Notas em Matemática Aplicada

ISSN 2175-3385
e-ISSN 2236-5915

Volume 88, 2019

Editores

Alexandre Loureiro Madureira (Editor Chefe)
Laboratório Nacional de Computação Científica - LNCC
Petrópolis, RJ, Brasil

Amanda Liz Pacífico Manfrim Perticarrari
Universidade Estadual Paulista Júlio de Mesquita Filho - UNESP
Jaboticabal, SP, Brasil

Max Oliveira de Souza
Universidade Federal Fluminense - UFF
Niterói, RJ, Brasil

Eduardo V. O. Teixeira (Editor Executivo)
University of Central Florida - UCF
Orlando, FL, EUA

Lilian Markenzon
Universidade Federal do Rio de Janeiro - UFRJ
Rio de Janeiro, RJ, Brasil

Sandra Augusta Santos
Universidade Estadual de Campinas - UNICAMP
Campinas, SP, Brasil

SBMAC **Sociedade Brasileira de Matemática Aplicada e Computacional**

2019

A Sociedade Brasileira de Matemática Aplicada e Computacional - SBMAC publica, desde as primeiras edições do evento, monografias dos cursos que são ministrados nos CNMAC.

Para a comemoração dos 25 anos da SBMAC, que ocorreu durante o XXVI CNMAC em 2003, foi criada a série **Notas em Matemática Aplicada** para publicar as monografias dos minicursos ministrados nos CNMAC, o que permaneceu até o XXXIII CNMAC em 2010.

A partir de 2011, a série passa a publicar, também, livros nas áreas de interesse da SBMAC. Os autores que submeterem textos à série Notas em Matemática Aplicada devem estar cientes de que poderão ser convidados a ministrarem minicursos nos eventos patrocinados pela SBMAC, em especial nos CNMAC, sobre assunto a que se refere o texto.

O livro deve ser preparado em **Latex (compatível com o Miktex versão 2.9), as figuras em eps** e deve ter entre **80 e 150 páginas**. O texto deve ser redigido de forma clara, acompanhado de uma excelente revisão bibliográfica e de **exercícios de verificação de aprendizagem** ao final de cada capítulo.

Veja todos os títulos publicados nesta série na página
http://www.sbmac.org.br/p_notas.php

SBMAC **Sociedade Brasileira de Matemática Aplicada e Computacional**

2019

Aproximações de Variedades Definidas Implicitamente Utilizando Técnicas de Contagem e Enumeração

Antonio Castelo Filho
castelo@icmc.usp.br
Lucas Moutinho Bueno
lucas@icmc.usp.br

Departamento de Matemática Aplicada e Estatística
Instituto de Ciências Matemáticas e de Computação
Universidade de São Paulo

Sociedade Brasileira de Matemática Aplicada e Computacional

São Carlos - SP, Brasil
2019

Coordenação Editorial: Luiz Mariano Carvalho

Coordenação Editorial da Série: Alexandre L. Madureira

Editora: SBMAC

Capa: Matheus Botossi Trindade

Patrocínio: SBMAC

Copyright ©2019 by Antonio Castelo Filho e Lucas Moutinho Bueno. Direitos reservados, 2019 pela SBMAC. A publicação nesta série não impede o autor de publicar parte ou a totalidade da obra por outra editora, em qualquer meio, desde que faça citação à edição original.

Catalogação elaborada pela Biblioteca do IBILCE/UNESP
Bibliotecária: Maria Luiza Fernandes Jardim Froner

Castelo, Antonio
Aproximações de Variedades Definidas Implicitamente Utilizando Técnicas de Contagem e Enumeração - São Carlos, SP :
SBMAC, 2019, 108 p., 21.5 cm - (Notas em Matemática Aplicada; v. 88)

ISBN 978-85-8215-091-7 e-ISBN 978-85-8215-090-0

1. Variedades Definidas Implicitamente 2. Aproximação Linear por Partes
3. Marching Simplex 4. Continuation Simplex
I. Castelo, Antonio II. Bueno, Lucas M.
IV. Título. V. Série

CDD - 51

A todos que transformam
Geometria em Álgebra para a
construção de Algoritmos
para visualizar formas
em altas dimensões.

Agradecimentos

As figuras deste livro foram geradas por um software de visualização desenvolvido por Gabriel Scalet Bicalho, a quem agradecemos, em um projeto de final de curso do curso de Ciências da Computação do ICMC.

Agradecemos também a Douglas Cedrim Oliveira pela parceria em um minicurso entitulado "Contagem, Enumeração e Algumas Aplicações em Matemática", ministrado no II Colóquio da Região Sudeste realizado no ICMC-USP em 2012. Este minicurso incentivou a escrita deste livro.

Por fim, mas não menos importante, agradecemos ao Professor Geovan Tavares dos Santos pela orientação e contribuição para a área de processamentos geométrico no Brasil que deu origem a vários trabalhos na linha de pesquisa que este livro se concentra.

Conteúdo

Prefácio **xiii**

1 Introdução **1**
1.1 Aproximação de Variedades Implícitas 1
1.2 Aplicações em alta dimensão 2
1.2.1 Física Moderna 3
1.2.2 Processamento de imagens 4D 3
1.3 Revisão de Métodos para Aproximação de Variedades 4
1.3.1 Métodos baseados em cubos 4
1.3.2 Métodos baseados em triangulação 5
1.3.3 Outros métodos de aproximação por partes 6
1.3.4 Problemas correlatos 6
1.4 Contagem e Enumeração 7
1.5 Marching Simplex e Continuation Simplex 7
1.6 Organização do Livro 8

2 Aproximações de Variedades Implícitas **9**
2.1 Conceitos Básicos sobre Variedades 9
2.2 Conceitos Básicos sobre Triangulações 10
2.3 Interpolação Linear Simplicial 11
2.4 Esquema de Diferenças Simplicial 12
2.5 Resultados sobre Aproximações Simpliciais 13
2.6 As Triangulações K_1 e J_1 15
2.7 Exercícios 16

3 Contagem e Enumeração **17**
3.1 Produtos Cartesianos Discretos 17
3.1.1 Definição 17
3.1.2 Exemplos 17
3.1.3 Programas em Matlab/Octave 19
3.2 Permutações 22
3.2.1 Definição 22
3.2.2 Exemplos 22
3.2.3 Programas em Matlab/Octave 23
3.3 Combinações 24
3.3.1 Definição 24
3.3.2 Exemplos 25
3.3.3 Programas em Matlab/Octave 25
3.4 Exercícios 26

4 Métodos Baseados em Simplexos **27**
4.1 Representação de Simplexos e suas Faces . . . 27
4.2 Os Vértices da Aproximação . . . 28
4.3 Perturbação . . . 29
4.4 Regras de Pivoteamento . . . 30
4.5 Esqueleto Combinatório . . . 34
4.6 Marching Simplex . . . 35
4.6.1 Parâmetros de Entrada e Saída . . . 36
4.7 Continuation Simplex . . . 37
4.7.1 Métodos de Continuação . . . 37
4.7.2 Condição Inicial . . . 38
4.7.3 Estruturas de Dados . . . 40
4.7.4 Parâmetros de Entrada e Saída . . . 41
4.8 Exemplos . . . 42
4.8.1 Exemplos da Utilização do Marching Simplex . . . 42
4.8.2 Exemplos da Utilização do Continuation Simplex . . . 49
4.9 Exercícios . . . 54
4.10 Projetos . . . 55

5 Modelagem Implícita **57**
5.1 Combinação de Hipersuperfícies . . . 57
5.2 Exemplos . . . 59
5.3 Exercícios . . . 65
5.4 Projetos . . . 65

A Códigos Auxiliares **71**
A.1 Os Vértices da Aproximação . . . 71
A.2 Perturbação . . . 72
A.3 Regras de Pivoteamento . . . 74
A.4 Esqueleto Combinatório . . . 77
A.5 Condição Inicial . . . 80
A.6 Estrutura de Dados . . . 83

B Código Marching Simplex **87**

C Código Continuation Simplex **91**

Prefácio

Este livro é destinado a alunos e profissionais de matemática aplicada, ciência da computação e áreas afins que estejam interessados em implementar um algoritmo que aproxima os zeros de funções em quaisquer dimensões por uma malha de politopos, seja para representação gráfica, seja para outras aplicações geométricas.

O livro inclui uma revisão bibliográfica de métodos de aproximação de variedades definidas implicitamente, exemplos de aplicações em alta dimensão, técnicas de contagem e enumeração usadas para resolução eficiente do problema, e dois algoritmos completos para este fim: um mais simples que varre todos os simplexos do domínio e outro que varre somente os simplexos que são transversais a variedade.

O texto também acompanha códigos em Matlab/Octave dos algoritmos apresentados.

São Carlos, 01 de setembro de 2019.

Antonio Castelo Filho
Lucas Moutinho Bueno

Capítulo 1

Introdução

Este livro apresenta dois algoritmos, o Marching Simplex e o Continuation Simplex, para aproximação de variedades implícitas de qualquer dimensão usando técnicas de contagem e enumeração. Estes algoritmos são baseados nos trabalhos de Allgower e Georg [1] e de Antonio Castelo [9]. O livro também acompanha códigos de Matlab/Octave usados para implementar os algoritmos. Os códigos estão disponíveis em "https://github.com/antoniocastelofilho/marching_simplex".

As figuras deste livro foram geradas por um software de visualização desenvolvido por Gabriel Scalet Bicalho, em um projeto de final de curso orientado pelo professor Antonio Castelo Filho. A entrada desse software de visualização é justamente a saída do código Matlab do Marching Simplex ou do Continuation Simplex. O código do software de visualização pode ser obtido em "https://github.com/GSBicalho/TrueNgine".

1.1 Aproximação de Variedades Implícitas

Dada uma função de classe C^1, $F : \mathbb{R}^n \to \mathbb{R}^k$, $n \geq k$, o conjunto dos pontos onde F se anula:

$$\mathcal{M} = F^{-1}(0) = \{x \in \mathbb{R}^n \mid F(x) = 0\},$$

é uma variedade definida implicitamente se para todo ponto em $\mathcal{M}$ a derivada de F neste ponto tem posto máximo. Neste caso a variedade $\mathcal{M}$ tem dimensão $n - k$ (veja o Teorema 2.1 da Seção 2.1 do Capítulo 2). Os casos mais comuns geralmente vistos nos cursos de cálculo são:

- Para $n = 2$ e $k = 1$, $F^{-1}(0)$ é uma curva em $\mathbb{R}^2$;
- Para $n = 3$ e $k = 2$, $F^{-1}(0)$ é uma curva em $\mathbb{R}^3$;
- Para $n = 3$ e $k = 1$, $F^{-1}(0)$ é uma superfície em $\mathbb{R}^3$.

Para visualizar uma variedade implícita, há basicamente dois tipos de técnicas computacionais: traçado de raio (ray tracing) e aproximação por politopos. O primeiro gera imagens mais realistas, principalmente em termos de iluminação, mas é mais lento. O segundo é menos realista, mas é mais simples, rápido e flexível. Neste livro vamos focar exclusivamente em aproximação de variedades implícitas por politopos. Para maiores informações sobre técnicas de traçado de raio, recomendamos [23].

