
86     M = wingM;
87     alpha0 = wingAlpha0;
88
89     sigMinus = exp(-M * (alphaRange - alpha0));
90     sigPlus = exp(M * (alphaRange + alpha0));
91     sigma = (1 + sigMinus + sigPlus) ./ ...
92             ((1 + sigMinus) .* (1 + sigPlus));
93
94     flatPlateCL = 2 * sign(alphaRange) .* ...
95                 sin(alphaRange).^2 .* cos(alphaRange);
96     CLCombined = sigma .* flatPlateCL + (1 - sigma) .* straightCL;
97
98     plot(alphaRange * 180 / pi, straightCL, 'r--', 'LineWidth', 1.5, ...
99          'DisplayName', 'Linear Model');
100    hold on;
101    plot(alphaRange * 180 / pi, flatPlateCL, 'g--', 'LineWidth', 1.5, ...
102         'DisplayName', 'Flat Plate');
103    plot(alphaRange * 180 / pi, CLCombined, 'b-', 'LineWidth', 2, ...
104         'DisplayName', 'Combined Model');
105    grid on;
106    xlabel('Angle of Attack (degrees)');
107    ylabel('Lift Coefficient');
108    title('Lift Coefficient Models');
109    legend('show');
110 end

```

Código Fonte 52 – Função testAeroForces do problema do eVTOL

A.7.14 testEvtoldynamics.m

```

1     % Test function to verify dynamics implementation
2     function testEvtoldynamics()
3         % Setup test parameters matching your current configuration
4         params = evtolParams();
5         GRAVITY = params.environment.GRAVITY;
6         AIR_DENSITY = params.environment.AIR_DENSITY;

```

```

7      mass = params.aircraft.mass;
8      rotorArea = params.aircraft.rotorArea;
9      chi = params.aircraft.chi;
10
11      % Test Case 1: Hover condition
12      hoverState = [0; 0; 0; 0; 0];
13      hoverControl = [0; mass * GRAVITY];
14      disp('Hover Test:');
15      dxHover = evtoldynamics(hoverState, hoverControl, params);
16      disp(['    Vertical acceleration: ' num2str(dxHover(4))]);
17      disp(['    Power consumption: ' num2str(dxHover(5))]);
18      % Debug output
19      disp('Hover calculations:');
20      disp(['    Total thrust: ' num2str(hoverControl(2)) ' N']);
21      disp(['    Thrust per rotor: ' num2str(hoverControl(2) / 4) ' N']);
22      disp(['    Hover induced velocity: ' ...
23          num2str(sqrt(hoverControl(2) / (8 * AIR_DENSITY * rotorArea))) ...
24          ' m/s']);
25      disp(['    Total power (computed): ' num2str(dxHover(5)) ' W']);
26
27      disp(' ');
28
29      % Test Case 2: Forward flight
30      xForward = [0; 0; 25; 0; 0];
31      % Calculate required thrust for steady forward flight
32      [lift, drag] = computeLiftDrag(xForward(3), xForward(4), params);
33      uForward = [drag; mass * GRAVITY - lift];
34      disp('Forward Flight Test:');
35      dxForward = evtoldynamics(xForward, uForward, params);
36      disp(['    Lift force: ' num2str(lift) ' N']);
37      disp(['    Drag force: ' num2str(drag) ' N']);
38      disp(['    Required thrust - horizontal: ' ...
39          num2str(uForward(1)) ' N']);
40      disp(['    Required thrust - vertical: ' ...
41          num2str(uForward(2)) ' N']);
42      disp(['    Horizontal acceleration: ' ...
43          num2str(dxForward(3))]);
44      disp(['    Vertical acceleration: ' ...
45          num2str(dxForward(4))]);
46

```

```

47     disp(' ');
48
49     % Test Case 3: Climbing flight
50     xClimb = [0; 0; 20; 5; 0];
51     gamma = atan2(5, 20);
52     [lift, drag] = computeLiftDrag(xClimb(3), xClimb(4), params);
53     requiredThrust = [drag * cos(gamma) + lift * sin(gamma); ...
54                     drag * sin(gamma) - lift * cos(gamma) + ...
55                     mass * GRAVITY];
56     climbControl = requiredThrust;
57     disp('Climbing Flight Test:');
58     dxClimb = evtolDynamics(xClimb, climbControl, params);
59     disp(['    Lift force: ' num2str(lift) ' N']);
60     disp(['    Drag force: ' num2str(drag) ' N']);
61     disp(['    Required thrust - horizontal: ' ...
62         num2str(climbControl(1)) ' N']);
63     disp(['    Required thrust - vertical: ' ...
64         num2str(climbControl(2)) ' N']);
65     disp(['    Horizontal acceleration: ' ...
66         num2str(dxClimb(3))]');
67     disp(['    Vertical acceleration: ' ...
68         num2str(dxClimb(4))]');
69
70     disp(' ');
71
72     % Verify energy model consistency
73     % For hover:  $P = T \cdot v_{induced}$ 
74     hoverThrust = hoverControl(2) / 4; % Per rotor
75     hoverInducedVel = sqrt(hoverThrust / ...
76                         (2 * AIR_DENSITY * rotorArea));
77     hoverPowerTheoretical = 4 * hoverThrust * hoverInducedVel * ...
78                         (1 + chi);
79     disp('Energy Model Test:');
80     disp(['    Hover power (computed): ' ...
81         num2str(dxHover(5)) ' W']);
82     disp(['    Hover power (theoretical): ' ...
83         num2str(hoverPowerTheoretical) ' W']);
84 end

```

FOLHA DE REGISTRO DO DOCUMENTO			
1. CLASSIFICAÇÃO/TIPO TC	2. DATA 25 de novembro de 2024	3. DOCUMENTO Nº DCTA/ITA/DM-115/2024	4. Nº DE PÁGINAS 134
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; MATLAB; eVTOL			
9. PALAVRAS-CHAVE RESULTANTES DE INDEXAÇÃO: Otimização de trajetória; Controle ótimo; Aeronaves de decolagem vertical; Controle			
10. APRESENTAÇÃO:		(X) Nacional () Internacional	
Trabalho de Graduação, ITA, São José dos Campos, 2024. 134 páginas.			
11. RESUMO: Este trabalho apresenta o desenvolvimento de uma biblioteca em MATLAB para solução de problemas de otimização de trajetória utilizando o método de colocação direta trapezoidal. A biblioteca visa simplificar a resolução deste tipo de problema, permitindo que o usuário concentre-se apenas na modelagem, sem necessidade de implementar a lógica matemática da solução numérica. A implementação é validada através de cinco casos de teste: um problema de movimento unidimensional simples, um problema de pêndulo invertido, um problema de manobra de mudança de faixa, o problema clássico da braquistócrona e um problema de otimização de trajetória de subida de uma aeronave eVTOL. Os resultados obtidos são comparados com soluções analíticas, quando disponíveis, e com resultados da literatura utilizando outros softwares de otimização. O trabalho contribui para a área de controle ótimo ao disponibilizar uma ferramenta que facilita a implementação de soluções numéricas para problemas de otimização de trajetória, sendo particularmente útil para aplicações em engenharia aeroespacial e áreas correlatas.			
12. GRAU DE SIGILO: (X) OSTENSIVO () RESERVADO () SECRETO			