Para se ter uma ideia melhor de uma aproximação de variedade, a Figura 1.1 mostra um toro tridimensional, $S^2 \times S^1$ (S^1 é uma esfera unidimensional e S^2 é uma esfera bidimensional). Neste exemplo, $F : \mathbb{R}^5 \to \mathbb{R}^2$ é definida por

$$\begin{cases} F_1(x,y,z,u,v) & = x^2 + y^2 + z^2 \quad -1 \\ F_2(x,y,z,u,v) & = u^2 + v^2 \qquad\quad -\frac{1}{4} \end{cases}$$

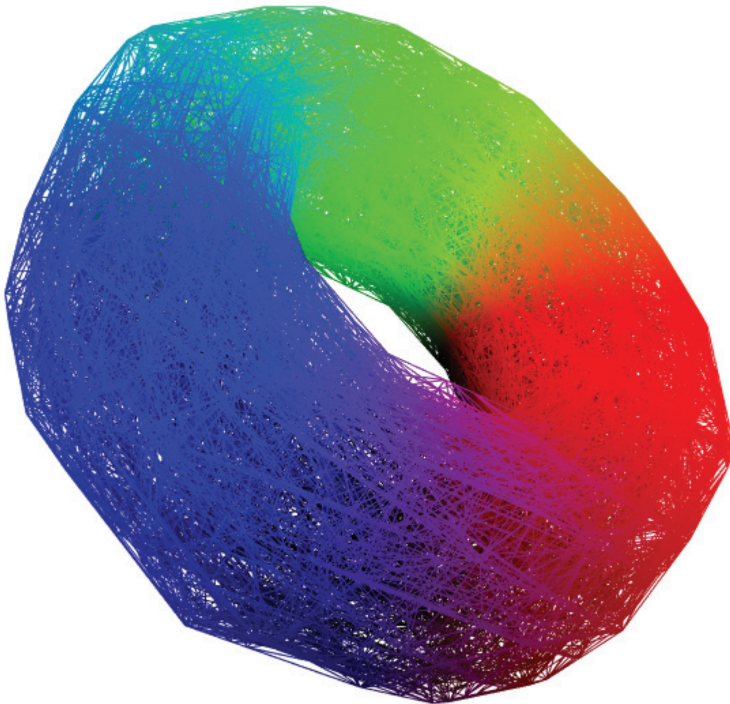

Figura 1.1: Projeção da aproximação de um toro $S^2 \times S^1$ definido em $\mathbb{R}^5$.

1.2 Aplicações em alta dimensão

Variedades definidas implicitamente aparecem em várias aplicações em duas ou três dimensões, tais como: modelagem geométrica, modelagem molecular, animação, jogos digitais, geometria computacional, métodos numéricos para equações diferenciais com restrição e bifurcações, dentre outras. Métodos tradicionais de aproximação de variedades implícitas podem ser usados para essas aplicações.

Um dos principais diferenciais dos algoritmos descritos neste livro é sua generalização para qualquer dimensão. O leitor pode se perguntar qual a utilidade de se trabalhar em dimensões maiores que três, se o mundo que percebemos tem somente três dimensões. Essa seção responde brevemente esta pergunta ao mostrar algumas aplicações com variedades de dimensão maior ou igual a 4.

1.2.1 Física Moderna

A física clássica newtoniana ensinava até o início do século XX que o mundo era tridimensional e que o espaço e o tempo eram variáveis independentes. Apesar da teoria clássica funcionar para a maior parte de nossa vida cotidiana, ela se mostrou equivocada em diversos casos desde a física de partículas sub-atômicas até a cosmologia.

Espaço-tempo

A teoria da relatividade geral desenvolvida por Albert Einstein mostrou uma relação entre o espaço tridimensional e o tempo, criando o espaço-tempo, de quatro dimensões.

Uma forma simples de representar o espaço-tempo é pelo espaço de Minkowski, que combina o espaço euclidiano tri-dimensional com o tempo de tal forma que o intervalo entre quaisquer dois eventos seja independente do frame de referência em que eles foram obtidos. O espaço de Minkowiski é um tipo especial e mais simples de uma variedade de Lorenz que, por sua vez, é um tipo especial de variedade pseudoriemanniana. Mais detalhes sobre variedades pseudoriemannianas e suas aplicações em física podem ser lidas em [43].

Teorias em dimensão maior que 4

A teoria da relatividade geral, apesar de explicar mais fenômenos que a física clássica, não se mostrou suficiente para explicar qualquer fenômeno físico e uma “teoria de tudo”, caso exista, ainda está para ser descoberta.

Ainda no início do século XX, a teoria de Kalusa-Klein foi concebida, e esta uniu gravitação e electromagnetismo em um espaço de 5 dimensões. A teoria foi iniciada por Theodor Kaluza [30] e Oskar Klein a complementou com uma interpretação quântica [32].

Em meados da década de 70 surgiu a teoria da supergravidade [14], que combina a teoria da relatividade com o princípio de supersimetria, advindo de teorias em física de partículas. A supergravidade foi inicialmente formulada em 11 dimensões mas, logo na década de 80, perdeu espaço para formulações de 10 dimensões [24], em um período de pesquisa que ficou conhecido como a primeira revolução das supercordas. Na década de 90, surgiu a Teoria M, que unificou teorias consistentes de supergravidade e de cordas [58], dando início à segunda revolução das supercordas.

Na teoria de supercordas, conjectura-se que as seis dimensões extras além do espaço-tempo tem a forma de uma variedade de Calabi-Yau. Esta variedade foi introduzida por Eugenio Calabi [6], aprofundada por Shing-Tung Yau [59] e nomeada e contextualizada na física por Candelas et al. [8].

Uma forma de reduzir a lacuna de percepção entre entre o universo observável 3D e teorias de múltiplas dimensões é através da compactação de 10 ou 11 dimensões em espaços menores, como a esfera esmagada de 7 dimensões [41].

Para maior aprofundamento no tema, um livro com diversas aplicações de variedades multi-dimensionais em física pode ser encontrado em [12].

1.2.2 Processamento de imagens 4D

Imagens 4D juntam três dimensões de espaço e uma dimensão de tempo em uma única matriz de pixels em quatro dimensões.

Cada coordenada de tempo da imagem indica o frame do qual os valores de espaço foram obtidos. A junção do espaço e tempo se mostra especialmente eficaz para imagens de objetos em movimento, por conta da dificuldade de se identificar dinâmica e oclusões desses objetos em imagens 3D estáticas.

O método proposto neste livro pode usar os próprios valores dos pixels das imagens para calcular a variedade $\mathcal{M}$ em vez de funções analíticas.

Aplicações em Medicina

Em medicina, modelos 4D podem ser usados em imagens de ressonância magnética, tomografia computadorizada e ultra-som.

No caso de ressonância magnética, métodos que utilizam imagens 4D se mostraram eficazes para facilitar o diagnóstico de doenças cardio-vasculares [38]. Eles se diferem a métodos anteriores, tanto pela maior precisão em obter um modelo 3D do coração, como por sua capacidade de calcular o fluxo sanguíneo em todas direções.

No caso de tomografia computadorizada, um novo protocolo de escaneamento foi criado para gerar imagens 4D do pulmão [47]. Comparativamente a protocolos anteriores, este novo possui um tempo de escaneamento menor e obtém imagens de todo o ciclo respiratório.

Em ultra-som aplicado a exames de pré-natal, um método 4D permite a visualização dinâmica de imagens do coração do feto em diversos níveis de profundidade e facilita o diagnóstico de anomalias congênitas[7].

Aplicações em reconstrução de objetos

É de interesse para diversas aplicações, como detecção de eventos, robótica, jogos eletrônicos, animação, interação humano-computador, etc., a reconstrução modelos computacionais 3D a partir de objetos físicos. Geralmente isso é feito com técnicas de visão computacional a partir de imagens 2D ou com dados de sensores, que também podem ser interpretados como imagens binárias.

Quando os objetos a serem reconstruídos estão em movimento, imagens 4D podem ser usadas para identificar dinâmica e oclusões dos objetos e, assim, possibilitar a criação de modelos computacionais de forma eficaz. Técnicas adicionais podem ser usadas para complementar a reconstrução, como propriedades de conservação de massa e topologia [53].

1.3 Revisão de Métodos para Aproximação de Variedades

Esta seção faz uma revisão histórica de métodos de aproximação de variedades implícitas e discursa brevemente sobre problemas correlatos.

1.3.1 Métodos baseados em cubos

Para uma função $F : \mathbb{R}^3 \to \mathbb{R}$ (com as mesmas propriedades apresentadas na Seção 1.1), os pontos onde F se anula geram uma variedade $\mathcal{M}$ bidimensional, também chamada de superfície. Um dos métodos mais conhecidos para aproximar uma superfície é o algoritmo chamado Marching Cubes, publicado por Lorensen e Cline em 1987 [37]. Nesse algoritmo, a região de interesse é subdividida em uma malha uniforme de cubos. Cada cubo da malha é testado para saber se ele contém

uma parte da superfície. Esse teste é feito por meio dos sinais de F nos vértices do cubo e uma tabela de referência: para cada padrão de sinais de F nos vértices, o algoritmo cria uma malha de polígonos determinada pela tabela, que separa vértices de sinais opostos. Quando todos cubos são testados, obtém-se uma aproximação para a superfície.

Um dos problemas do Marching Cubes é a presença de ambiguidades nos testes da malha, que podem resultar em inconsistências topológicas. Uma ambiguidade ocorre quando há mais de uma forma de gerar uma malha para um mesmo padrão de sinais de F nos vértices. Ela é um problema se o algoritmo escolhe uma forma inconsistente com a topologia de $\mathcal{M}$. A Figura 1.2 mostra um exemplo de ambiguidade, onde os vértices destacados tem mesmo sinal quando aplicados por F.

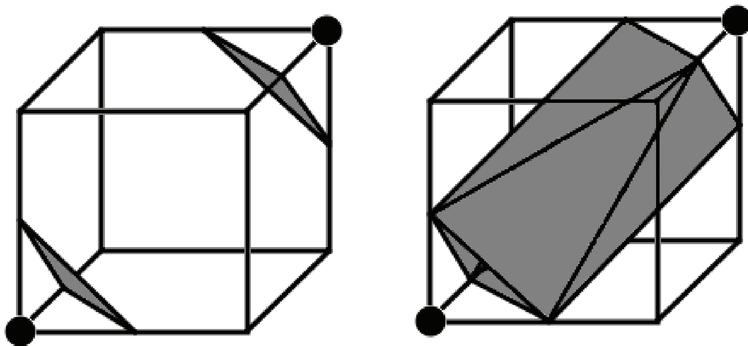

Figura 1.2: Exemplo de ambiguidade gerada pelo algoritmo Marching Cubes. Duas malhas que separam vértices de sinais opostos podem ser geradas.

Trabalhos posteriores resolvem o problema de ambiguidade, sem alterar a essência do algoritmo original: Chernyaev [11] propõe uma tabela de referência extendida, usada também por Lewiner et. al. [34], além de testes de consistência nas faces dos cubos. Já Plantinga e Vegter [48] usam octrees para refinar a malha inicial, aumentando a densidade de cubos em partes ambíguas, e geometria intervalar para garantir consistência topológica.

Mais recentemente, uma extensão do Marching Cubes para $F : \mathbb{R}^n \to \mathbb{R}$ $(n \geq 3)$ foi proposta, mas que necessita de uma tabela de referência de tamanho 2^{2^n} [4], o que torna a extensão custosa computacionalmente.

1.3.2 Métodos baseados em triangulação

Esta seção faz referência ao conceito de simplexo, que é uma generalizão do conceito de triângulo para qualquer dimensão (por exemplo, um tetraedro em dimensão três). Sua definição formal será retomada no Capítulo 2. Uma subdivisão simplicial, ou triangulação, é a subdivisão de um espaço em simplexos.

Outro problema do Marching Cubes é sua difícil extensão para dimensões elevadas, por conta do aumento exponencial da tabela de referência na medida que a dimensão aumenta. Neste caso, os métodos mais utilizados são os baseados em

subdivisão simplicial do domínio e aproximação de F em cada simplexo por uma função mais simples (função afim, por exemplo), encontrando os pontos onde estas aproximações se anulam. Esse tipo de método teve origem em 1967 (antes do Marching Cubes ser criado) no trabalho de Scarf [52], que mapeia um simplexo S n-dimensional para outro pela aproximação de uma função $F : S \in \mathbb{R}^{n+1} \to \mathbb{R}^n$.

Posteriormente, novos métodos de subdivisão simplicial extenderam o trabalho de Scarf usando uma grade de hipercubos no espaço do domínio [16] [17], onde cada hipercubo é subdividido por um padrão de simplexos. Até então, os trabalhos citados de subdivisão simplicial não eram diretamente aplicados em aproximação de variedades. Dentre os pioneiros que perceberam que tais trabalhos podiam ser usados para esse fim, Allgower e Georg tiveram grande importância [1]. Desde então, a imagem de F pôde ter qualquer dimensão k menor que n e variedades implícitas de dimensão $n - k$ puderam ser formadas. Apesar disso, provavelmente o algoritmo mais usado baseado em triangulação até hoje é o Marching Tetrahedra, com $n = 3$ e $k = 1$, original de Doi e Koide [15] e reimplementado por outros autores [26][56].

Métodos baseados em triangulação, embora mais simples e generalizáveis que métodos baseados em cubos, têm a desvantagem de serem geometricamente mais irregulares, precisando de um maior refinamento para a aproximação parecer mais suave. Para amenizar esse problema, foram propostos métodos de aproximação com refinamento adaptativo, ou seja, onde cada região triangulada tem uma densidade de simplexos determinada conforme a necessidade. O refinamento é geralmente baseado na triangulação de quadtrees (2D), octrees (3D) [27] e suas generalizações para dimensão elevada [10][45][57].

1.3.3 Outros métodos de aproximação por partes

Outros métodos de aproximação de variedades implícitas foram propostos na literatura, que não se encaixam exatamente nos dois tipos anteriores.

Em particular, uma série de trabalhos fazem aproximação de F por partes, assim como nos métodos baseados em triangulação, porém usam um domínio subdividido em politopos variados (não necessariamente simplexos). O algoritmo de Bloomenthal [5] é um deles, que inclui refinamento adaptativo por octrees, mas funciona somente para superfícies.

Posteriormente, Henderson [28] cria um método em que cada região do domínio de F é formada por politopos de qualquer dimensão n que, por sua vez, são formados pela interseção de bolas de dimensão n. A imagem de F pode estar em qualquer dimensão $k \leq n$.

1.3.4 Problemas correlatos

Alguns trabalhos mais recentes na literatura resolvem problemas de aproximação de variedades que desviam parcialmente do problema que tratamos neste livro, mas que podem despertar interesse do leitor.

Em computação gráfica, Ohtak et. al. [42] usam pontos dispersos em $\mathbb{R}^3$ para criar superfícies. O algoritmo desenvolvido aproxima os pontos do domínio em funções usando o método de mínimos quadrados. A aproximação é feita por partes e cada parte é uma célula de uma octree (possui, portanto, refinamento adaptativo).

A criação de variedades a partir de pontos dispersos é também um assunto de interesse das áreas de aprendizado de máquina e análise visual de dados, onde técnicas são usadas para aproximar pontos de alta dimensionalidade por varieades e representá-los em dimensões menores. Dentre as técnicas mais conhecidas, temos o

Isomap [29] e o LLE [51], que foram publicados contemporaneamente na prestigiada revista científica Science.

Finalmente, na área de sistemas dinâmicos, métodos foram criados para aproximar variedades invariantes, que resumidamente são variedades que se modificam com o tempo em óbitas periódicas, dentro de um sub-espaço limitado de $\mathbb{R}^n$ [54][25].

1.4 Contagem e Enumeração

À medida que a dimensão n do domínio aumenta, o volume de informações necessárias para todos os dados do domínio cresce exponencialmente. Portanto, uma representação compacta se torna essencial. Os algoritmos apresentados neste livro usam técnicas baseadas em contagem e enumeração para este fim. A contagem é usada especialmente para definir contadores em comandos de repetição. A enumeração serve para discernir os vários dados do problema em tempo de execução, de forma que apenas uma fração mínima dos dados estejam guardados na memória para um mesmo instante de tempo.

Esta área de pesquisa denomina-se Algoritmos Combinatórios e exemplos de livros clássicos são [19], [40] e [44]. Dentre os tipos de técnicas mais importantes que usam contagem e enumeração, podemos citar o produto cartesiano discreto, permutações e combinações. Com o primeiro, podemos simular um ninho de laços em um único comando de repetição em um programa de computador, enumerar malhas cartesianas e enumerar vértices de um hipercubo em qualquer dimensão, dentre outros problemas. A enumeração de combinações é usada quando temos um conjunto de n dados e queremos obter todos os seus subconjuntos de $k \leq n$ dados, como quando queremos listar todas as faces de dimensão k de um simplexo de dimensão n. Por último, mas não menos importante, a enumeração das permutações de um conjunto de n símbolos ordenados é necessária em problemas onde a ordem é requerida, como seguir uma ordem de processamento de simplexos de um hipercubo, evitando-se repetições.

1.5 Marching Simplex e Continuation Simplex

Dois algoritmos são apresentados neste livro para aproximação de variedades definidas implicitamente por uma função $F : \mathbb{R}^n \to \mathbb{R}^k$, $n \geq k$: o Marching Simplex e o Continuation Simplex.

Ambos são métodos baseados em triangulação e criados a partir dos trabalhos de Castelo [9] e, especialmente para o segundo algoritmo, Allgower e Georg [1].

O Marching Simplex usa técnicas de contagem e enumeração pra dividir, em tempo de execução, o domínio em uma malha de hipercubos onde cada hipercubo é subdividido em simplexos de mesma dimensão. Em seguida, verifica-se as faces de dimensão $k + i$, $i = 0, \ldots, n - k$ de cada simplexo que são transversais a $\mathcal{M}$, gerando assim as faces da aproximação.

O Continuation Simplex é uma extensão do Marching Simplex que permite, através de um pré processamento, percorrer apenas hipercubos do domínio que são transversais a F. O Continuation Simplex é especialmente vantajoso para variedades esparsas, onde poucos hipercubos são processados.

1.6 Organização do Livro

O Capítulo 2 define alguns conceitos de topologia e aproximação de funções implícitas que serão usados posteriormente no livro.

O Capítulo 3 apresenta os algoritmos combinatórios que serão usados para a codificação da aproximação de variedades implícitas.

O Capítulo 4 apresenta os algoritmos Marching Simplex e Continuation Simplex para aproximação de variedades implícitas. Seus códigos em Matlab/Octave se encontram nos apêndices deste livro, após as referências.

Finalmente o Capítulo 5 apresenta técnicas de modelagem de hipersuperfícies, permitindo combinar funções implícitas utilizando operações booleanas.

Este livro adota as convenções do software Matlab/Octave para fazer referência a vetores e matrizes, ou às suas partes. Algumas partes de código são mostradas ao longo do livro também em Matlab/Octave.

Capítulo 2

Aproximações de Variedades Implícitas

Este capítulo apresenta definições sobre variedades, triangulações e aproximação de funções por interpolação linear por partes. Ele também cita alguns teoremas da literatura que são importantes para que os algoritmos apresentados neste livro funcionem corretamente. As definições e teroremas apresentados neste capítulo são de grande importância para a leitura do restante do texto, pois os conceitos serão utilizados nos capítulos que tratam de aproximações lineares por partes de variedades definidas implicitamente, que é o foco deste livro. As provas dos teoremas citados não serão apresentadas, porém referências para tais provas se encontram ao longo do texto. As definições e teoremas das primeiras seções serão utilizadas nas demais seções deste capítulo e dos Capítulos 4 e 5.

Este capítulo é matematicamente denso. Caso o leitor esteja interessado primordialmente na parte prática dos algoritmos, e tenha alguma familiaridade com topologia, geometria e interpolação de funções, sugerimos começar pelo o Capítulo 3 e usar este aqui como referência.

2.1 Conceitos Básicos sobre Variedades

Vamos apresentar alguns conceitos básicos sobre variedades diferenciáveis, mais especificamemnte variedades definidas implicitamente. Para detalhes e provas sugerimos a leitura dos livros clássicos "Curso de Análise Vol. 2"[36] e "Variedades Diferenciáveis"de Elon Lages Lima [35], em "Introdução aos Sistemas Dinâmicos"de Jabob Palis Junior e Welington de Melo [46] e em "Toplology form the Differentiable Viewpoint"de John Milnor [39].

Definição 2.1. (*Difeomorfismo*) *Uma aplicação* $f : U \to V$ *é um difeomorfismo de classe* C^r $(1 \leq r \leq \infty)$ *se é uma aplicação invertível e sua inversa* $f^{-1} : V \to U$ *é uma aplicação de classe* C^r.

Definição 2.2. (*Valor Regular*) *Seja* $f : U \subset \mathbb{R}^n \to \mathbb{R}^k$ *uma aplicação de classe* C^r $(r \geq 1)$. $c \in \mathbb{R}^k$ *é valor regular de* f *se para todo* $x \in f^{-1}(c)$, $Df(x) : \mathbb{R}^n \to \mathbb{R}^k$ *tem posto máximo. Se* c *não for valor regular de* f, *diremos que é valor crítico de* f.

Definição 2.3. (*Variedade Diferenciável*) $\mathcal{M} \subset \mathbb{R}^m$ *é uma variedade diferenciável de dimensão* n *e de classe* C^r $(1 \leq r \leq \infty)$ *se para todo* $x \in \mathcal{M}$ *exitirem vizinhanças*

abertas $U \subset \mathbb{R}^m$ de x, $V \subset \mathbb{R}^{n-1} \times \mathbb{R}^+$ e um difeomorfismo de classe C^r de $\mathcal{M} \cap U$ em V. Um difeomorfismo $\varphi : V \to U \cap \mathcal{M}$ é chamado de parametrização da região $U \cap \mathcal{M}$. A fronteira de $\mathcal{M}$, $\partial\mathcal{M}$ é o conjunto dos pontos de $\mathcal{M}$ correspondentes a $\mathbb{R}^{n-1} \times \{0\}$ pelas parametrizações.

Definição 2.4. (*Espaço Tangente*) *Sejam $\mathcal{M}$ uma variedade diferenciável de dimensão n e $\varphi : V \to U \cap \mathcal{M}$ uma parametrização de $U \cap \mathcal{M}$, onde $\varphi(u) = v$. Definimos o espaço vetorial tangente a $\mathcal{M}$ no ponto v como $T_v\mathcal{M} = \{D\varphi(u) \cdot y \mid y \in \mathbb{R}^n\}$, onde $D\varphi(u)$ é o operador derivada da parametrização φ no ponto $u \in V$.*

Teorema 2.1. (*Variedade Definida Implicitamente*) *Sejam $U \subset \mathbb{R}^n$ um aberto e $f : U \to \mathbb{R}^k$ uma aplicação de classe C^r $(1 \leq r \leq \infty)$. Se $c \in \mathbb{R}^k$ é valor regular de f, ou $f^{-1}(c)$ é vazio ou é uma variedade diferenciável de dimensão $n-k$ e de classe C^r em $\mathbb{R}^n$. Além disso, para todo $p \in f^{-1}(c)$, o espaço tangente em p é o núcleo de $Df(p) : \mathbb{R}^n \to \mathbb{R}^k$.*

Teorema 2.2. (*Vizinhança Tubular*) *Seja $F : U \to \mathbb{R}^n$ uma aplicação de classe C^r no aberto $U \subset \mathbb{R}^m$ com $m \geq n$ e $1 \leq r \leq \infty$. Se $c \in \mathbb{R}^n$ é um valor regular de F, então existe uma vizinhança fechada $V_{\mathcal{M}}$ da variedade $\mathcal{M} = F^{-1}(c)$ constituída de variedades, isto é, se $x \in V_{\mathcal{M}}$, então $F^{-1}(F(x))$ é uma variedade contida em $V_{\mathcal{M}}$.*

Definição 2.5. (*Conjunto de Medida Nula*) *Uma conjunto $X \subset \mathbb{R}^n$ é um conjunto de medida nula em $\mathbb{R}^n$ se para cada $\epsilon > 0$ dado, é possível obter uma sequência de hipercubos de dimensão n abertos $C_1, C_2, \ldots, C_k, \ldots$ em $\mathbb{R}^n$ tais que $X \subset \cup_{k=1}^{\infty} C_k$ e $\sum_{k=1}^{\infty} vol(C_k) < \epsilon$.*

Definição 2.6. (*Conjunto Denso*) *Uma conjunto $X \subset \mathbb{R}^n$ é um conjunto denso em $\mathbb{R}^n$ se o complementar $R^n - X$ tem medida nula em $\mathbb{R}^n$.*

Teorema 2.3. (*Sard*) *Seja $f : U \subset \mathbb{R}^n \to \mathbb{R}^k$ uma aplicação de classe C^r $(r \geq max\{1, n-k+1\})$. O conjunto dos valores regulares de f é aberto e denso em $\mathbb{R}^k$.*

Definição 2.7. (*Subvariedade Diferenciável*) *Seja $\mathcal{M} \subset \mathbb{R}^m$ uma variedade diferenciável de dimensão n e de classe C^r $(1 \leq r \leq \infty)$. $\mathcal{N} \subset \mathcal{M}$ é uma subvariedade diferenciável de dimensão k de $\mathcal{M}$ e de classe C^s $(1 \leq s \leq r)$ se para todo $x \in \mathcal{N}$ exitirem vizinhanças abertas $U \subset \mathbb{R}^m$ de x, $V \subset \mathbb{R}^{k-1} \times \mathbb{R}^+$ e um difeomorfismo de classe C^s de $\mathcal{N} \cap U$ em V.*

Definição 2.8. (*Transversalidade*) *Sejam $\mathcal{M}$ e $\mathcal{N}$ duas variedades de dimensão m e n respectivamente em $\mathbb{R}^k$ $(k \geq max\{m, n\})$ e de classe C^r $(1 \leq r \leq \infty)$. Dizemos que $\mathcal{M}$ e $\mathcal{N}$ são transversais em um ponto $p \in \mathcal{M} \cap \mathcal{N}$ se $T\mathcal{M}_p + T\mathcal{N}_p = \mathbb{R}^k$. Se $\mathcal{M} \cap \mathcal{N} = \emptyset$, então $\mathcal{M}$ e $\mathcal{N}$ são transversais por vacuidade.*

2.2 Conceitos Básicos sobre Triangulações

Para maior aprofundamento no tema apresentado nesta e nas próximas seções, é sugerida a leitura da tese [9], do livro [1], do artigo [10] e do curso no proceedings [18].

Definição 2.9. (*Espaço Afim*) *Dados os pontos $v_0, v_1, \ldots, v_k \in \mathbb{R}^n$, chama-se o conjunto $aff(v_0, \ldots, v_k) = \{v \in \mathbb{R}^n \mid \sum_{i=0}^{k} \lambda_i v_i = v$ e $\sum_{i=0}^{k} \lambda_i = 1\}$ de conjunto afim gerado pelos pontos $v_0, \ldots, v_k$.*

Definição 2.10. (*Dimensão*) *Chama-se de dimensão de $aff(v_0, \ldots, v_k)$ e denota-se por $dim(aff(v_0, \ldots, v_k))$ o maior número de vetores linearmente independentes entre os do conjunto $\{v_1 - v_0, \ldots, v_k - v_0\}$.*

Definição 2.11. (*Célula Convexa Afim*) *Chama-se de célula convexa afim, gerada pelos pontos* $v_0, v_1, \ldots, v_k$*, o conjunto* $\sigma = [v_0, \ldots, v_k] = \{v \in \mathbb{R}^n \mid \sum_{i=0}^{k} \lambda_i v_i = v, \sum_{i=0}^{k} \lambda_i = 1 \text{ e } \lambda_i \geq 0\}$*. Definimos a dimensão de* σ *por* $dim(\sigma) = dim(aff(v_0, \ldots, v_n))$.

Definição 2.12. (*Decomposição Celular*) *Uma coleção* $\mathcal{C}$ *de células convexas afins é dita decomposição celular de um conjunto* $S \subset \mathbb{R}^n$ *se:*

- $S = \bigcup_{\sigma \in \mathcal{C}} \sigma$;
- *se* $\sigma_1, \sigma_2 \in \mathcal{C}$*, então* $\sigma_1 \cap \sigma_2 = \emptyset$ *ou* $\sigma_1 \cap \sigma_2 \in \mathcal{C}$;
- *todo subconjunto compacto de* S *intersecta um número finito de células de* $\mathcal{C}$.

Definição 2.13. (*Simplexo*) *Uma célula convexa afim* $\sigma = [v_0, \ldots, v_k]$ *de* $\mathbb{R}^n$ *é chamada de simplexo, se* $dim(\sigma) = k$*, isto é, um simplexo* k*-dimensional é uma célula convexa afim de dimensão* k *gerada por* $k+1$ *pontos.*

Definição 2.14. (*Faces*) *Dado um simplexo* $\sigma = [v_0, \ldots, v_k]$ *de* $\mathbb{R}^n$ *e* $\{\omega_0, \ldots, \omega_l\} \subset \{v_0, \ldots, v_k\}$*, chama-se a célula convexa afim* $\tau = [\omega_0, \ldots, \omega_l]$ *de face de dimensão* l *de* σ*. Às faces de* σ *geradas por um único ponto denomina-se vértices de* σ*; por dois pontos, de arestas de* σ*; e por* k *pontos, de facetas de* σ.

Definição 2.15. (*Triangulação*) *Uma decomposição celular* $\mathcal{C}$ *de* $S \subset \mathbb{R}^n$ *é chamada de triangulação de* S*, se todas as células de* $\mathcal{C}$ *são simplexos.*

Definição 2.16. (*Propriedades de Simplexos*) *Dado um simplexo* $\sigma = [v_0, \ldots, v_k]$ *de* $\mathbb{R}^n$*, define-se :*

- *a fronteira de* σ*,* $\partial\sigma$*, como a união de todas as faces de dimensão* $k-1$ *de* σ;
- *o baricentro de* σ*,* $b(\sigma) = \frac{1}{k+1} \sum_{i=0}^{k} v_i$;
- *o diâmetro de* σ*,* $\rho(\sigma) = max\{\| v_i - v_j \|; i, j = 0, \ldots, k\}$;
- *o raio de* σ*,* $r(\sigma) = min\{\| v - b(\sigma) \| \mid v \in \partial(\sigma)\}$;
- *a robustez de* σ*,* $\theta(\sigma) = r(\sigma)/\rho(\sigma)$.

Definição 2.17. (*Propriedades de Triangulação*) *Se* T *é uma triangulação de* $S \subset \mathbb{R}^n$*, define-se:*

- *o diâmetro de* T*,* $\rho(T) = sup\{\rho(\sigma) \mid \sigma \in T \text{ com } dim(\sigma) > 0\}$;
- *a robustez de* T*,* $\theta(T) = inf\{\theta(\sigma) \mid \sigma \in T \text{ com } dim(\sigma) > 0\}$.

2.3 Interpolação Linear Simplicial

Definição 2.18. (*Aproximação Simplicial*) *Seja* $F : S \subset \mathbb{R}^n \to \mathbb{R}^k$ *uma aplicação e* T *uma triangulação de* S*. Se* $\sigma = [v_0, \ldots, v_n] \in T$*, tem-se para cada* $v \in \sigma$ *uma única* $(n+1)$*-upla* $\lambda = (\lambda_0, \ldots, \lambda_n)$ *tal que* $\sum_{i=0}^{n} \lambda_i v_i = v$*,* $\sum_{i=0}^{n} \lambda_i = 1$ *e* $\lambda_i \geq 0$*,* $i = 0, \ldots, n$*. Define-se* $F_\sigma : \sigma \to \mathbb{R}^k$ *onde* $F_\sigma(v) = F_\sigma(\sum_{i=0}^{n} \lambda_i v_i) = \sum_{i=0}^{n} \lambda_i F(v_i)$.

Observe que F_σ é uma aplicação afim e que $F_\sigma(v_i) = F(v_i)$, $i = 0, \ldots, n$, isto é, F_σ é uma interpolação linear de F nos vértices de σ.

Definição 2.19. (*Aproximação Linear por Partes*) *Define-se uma aproximação linear por partes para F sendo, $F_T : S \subset \mathbb{R}^n \to \mathbb{R}^k$ onde $F_T(v) = F_\sigma(v)$ para $v \in \sigma \in T$.*

Portanto, F_T é uma interpolação de F para os vértices de T (nos simplexos de dimensão zero de T) e que é afim em cada simplexo de dimensão n de T.

Teorema 2.4. *Sejam $S \subset \mathbb{R}^n$ aberto, $F : S \to \mathbb{R}^k$ uma aplicação de classe C^2 com $\| D^2F(x) \| \leq \alpha$ para todo $x \in S$ e $\sigma = [v_0, \ldots, v_n]$ um simplexo de uma triangulação robusta T de S ($\theta(T) > 0$), então*

1. $\| F(v) - F_T(v) \| \leq \alpha\rho^2(T)/2$ *para todo $v \in S$ e*

2. $\| DF(v) - DF_T(v) \| \leq \alpha\rho(T)/\theta(T)$ *para todo v no interior de simplexos de dimensão n de T.*

Embora $DF_\sigma(v)$ esteja bem definida para todo simplexo σ de dimensão n, se τ é uma face comum de dois simplexos σ_1 e σ_2 de dimensão n e $v \in \tau, DF_{\sigma_1}(v)$ e $DF_{\sigma_2}(v)$ podem ser distintas e, portanto, $DF_T(v)$ não está bem definida.

2.4 Esquema de Diferenças Simplicial

Esta seção e a próxima são importante para mostrar o funcionamento das aproximações de variedades implícitas e suas limitações.

Seja $F : S \subset \mathbb{R}^n \to \mathbb{R}^k$ uma aplicação de classe C^2 e T uma triangulação de S.

Seja $\sigma = [v_0, \ldots, v_n] \in T$. Como σ é um simplexo de dimensão n, os vetores $\{v_1 - v_0, v_2 - v_0, \ldots, v_n - v_0\}$ são linearmente independentes; logo, fazendo $\delta_i = \| v_i - v_0 \|$ e $\omega_i = (v_i - v_0)/\delta_i$, $i = 1, \ldots, n$, temos que a matriz cujas colunas são os vetores $\{\omega_1, \ldots, \omega_n\}$ é invertível.

Para $v \in \sigma$, seja λ, tal que $\sum_{i=0}^n \lambda_i v_i = v$, $\sum_{i=0}^n \lambda_i = 1$ e $\lambda_i \geq 0$, $i = 0, \ldots, n$. Então, pode-se escrever

$$\begin{aligned}
&v - v_0 = \\
&\textstyle\sum_{i=0}^n \lambda_i v_i - v_0 \sum_{i=0}^n \lambda_i = \\
&\textstyle\sum_{i=1}^n \lambda_i (v_i - v_0) = \\
&\textstyle\sum_{i=1}^n \lambda_i \delta_i \omega_i = \\
&(\omega_1 \cdots \omega_n) \begin{pmatrix} \lambda_1\delta_1 \\ \vdots \\ \lambda_n\delta_n \end{pmatrix}
\end{aligned}$$

ou

$$\begin{pmatrix} \lambda_1\delta_1 \\ \vdots \\ \lambda_n\delta_n \end{pmatrix} = (\omega_1 \cdots \omega_n)^{-1} (v - v_0) .$$

Agora

$$\begin{aligned}
&F_T(v) - F(v_0) = \\
&F_\sigma(v) - F(v_0) = \\
&\textstyle\sum_{i=0}^n \lambda_i F(v_i) - F(v_0) \sum_{i=0}^n \lambda_i = \\
&\textstyle\sum_{i=1}^n \lambda_i (F(v_i) - F(v_0)) = \\
&\textstyle\sum_{i=1}^n \lambda_i \delta_i (F(v_i) - F(v_0))/\delta_i
\end{aligned}$$

ou

$$F_T(v) - F(v_0) =$$

$$\begin{pmatrix} \frac{F(v_1)-F(v_0)}{\delta_1} & \cdots & \frac{F(v_n)-F(v_0)}{\delta_n} \end{pmatrix} \begin{pmatrix} \lambda_1\delta_1 \\ \vdots \\ \lambda_n\delta_n \end{pmatrix} =$$

$$\begin{pmatrix} \frac{F(v_1)-F(v_0)}{\delta_1} & \cdots & \frac{F(v_n)-F(v_0)}{\delta_n} \end{pmatrix} (\omega_1 \cdots \omega_n)^{-1} (v - v_0) .$$

Daí, como $\omega_i = (v_i - v_0)/\delta_i$, tem-se $v_i = v_0 + \delta_i\omega_i$, $i = 1, \ldots, n$, e portanto

$$F_T(v) - F(v_0) = \left(\frac{F(v_0 + \delta_1\omega_1) - F(v_0)}{\delta_1} \cdots \frac{F(v_0 + \delta_n\omega_n) - F(v_0)}{\delta_n} \right) (\omega_1 \cdots \omega_n)^{-1} (v - v_0) .$$

Observe que as colunas da matriz

$$\left(\frac{F(v_0 + \delta_1\omega_1) - F(v_0)}{\delta_1} \cdots \frac{F(v_0 + \delta_n\omega_n) - F(v_0)}{\delta_n} \right)$$

são aproximações para as derivadas direcionais de F nas direções $\omega_1, \ldots, \omega_n$, em torno do ponto v_0; portanto este é um esquema de diferenças que chamaremos de esquema de diferenças simplicial.

Assim, se $\delta_i \to 0$ para $i = 1, \ldots, n$ com uma certa proporcionalidade, isto é, se o diâmetro de σ tende a zero mantendo a robustez limitada para que os vetores $\omega_1, \ldots, \omega_n$ continuem sendo linearmente independentes, então F_σ é uma aproximação para os dois primeiros termos da série de Taylor de F, em torno do ponto v_0. Outra observação é que a mesma análise pode ser feita para qualquer outro vértice de σ; portanto, o esquema descrito anteriormente não depende apenas do ponto v_0 e sim do simplexo σ.

2.5 Resultados sobre Aproximações Simpliciais

Seja $F : U \to \mathbb{R}^k$ uma aplicação de classe C^p no aberto $U \subset \mathbb{R}^n$, com $n > k$, $p > max\{1, n/k - 1\}$ e tendo $\mathbf{0} \in \mathbb{R}^k$ como valor regular. Considere a variedade de dimensão $n-k$ e classe C^p, $\mathcal{M} = F^{-1}(\mathbf{0})$ e uma vizinhança tubular de $\mathcal{M}$, $V_\mathcal{M}$, tal como no Teorema 2.2.

Como desejamos uma aproximação de $\mathcal{M}$ que possa ser representada computacionalmente, vamos restringir o domínio de F a um compacto $S \subset U$. Assim, existe $\alpha > 0$ tal que $\| D^2F(x) \| \leq \alpha$ para todo $x \in K$ e, portanto, se T for uma triangulação robusta de S, fica valendo o Teorema 2.4 para F_T.

Para estabelecermos algumas propriedades sobre a aproximação $\mathcal{M}_T = F_T^{-1}(\mathbf{0})$ de $\mathcal{M}$, vamos observar algumas propriedades que F e T devem satisfazer.

Seja $\sigma = [v_0, \ldots, v_k]$ um simplexo de dimensão k de U. Para que $F_\sigma(\sigma) = [F(v_0), \ldots, F(v_k)]$ seja um simplexo de dimensão k em $\mathbb{R}^k$, basta que os de vetores $F(v_1) - F(v_0), \ldots, F(v_k) - F(v_0)$ sejam linearmente independentes, pelo fato de F_σ ser uma aplicação afim. Neste caso, se τ é uma face de σ de dimensão $r \leq k$, teremos que $F_\sigma(\tau)$ é uma face de $F_\sigma(\sigma)$ de dimensão r.

Com relação a este assunto temos o seguinte teorema:

Teorema 2.5. (Veja [20]) *Para quase todo simplexo σ de dimensão $r \leq k$ de $S \cap V_\mathcal{M}$, temos que $F_\sigma(\sigma)$ é um simplexo de dimensão r em $\mathbb{R}^k$.*

Definição 2.20. *Diremos que $v \in \sigma = [v_0, \ldots, v_n] \in T$ é um ponto regular de F_T, se $DF_\sigma(v)$ for sobrejetora, ou de forma equivalente, que a matriz*

$$\begin{pmatrix} 1 & \cdots & 1 \\ F(v_0) & \cdots & F(v_n) \end{pmatrix}$$

tenha posto $k+1$. Se v não for um ponto regular de F_T, diremos que é um ponto crítico de F_T. Diremos que $c \in \mathbb{R}^k$ é valor regular de F_T se todo $v \in F_T^{-1}(c)$ for ponto regular de F_T. Se c não for valor regular de F_T, diremos que é valor crítico de F_T.

Teorema 2.6. (Veja [18]) *Seja T uma triangulação de S. Se $0 \in \mathbb{R}^n$ for valor regular de F_T, então $\mathcal{M}_T = F_T^{-1}(0)$ é uma variedade linear por partes de dimensão $n-k$.*

Observe-se que o teorema 2.6 não garante que $F_T^{-1}(0) \cap \sigma$ seja uma célula de dimensão r se σ for um simplexo de dimensão $k+r$, $r = 1, \ldots, n-k$. Com relação a esta observação, temos o seguinte exemplo: $F : \mathbb{R}^2 \to \mathbb{R}$ definida por $F(x,y) = x + 2y$ e $\sigma = [v_0, v_1, v_2]$ com $v_0 = (0,0), v_1 = (1,0)$ e $v_2 = (0,1)$. Como $F(v_0) = 0, F(v_1) = 1$ e $F(v_2) = 2$, temos que $F_\sigma^{-1}(0) \cap \sigma = \{(0,0)\}$, que é uma célula de dimensão 0.

Como $DF(x)$ tem posto k para todo $x \in S \cap V_\mathcal{M}$, a inversa de Moore-Penrose para $DF(x)$, $DF(x)^+ = DF(x)^t \cdot (DF(x) \cdot DF(x)^t)^{-1}$ está bem definida. Como $S \cap V_\mathcal{M}$ é um conjunto compacto, existe $\mu > 0$, tal que $\| DF(x)^+ \| \leq \mu$ para todo $x \in S \cap V_\mathcal{M}$.

Teorema 2.7. (Veja [20]) *Para quase todo simplexo σ de dimensão k de $S \cap V_\mathcal{M}$ com $\alpha\mu\rho(\sigma)/\theta(\sigma) < 1$, cujo interior intercepta $\mathcal{M}$, temos que o interior de σ intercepta $\mathcal{M}_\sigma$.*

Como S é compacto, para quase toda triangulação T de S, seus simplexos de dimensão maior ou igual a k têm interior transversal a $\mathcal{M}$. Daí, segue o corolário do teorema 2.7.

Corolário 2.1. (Veja [20]) *Para quase toda triangulação robusta T de S com diâmetro suficientemente pequeno, temos que todo simplexo de dimensão maior ou igual a k tem interior transversal a $\mathcal{M}$ e $\mathcal{M}_T$, e se intercepta $\mathcal{M}$, então também intercepta $\mathcal{M}_T$.*

O corolário 2.1 nos dá duas propriedades muito importantes: para quase toda triangulação T de S com diâmetro suficientemente pequeno, temos

- se σ for um simplexo de dimensão k de T e $x \in \sigma \cap \mathcal{M}$, existe $v \in \sigma \cap \mathcal{M}_T$ e consequentemente $d(x,v) = \| x - v \| \leq \rho(\sigma) \leq \rho(T)$;
- se σ for um simplexo de dimensão $k+r$ de T com $0 \leq r \leq n-k$, temos que se $F_T^{-1}(0) \cap \sigma \neq \emptyset$, então $F_T^{-1}(0) \cap \sigma$ é uma célula convexa afim de dimensão r.

A segunda propriedade será muito importante para representação computacional das células de $\mathcal{M}_T$. Também com relação à aproximação entre $\mathcal{M}$ e $\mathcal{M}_T$, temos os seguintes teoremas:

Teorema 2.8. (Veja [22]) *Dado $\epsilon > 0$, existe uma triangulação T de S tal que, se $v \in \mathcal{M}_T = F_T^{-1}(0)$, então $d(v, \mathcal{M}) = inf\{\| u - v \|; u \in \mathcal{M}\} < \epsilon$.*

Teorema 2.9. (Veja [2]) *Sejam $x \in \mathcal{M}$ e $y \in \mathcal{M}_T$. Se $\| x - y \| \leq 1/(\alpha\mu)$, então $\| x - y \| \leq \alpha\mu\rho^2(T)$.*

2.6 As Triangulações K_1 e J_1

Inicialmente será apresentada a triangulação $\mathcal{CFK}$ (devida a Coxeter [13], Freudenthal [21] e Kuhn [33]) do cubo unitário $I^n = [0,1]^n$, da qual derivam as triangulações K_1 e J_1 de $\mathbb{R}^n$.

Sejam $\pi : \{1, 2, \dots, n\} \to \{1, 2, \dots, n\}$ uma permutação de $\{1, 2, \dots, n\}$ a qual vamos denotar pela n-upla ordenada $\pi = (\pi_1, \pi_2, \dots, \pi_n)$, e $\{e_1, e_2, \dots, e_n\}$ base canônica de $\mathbb{R}^n$.

Considere o simplexo $\sigma_\pi = [v_0, v_1, \dots, v_n]$, onde

$$\begin{aligned} v_0 &= (0, \dots, 0) \in I^n; \\ v_i &= v_{i-1} + e_{\pi_i} = e_{\pi_1} + \cdots + e_{\pi_i}, \quad i = 1, \dots, n. \end{aligned}$$

É fácil verificar que σ_π é um simplexo de dimensão n contido em I^n.

A triangulação $\mathcal{CFK}$ é definida como o conjunto dos simplexos σ_π e suas faces, tal que π seja uma permutação de $\{1, 2, \dots, n\}$.

Para exemplificar, a Figura 2.1 representa as triangulações $\mathcal{CFK}$ de I^2 e I^3, respectivamente.

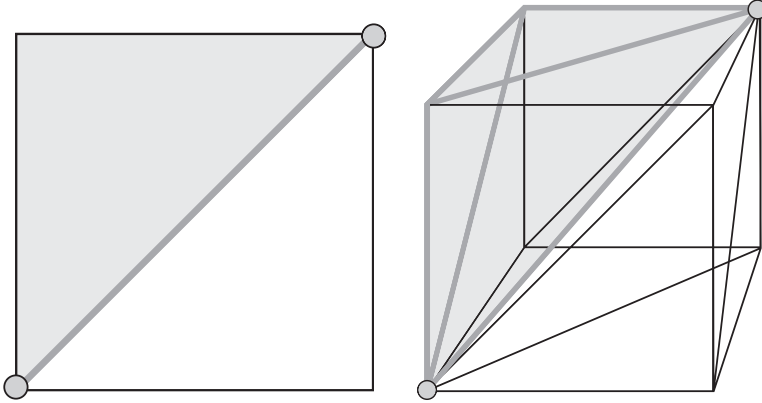

Figura 2.1: À esquerda, a triangulação $\mathcal{CFK}$ de I^2 e à direita, a triangulação $\mathcal{CFK}$ de I^3.

Observe que existe uma relação biunívoca entre π e σ_π; portanto, a triangulação $\mathcal{CFK}$ de I^n contém $n!$ simplexos de dimensão n.

A triangulação K_1 de $\mathbb{R}^n$ é obtida pela translação da triangulação $\mathcal{CFK}$ de I^n.

Para cada $v_0 \in \mathbb{Z}^n$ e π permutação de $\{1, 2, \dots, n\}$, definimos $\sigma = K_1(v_0, \pi) = [v_0, v_1, \dots, v_n]$, onde

$$v_i = v_{i-1} + e_{\pi_i} = v_0 + e_{\pi_1} + \cdots + e_{\pi_i}, \quad i = 1, \dots, n.$$

A triangulação K_1 de $\mathbb{R}^n$ é formada pelos simplexos $K_1(v_0, \pi)$ e suas faces.

A triangulação J_1 de $\mathbb{R}^n$ é obtida pela reflexção da triangulação $\mathcal{CFK}$ de I^n, com relação às faces de dimensão $n-1$ de I^n, isto é, para cada $v_0 \in \{(x_1, x_2, \dots, x_n) \in \mathbb{Z}^n \mid x_i \text{ é par}\}$, $s = (s_1, \dots, s_n) \in \{-1, 1\}^n$ e π permutação de $\{1, 2, \dots, n\}$, definimos $\sigma = J_1(v_0, \pi, s) = [v_0, v_1, \dots, v_n]$, onde

$$v_i = v_{i-1} + s_{\pi_i} e_{\pi_i} = v_0 + s_{\pi_1} e_{\pi_1} + \cdots + s_{\pi_i} e_{\pi_i}, \quad i = 1, \dots, n.$$

A triangulação J_1 de $\mathbb{R}^n$ é formada pelos simplexos $J_1(v_0, \pi, s)$ e suas faces.
A figura 2.2 apresenta as triangulações K_1 e J_1 de $\mathbb{R}^2$, respectivamente.

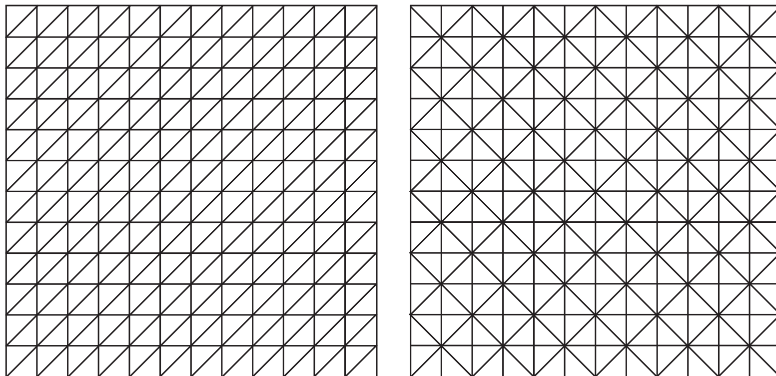

Figura 2.2: À esquerda, a triangulação K_1 de $\mathbb{R}^2$ e à direita, a triangulação J_1 de $\mathbb{R}^2$.

Dados dois vetores v e δ em $\mathbb{R}^n$ com $\delta_i > 0$, $i = 1, \ldots, n$, pode-se obter as triangulações $v + \delta K_1$ e $v + \delta J_1$, trocando os vetores da base canônica $e_1, \ldots, e_n$ pelos vetores $\delta_1 e_1, \ldots, \delta_n e_n$ nas triangulações K_1 e J_1, e somando a seus vértices o vetor v. Estas triangulações são as triangulações K_1 e J_1 escalonadas e transladadas.

Note-se que $\rho(K_1) = \rho(J_1) = \sqrt{n}$ e $\rho(v + \delta K_1) = \rho(v + \delta J_1) = \sqrt{\delta_1^2 + \cdots + \delta_n^2} = \| \delta \|_2$.

2.7 Exercícios

1. Suponha que $F : S \subset \mathbb{R}^n \to \mathbb{R}^k$ seja uma aplicação de classe C^2 com $\| D^2F(x) \| \leq \alpha$ para todo $x \in S$ e seja T uma triangulação robusta de S ($\theta(T) > 0$). Mostre que $\| F(v) - F_T(v) \| \leq \alpha\rho^2(T)/2$ para todo $v \in S$.

2. Suponha que $F : S \subset \mathbb{R}^n \to \mathbb{R}^k$ seja uma aplicação de classe C^2 com $\| D^2F(x) \| \leq \alpha$ para todo $x \in S$ e seja T uma triangulação robusta de S ($\theta(T) > 0$). Mostre que $\| DF(v) - DF_T(v) \| \leq \alpha\rho(T)/\theta(T)$ para todo v no interior de simplexos de dimensão n de T.

3. Refaça os cálculos do esquema de diferenças simplicial da seção 2.4 em torno do vértice v_1.

4. Combine translações e reflexões da triangulação $\mathcal{CFK}$ de I^n para gerar novas triangulações de $\mathbb{R}^n$.

Capítulo 3

Contagem e Enumeração

Este capítulo mostra técnicas de contagem e enumeração usadas para representar malhas cartesianas e triangulações pelos algoritmos de aproximação de variedades apresentados neste livro. Para maior aprofundamento nos temas de contagem e enumeração, sugerimos textos clássicos sobre algoritmos combinatórios, tais como [44], [19] e [40]. Os códigos em Matlab/Octave que são apresentados neste capítulo são, em geral, sub-rotinas dos algoritmos Marching Simplex e Continuation Simplex.

3.1 Produtos Cartesianos Discretos

3.1.1 Definição

Dado um conjunto de n conjuntos discretos, $I_1^{k_1}, I_2^{k_2}, \dots, I_n^{k_n}$, onde k_i é o número de elementos do conjunto $I_i^{k_i}$, define-se o produto cartesiano $I = I_1^{k_1} \times I_2^{k_2} \times \cdots \times I_n^{k_n}$ como o conjunto

$$I = \{(i_1, i_2, \dots, i_n) \quad | \quad i_1 \in I_1^{k_1}, i_2 \in I_2^{k_2}, \dots, i_n \in I_n^{k_n}\}$$

Como exemplo, considere os conjuntos $I_1^2 = \{0,1\}$ e $I_2^3 = \{a,b,c\}$, então pela definição acima tem-se

$$I = I_1^2 \times I_2^3 = \{(0,a),(0,b),(0,c),(1,a),(1,b),(1,c)\}$$

A contagem (número de elementos neste caso) do conjunto $I = I_1^{k_1} \times I_2^{k_2} \times \cdots \times I_n^{k_n}$ é bastante simples e é dada por $k_1 k_2 \dots k_n$.

A enumeração (quais são os elementos apresentados em alguma ordem) já pode não ser uma tarefa tão simples. Veremos a seguir alguns casos de enumeração de produto cartesiano importantes para este livro.

3.1.2 Exemplos

Considere $I = I_0^{k_0} \times I_1^{k_1} \times \cdots \times I_n^{k_n}$, onde $I_i^{k_i} = \{0,1\}$. Como os elementos de cada conjunto $I_i^{k_i}$ são 0 e 1, o conjunto I pode ser olhado como a representação na base 2 dos números 0 a $2^{n+1} - 1$. Neste caso, podemos enumerar o conjunto com a

seguinte ordem:

$$\begin{array}{rll} 0 = & 0 \cdot 2^0 + 0 \cdot 2^1 + \cdots + 0 \cdot 2^n & \text{correspondente a} \quad (0,0,\ldots,0) \\ 1 = & 1 \cdot 2^0 + 0 \cdot 2^1 + \cdots + 0 \cdot 2^n & \text{correspondente a} \quad (1,0,\ldots,0) \\ 2 = & 0 \cdot 2^0 + 1 \cdot 2^1 + \cdots + 0 \cdot 2^n & \text{correspondente a} \quad (0,1,\ldots,0) \\ \vdots & & \\ 2^{n+1}-1 = & 1 \cdot 2^0 + 1 \cdot 2^1 + \cdots + 1 \cdot 2^n & \text{correspondente a} \quad (1,1,\ldots,1) \end{array}$$

Um raciocínio análogo pode ser feito para a representação de números em qualquer base como produto cartesiano com uma enumeração apropriada. Observem que é possível obter a ordem de uma enumeração de um produto cartesiano a partir de cada elemento, tal como a regra do exemplo acima, e também é possível obter o elemento a partir da ordem deste elemento na enumeração utilizando um algoritmo bem simples de divisão por 2 (que neste caso é a base).

Se y é um número entre 0 e $2^{n+1}-1$, para obter o dígito que multiplica 2^0 que vamos denominar $x(1)$, basta encontrar o resto da divisão de y por 2. Para encontrar o dígito que multiplica 2^1 que vamos denominar $x(2)$, basta encontrar o resto da divisão de $(y - x(1))/2$ por 2, e assim sucessivamente.

Uma aplicação importante para este livro envolvendo a enumeração de produtos cartesianos é a possibilidade de enumerar as células de uma malha cartesiana e também obter uma célula a partir da ordem da enumeração desta. Por exemplo, uma malha cartesiana no plano pode ser rotulada da seguinte forma:

5	(0,5)	(1,5)	(2,5)	(3,5)	(4,5)	(5,5)	(6,5)
4	(0,4)	(1,4)	(2,4)	(3,4)	(4,4)	(5,4)	(6,4)
3	(0,3)	(1,3)	(2,3)	(3,3)	(4,3)	(5,3)	(6,3)
2	(0,2)	(1,2)	(2,2)	(3,2)	(4,2)	(5,2)	(6,2)
1	(0,1)	(1,1)	(2,1)	(3,1)	(4,1)	(5,1)	(6,1)
0	(0,0)	(1,0)	(2,0)	(3,0)	(4,0)	(5,0)	(6,0)
	0	1	2	3	4	5	6

Se uma enumeração for feita por linha, a ordem da enumeração fica:

5	(0,5) 35	(1,5) 36	(2,5) 37	(3,5) 38	(4,5) 39	(5,5) 40	(6,5) 41
4	(0,4) 28	(1,4) 29	(2,4) 30	(3,4) 31	(4,4) 32	(5,4) 33	(6,4) 34
3	(0,3) 21	(1,3) 22	(2,3) 23	(3,3) 24	(4,3) 25	(5,3) 26	(6,3) 27
2	(0,2) 14	(1,2) 15	(2,2) 16	(3,2) 17	(4,2) 18	(5,2) 19	(6,2) 20
1	(0,1) 7	(1,1) 8	(2,1) 9	(3,1) 10	(4,1) 11	(5,1) 12	(6,1) 13
0	(0,0) 0	(1,0) 1	(2,0) 2	(3,0) 3	(4,0) 4	(5,0) 5	(6,0) 6
	0	1	2	3	4	5	6

Assim se $(x(1), x(2))$ é um elemento da malha, a enumeração fica $y = x(1) + 7 \cdot x(2)$ e por exemplo $(4,3)$ é o elemento número $4 + 7 \cdot 3 = 25$. Para obter $(x(1), x(2))$ a partir de y, $x(1)$ é o resto da divisão de y por 7 e $x(2) = (y - x(1))/7$. No exemplo anterior se $y = 25$, $x(1) = mod(y,7) = mod(25,7) = 4$ e $x(2) = (y - x(1))/7 = 3$.

3.1.3 Programas em Matlab/Octave

Se y é um inteiro entre 0 e $2^n - 1$, então a n-upla $(x_1, x_2, \ldots, x_n)$ pertencente a I que corresponde a representação de y na base 2 pode ser obtida com o seguinte algoritmo em Matlab/Octave:

```
copy = y;
for j = 1:n-1
     x(j)   = mod(copy,2);
     copy   = (copy-x(j))/2;
end
x(n) = copy;
```

Por exemplo, se $n = 4$ e $y = 13$, temos:

$$\begin{array}{lll} copy & = 13 & \\ x(1) & = mod(13,2) & = 1 \\ copy & = (13-1)/2 & = 6 \\ x(2) & = mod(6,2) & = 0 \\ copy & = (6-0)/2 & = 3 \\ x(3) & = mod(3,2) & = 1 \\ copy & = (3-1)/2 & = 1 \\ x(4) & = 1 & \end{array}$$

e

$$y = x(1) \cdot 2^0 + x(2) \cdot 2^1 + x(3) \cdot 2^2 + x(4) \cdot 2^3 = 1 \cdot 1 + 0 \cdot 2 + 1 \cdot 4 + 1 \cdot 8 = 13$$

Observe que os vértices do hipercubo $I^n = [0,1]^n$ podem ser obtidos por este mesmo algoritmo.

```
for i = 0:2^n-1
      [Coords]  = HyperCubeCoords(n,i);
end

function [Coords] = HyperCubeCoords(n,i)
   copy = i;
   for j = 1:n-1
      Coords(j) = mod(copy,2);
      copy      = (copy-Coords(j))/2;
   end
   Coords(n) = copy;
   return
end
```

Se $I = I_1^{k_1} \times I_2^{k_2} \times \cdots \times I_n^{k_n}$ com $I_j^{k_j} = \{1, \ldots, k_j\}$, definimos $Base(1) = 1$ e $Base(j+1) = k_{j+1} \cdot Base(j)$, $j = 1, \ldots, n-1$. Assim se $(x(1), x(2), \ldots, x(n))$ é um elemento de I, a enumeração equivalente ao exemplo anterior é dado por:

$$y = \sum_{j=1}^{n} x(j) \cdot Base(j)$$

O processo de recuperação do elemento de uma malha cartesiana com coordenadas inteiras a partir de seu rótulo y é obtida com o seguinte algoritmo em Matlab/Octave:

```
copy = y;
for j = n:-1:2
     aux    = mod(copy,Base(j));
     x(j)   = (copy-aux)/Base(j);
     copy   = aux;
end
x(1) = copy;
```

No exemplo anterior, $Base(1) = 1$, Base(2) $= 7$ e o elemento número 18 da malha é dado por:

$$\begin{array}{lll} copy & = 18 & \\ aux & = mod(18,7) & = 4 \\ x(2) & = (18-4)/7 & = 2 \\ copy & = 4 & \\ x(1) & = 4 & \end{array}$$

e

$$y = x(1) \cdot Base(1) + x(2) \cdot Base(2) = 4 \cdot 1 + 2 \cdot 7 = 18$$

A seguir, temos um trecho de código Matlab/Octave para gerar uma malha cartesiana com o número de células em cada coordenada dado pelo vetor $Division = [k_1, k_2, \ldots, k_n]$. A função $InitGrid$ define os valores de $Base(i)$ a partir de $Division$, para $1 \leq i \leq n+1$. A função $GridCoords$ encontra as coordenadas inteiras de um elemento da malha cartesiana a partir de sua enumeração. A Base calculada na função $InitGrid$ é usada como parâmetro da função $GridCoords$.

```
    Base  = InitGrid(n, Division);
    for g = 0:Base(n+1)-1
       [Grid] = GridCoords(n,g,Base) ;
    end

function [Base] = InitGrid(n, Division)
   Base(1) = 1;
   for i = 2:n+1
      Base(i) = Base(i-1)*Division(i-1);
   end
   return
end

function [Grid] = GridCoords(n,i,Base)
   copy = i;
   for j = n:-1:2
      aux     = mod(copy,Base(j));
      Grid(j) = (copy-aux)/Base(j);
      copy    = aux;
   end
   Grid(1) = copy;
   return
end
```

Para gerar as coordenadas dos vértices dos hipercubos de uma malha cartesiana, usamos os vetores $First$, correspondendo ao canto com as menores coordenadas da malha, $Last$, correspondendo ao canto com as maiores coodenadas da malha, e

Division, correspondendo ao número de divisões da malha em cada direção. Com eles obtemos o vetor *Delta*, que equivale aos comprimentos dos lados dos hipercubos. Todo hipercubo tem o mesmo tamanho definido por *Delta*.

Pode-se agrupar os trechos de código anteriores como segue. A nova função *GenVert* é chamada para cada elemento g da malha cartesiana e retorna as coordenadas dos vértices do hipercubo correspondente a g. Por sua vez, para cada vértice u do hipercubo, a função *GenVert* chama duas outras: *HyperCubeCoords*, que calcula as coordenadas de u em um hipercubo unitário e *HyperCube*, que translada e escalona as coordenadas de u do hipercubo unitário para sua posição final (*Vert*) na malha.

```
   Delta = (Last-First)./Division;
   Base  = InitGrid(n, Division);
   for g = 0:Base(n+1)-1
      [Grid] = GridCoords(n, g, Base)
      [Vert] = GenVert(n, Grid, First, Delta)
   end

function [Base] = InitGrid(n, Division)
   Base(1) = 1;
   for i = 2:n+1
      Base(i) = Base(i-1)*Division(i-1);
   end
   return
end

function [Grid] = GridCoords(n,i,Base)
   copy = i;
   for j = n:-1:2
      aux     = mod(copy,Base(j));
      Grid(j) = (copy-aux)/Base(j);
      copy    = aux;
   end
   Grid(1) = copy;
   return
end

function [Vert] = GenVert(n, Grid, First, Delta)
   Vert  = [];
   for i = 0:2^n-1
      [Coords]  = HyperCubeCoords(n,i);
      [VHC]     = HyperCube(n,First,Delta,Grid,Coords);
      Vert      = [Vert; VHC];
   end
   return
end

function [Coords] = HyperCubeCoords(n,i)
   copy = i;
   for j = 1:n-1
      Coords(j) = mod(copy,2);
```

```
      copy      = (copy-Coords(j))/2;
   end
   Coords(n) = copy;
   return
end

function [VHC] = HyperCube(n, First, Delta, I, Coords)
   for i = 1:n
      VHC(i) = First(i) + (I(i)+Coords(i))*Delta(i);
   end
   return
end
```

3.2 Permutações

3.2.1 Definição

Seja $\pi : \{1, 2, \dots, m\} \to \{1, 2, \dots, m\}$ uma permutação de $\{1, 2, \dots, m\}$ a qual pode-se denotar pela m-upla ordenada $\pi = (\pi_1, \pi_2, \dots, \pi_m)$.

Pretende-se enumerar o conjunto das permutações de $\{1, 2, \dots, m\}$ e para isto pode-se listar o conjunto das permutações de maneira indutiva, usando o fato que cada permutação de $\{1, 2, \dots, m\}$ pode ser obtida das permutações de $\{1, 2, \dots, m-1\}$, adicionando a coordenada m em todas as posições possíveis. Por exemplo se $(i_1, i_2, \dots, i_{m-1})$ é uma permutação de $\{1, 2, \dots, m-1\}$ pode-se gerar m permutações de $\{1, 2, \dots, m\}$ incluindo m como segue:

$$
\begin{array}{c}
(m, i_1, i_2, \dots, i_{m-1}) \\
(i_1, m, i_2, \dots, i_{m-1}) \\
(i_1, i_2, m, \dots, i_{m-1}) \\
\vdots \\
(i_1, i_2, \dots, m, i_{m-1}) \\
(i_1, i_2, \dots, i_{m-1}, m)
\end{array}
$$

Permutações serão usadas pelos algoritmos de aproximação de variedades deste livro para representar os simplexos de dimensão m de um hipercubo das triangulações J_1 ou K_1, vistas na Seção 2.6 do Capítulo 2.

3.2.2 Exemplos

A seguir mostra-se como são geradas as permutações de $\{1\}$, $\{1, 2\}$, $\{1, 2, 3\}$ e $\{1, 2, 3, 4\}$.

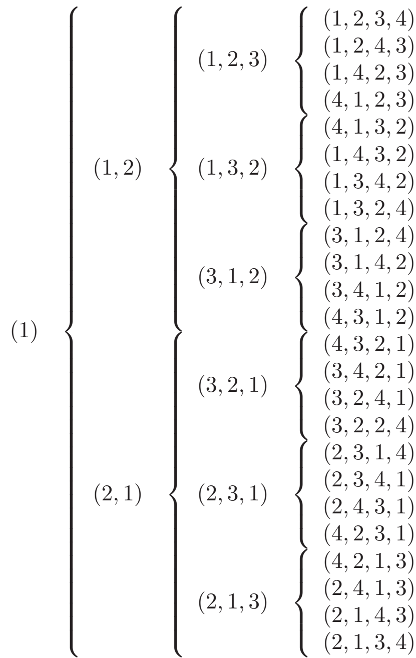

Figura 3.2.3

3.2.3 Programas em Matlab/Octave

Pode-se gerar as permutações de $\{1, 2, \ldots, m\}$ a partir do produto cartesiano. Primeiro observa-se que o conjunto $\{1\} \times \{1, 2\} \times \{1, 2, 3\} \times \cdots \times \{1, 2, \ldots, m\}$ tem exatamente $m!$ elementos e pode-se relacionar cada elemento deste conjunto $I_k = (i_1, i_2, \ldots, i_m)$ unicamente com uma permutação de $\{1, 2, \ldots, m\}$. A seguir vamos descrever um mapeamento destes dos conjuntos. Primeiro sabe-se que $i_1 = 1$, $i_2 = 1$ ou 2, $i_3 = 1$ ou 2 ou 3, assim por diante. A permutação é construída da seguinte forma:

- O dígito 1 é colocado na posição i_1^k, ou seja (1).
- O dígito 2 é colocado na posição i_2^k, isto é, se $i_2^k = 1$ tem-se $(2, 1)$ e caso $i_2^k = 2$ tem-se $(1, 2)$.
- O dígito 3 é colocado na posição i_3^k, ou seja se a permutação de $\{1, 2\}$ obtida foi $(2, 1)$ por exemplo e $i_3^k = 1$ tem-se $(3, 2, 1)$ e caso $i_3^k = 2$ tem-se $(2, 3, 1)$ e se $i_3^k = 3$ tem-se $(2, 1, 3)$.

Por exemplo, se $m = 4$ e $I_k = (i_1^k, i_2^k, i_3^k, i_4^k) = (1, 1, 3, 2)$ a permutação correspondente será gerada como segue:

$$\begin{array}{l} i_1^k = 1 \rightarrow (1) \\ i_2^k = 1 \rightarrow (2, 1) \\ i_3^k = 3 \rightarrow (2, 1, 3) \\ i_4^k = 2 \rightarrow (2, 4, 1, 3) \end{array}$$

Um algoritmo para obter uma permutação a partir de um elemento I_k é dado abaixo

```
function [F] = Map_Perm(m,f)
    for i = 1:m
        F(i) = 0;
    end
    for i = 1:m
        if F(f(i)) == 0
            F(f(i)) = i;
        else
            for j = m:-1:f(i)+1
                F(j) = F(j-1);
            end
            F(f(i)) = i;
        end
    end
    return
end
```

Para gerar todas as permutações de $\{1, 2, \ldots, m\}$ temos:

```
function [P] = Enumerate_Perm(m)
    P = [];
    for i = 1:m
        Division(i) = i;
    end
    [Base] = InitGrid(m,Division);
    for k = 0:Base(m+1)-1
        [f] = GridCoords(m,k,Base);
        f = f+1;
        [F] = Map_Perm(m,f);
        P   = [P; F];
    end
    return
end
```

3.3 Combinações

3.3.1 Definição

Combinações podem ser definidas como todos os subconjuntos de k elementos de um conjunto de n elementos. Se $C = \{1, 2, 3, \ldots, n\}$, as combinações de k elementos deste conjunto variando-se um único elemento são: $\{1, 2, \ldots, k-1, k\}$, $\{1, 2, \ldots, k-1, k+1\}$, ..., $\{1, 2, \ldots, k-1, n\}$, ... Para obter as outras combinações, varia-se os demais elementos. Para isso, pode ser usado o produto cartesiano, descartando-se repetições.

Se $C = \{1, 2, 3, \ldots, n\}$, para obter as combinações de k elementos de C, considere $I = I_1 \times I_2 \times \cdots \times I_k$, $I_i = \{1, \ldots, n-k+i\}, i = 1, \ldots, k$ e observe que o número de elementos de I é $(n-k+1)(n-k+2)\cdots n = k! \begin{pmatrix} n \\ k \end{pmatrix}$. Então, basta descartar

os elementos que não estão em ordem lexicográfica (ordem crescente), isto é, se $(i_1, i_2, \ldots, i_k)$ é um elemento do produto cartesiano, ele vai pertencer ao conjunto formado pelas combinações se $i_1 < i_2 < \cdots < i_k$.

Combinações serão usadas pelos algoritmos de aproximação de variedades deste livro para representar as faces de dimensão k de simplexos de dimensão n, na Seção 4.1 do Capítulo 4.

3.3.2 Exemplos

Por exemplo se $C = \{1, 2, 3, 4, 5\}$, todas as combinações com 3 elementos são: $\{1, 2, 3\}$, $\{1, 2, 4\}$, $\{1, 2, 5\}$, $\{1, 3, 4\}$, $\{1, 3, 5\}$, $\{1, 4, 5\}$, $\{2, 3, 4\}$, $\{2, 3, 5\}$, $\{2, 4, 5\}$ e $\{3, 4, 5\}$.

Para este exemplo, se $C = \{1, 2, 3, 4, 5\}$, para obter todas as combinações com 3 símbolos, enumera-se os elementos de $I = \{1, 2, 3\} \times \{1, 2, 3, 4\} \times \{1, 2, 3, 4, 5\}$ que são: $(1, 1, 1), (2, 1, 1)$, $(3, 1, 1)$, $(1, 2, 1), (2, 2, 1), (3, 2, 1)$, $(1, 3, 1), (2, 3, 1), (3, 3, 1)$, $(1, 4, 1), (2, 4, 1)$, $(3, 4, 1)$, $(1, 1, 2), (2, 1, 2), (3, 1, 2)$, $(1, 2, 2), (2, 2, 2), (3, 2, 2)$, $(1, 3, 2)$, $(2, 3, 2)$, $(3, 3, 2)$, $(1, 4, 2)$, $(2, 4, 2), (3, 4, 2)$, $(1, 1, 3), (2, 1, 3), (3, 1, 3)$, $(1, 2, 3), (2, 2, 3)$, $(3, 2, 3)$, $(1, 3, 3), (2, 3, 3)$, $(3, 3, 3)$, $(1, 4, 3)$, $(2, 4, 3)$, $(3, 4, 3)$, $(1, 1, 4), (2, 1, 4), (3, 1, 4)$, $(1, 2, 4), (2, 2, 4), (3, 2, 4)$, $(1, 3, 4), (2, 3, 4)$, $(3, 3, 4)$, $(1, 4, 4), (2, 4, 4), (3, 4, 4)$, $(1, 1, 5)$, $(2, 1, 5), (3, 1, 5)$, $(1, 2, 5), (2, 2, 5)$, $(3, 2, 5)$, $(1, 3, 5)$, $(2, 3, 5), (3, 3, 5)$, $(1, 4, 5)$, $(2, 4, 5)$, $(3, 4, 5)$.

Os elementos que estão em ordem lexicográfica são: $(1, 2, 3)$, $(1, 2, 4)$, $(1, 3, 4)$, $(2, 3, 4)$, $(1, 2, 5)$, $(1, 3, 5)$, $(2, 3, 5)$, $(1, 4, 5), (2, 4, 5), (3, 4, 5)$.

3.3.3 Programas em Matlab/Octave

Um algoritmo para gerar todas as combinações P de k elementos de um conjunto com elementos $\{1, 2, \ldots, n\}$ é apresentado a seguir. A função Lexico é responsável por eliminar combinações repetidas, aceitando apenas as que estão em ordem lexicográfica.

```
    P = [];
    for i = 1:k
        Division(i) = n-k+i;
    end
    [Base] = InitGrid(k,Division);
    for j = 0:Base(k+1)-1
        [f] = GridCoords(k,j,Base);
        f = f+1;
        [lex] = Lexico(k,f);
        if lex == 1
            P   = [P; f];
        end
    end

function [lex] = Lexico(k,f)
    for i = 1:k-1
        if f(i) >= f(i+1)
            lex = 0;
            return
        end
```

```
    end
    lex = 1;
    return
end
```

3.4 Exercícios

1. Gere um algoritmo para gerar os dígitos de um número na base 16.

2. Considere que você tem a enumeração dos elementos de $I = I_1 \times I_2$, pelo algoritmo dado na seção 3.1. Escreva um algoritmo que gere os elementos de $I \times I_3$ a partir da enumeração de I.

3. Gere um código em Matlab/Octave que leia um número n e dois vetores $First$ e $Last$ de dimensão n e simule o ninho de *loops* como o descrito abaixo

   ```
   for i_1 = First(1):Last(1)
      for i_2 = First(2):Last(2)
                  .
                  .
            for i_n = First(n):Last(n)
                  ...
            end
                  .
                  .
      end
   end
   ```

   utilizando enumeração de produto cartesiano.

4. Defina um mapeamento diferente do que foi definido na seção 3.2 entre o conjunto $\{1\} \times \{1,2\} \times \{1,2,3\} \times \cdots \times \{1,2,\ldots,m\}$ e o conjunto de todas as permutações de $\{1,2,\ldots,m\}$, e assim gere uma nova enumeração destas permutações.

5. Utilizando produto cartesiano discreto e permutações, escreva um código em Matlab/Octave que gere todos os arranjos de $\{1,2,\ldots,n\}$ com $k < n$ elementos.

6. Modifique a função

   ```
   function [lex] = Lexico(k,f)
   ```

   para gerar as combinações de $\{1,2,\ldots,n\}$ com k elementos em ordem decrescente.

Capítulo 4

Métodos Baseados em Simplexos

Este capítulo apresenta os algoritmos Marching Simplex (Seção 4.6) e Continuation Simplex (Seção 4.7) para aproximação de variedades implícitas. Os códigos completos e comentados desses algorítmos se encontram nos Apêncices A, B e C, respectivamente. Para que os algoritmos sejam implementados de forma eficiente, algumas estruturas de dados, descritas nas Seções 4.1-4.5 são usadas.

4.1 Representação de Simplexos e suas Faces

Observe que nosso objetivo aqui não é uma representação geométrica e sim algébrica utilizando rótulos.

Um simplexo de dimensão n em $\mathbb{R}^m$, $m \geq n$, gerado pelos vértices $\{v_1, v_2, \ldots, v_n, v_{n+1}\}$ e denotado por $\sigma = [v_1, v_2, \ldots, v_n, v_{n+1}]$, pode ser representado pelos índices dos vértices em ordem lexicográfica e denotado por $\ell(\sigma) = (1, 2, \ldots, n, n+1)$.

Como uma face de dimensão k, denotada por τ, de um simplexo de dimensão n, σ, também é um simplexo de dimensão k gerado pelo fecho convexo de $k+1$ vértices de σ, a representação de τ, $\ell(\tau) = (i_1, \ldots, i_k, i_{k+1})$ é uma combinação de $k+1$ rótulos de $\ell(\sigma) = (1, 2, \ldots, n, n+1)$ em ordem lexicográfica.

Exemplo: Se $\ell(\sigma) = (1, 2, 3, 4, 5)$ representa o simplexo $\sigma = [v_1, v_2, v_3, v_4, v_5]$, suas faces de dimensão 2 são:

1. $\ell(\tau_1) = (1, 2, 3)$ representando a face $\tau_1 = [v_1, v_2, v_3]$;
2. $\ell(\tau_2) = (1, 2, 4)$ representando a face $\tau_2 = [v_1, v_2, v_4]$;
3. $\ell(\tau_3) = (1, 2, 5)$ representando a face $\tau_3 = [v_1, v_2, v_5]$;
4. $\ell(\tau_4) = (1, 3, 4)$ representando a face $\tau_4 = [v_1, v_3, v_4]$;
5. $\ell(\tau_5) = (1, 3, 5)$ representando a face $\tau_5 = [v_1, v_3, v_5]$;
6. $\ell(\tau_6) = (1, 4, 5)$ representando a face $\tau_6 = [v_1, v_4, v_5]$;
7. $\ell(\tau_7) = (2, 3, 4)$ representando a face $\tau_7 = [v_2, v_3, v_4]$;
8. $\ell(\tau_8) = (2, 3, 5)$ representando a face $\tau_8 = [v_2, v_3, v_5]$;

9. $\ell(\tau_9) = (2, 4, 5)$ representando a face $\tau_9 = [v_2, v_4, v_5]$;

10. $\ell(\tau_{10}) = (3, 4, 5)$ representando a face $\tau_{10} = [v_3, v_4, v_5]$;

Um algoritmo para gerar todas as faces de dimensão k de um simplexo de dimensão n é um mero exercício da utilização de combinações, como o trecho de código abaixo.

```
for i = 1:k+1
     DivisionC(i) = n-k+i;
end
[BaseC] = InitGrid(k+1,DivisionC);
for j = 0:BaseC(k+2)-1
 [C]   = GridCoords(k+1,j,BaseC);
     C     = C+1;
     [lex] = Lexico(k+1,C);
     if lex == 1
          C
     end
end
```

4.2 Os Vértices da Aproximação

Seja uma aplicação $F : S \to \mathbb{R}^k$ de classe C^2 no aberto $S \subset \mathbb{R}^n$ com $n > k$ e seja $\mathcal{M} = \{x \in S \mid F(x) = 0\}$. Suponha que $0 \in \mathbb{R}^k$ é valor regular de F, isto é, se $x \in \mathcal{M}$ então $DF(x)$ tem posto máximo. Neste caso $\mathcal{M}$ é uma variedade de dimensão $n - k$, pelo Teorema 2.1.

Caso $0 \in \mathbb{R}^k$ não seja valor regular de F, podemos escolher $\epsilon > 0$ pequeno e arbitrário de modo de $\mathcal{M}_\epsilon = \{x \in S \mid F(x) = \epsilon\}$ seja uma variedade. A existência de ϵ tal que $\mathcal{M}_\epsilon$ seja uma variedade é garantido pelo Teorema de Sard 2.3.

Dado um simplexo de dimensão k, $\sigma = [v_0, \ldots, v_k]$ a interseção de $\mathcal{M}$ com σ é aproximada por

$$v = \sum_{i=0}^{k} \lambda_i v_i$$

onde λ é dado por

$$\begin{pmatrix} 1 & \cdots & 1 \\ F(v_0) & \cdots & F(v_k) \end{pmatrix} \begin{pmatrix} \lambda_0 \\ \vdots \\ \lambda_k \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \tag{4.2.1}$$

Caso $\lambda_i \geq 0$, v pertence ao simplexo σ e caso $\lambda_i > 0$, v pertence ao interior do simplexo σ.

Para garantir que $\mathcal{M}$ e o interior de σ sejam transversais de acordo com a Definição 2.8 é necessário que a matriz da Equação 4.2.1 seja invertível e que a solução $\lambda_i > 0$.

Para calcular uma aproximação para $\mathcal{M}$ em um simplexo de dimensão n, basta obter o fecho convexo da aproximação para $\mathcal{M}$ em cada face de dimensão k, isto é, escolhendo todas as combinações de k vértices dos n vértices deste simplexo. No caso de que cada face de dimensão k de um simplexo σ de dimensão n tenha o interior transversal a $\mathcal{M}$ diremos que σ é transversal a $\mathcal{M}$.

No Apêndice A Seção A.1 é apresentado um trecho de código para o cálculo dos vétices da aproximação.

4.3 Perturbação

Observe que se a matriz do sistema linear definido em 4.2.1 for invertível, existe uma única solução para v. Pode-se escolher os vértices do simplexo σ com uma pequena perturbação de modo que esta matriz seja sempre invertível, e que $\lambda_i \neq 0$, concluindo que v pertence ao interior de σ ou não pertence a σ. Caso isto ocorra teremos o simplexo transversal a variedade.

A perturbação para o domínio todo pode ser feita a partir de um bloco inicial e estendido para todo o domínio por refexão. A figura 4.1 mostra como uma malha cartesiana de um paralelepípedo em $\mathbb{R}^2$ pode ser pertubada somando-se um valor fixo para cada vértice do bloco inicial na posição $g = (0, 0)$ e refletindo-se este para todo o domínio.

$$\begin{array}{ccccccccccccccc}
0 & -- & 1 & -- & 0 & -- & 1 & -- & 0 & -- & 1 & -- & 0 & -- & 1 \\
| & (0,5) & | & (1,5) & | & (2,5) & | & (3,5) & | & (4,5) & | & (5,5) & | & (6,5) & | \\
2 & -- & 3 & -- & 2 & -- & 3 & -- & 2 & -- & 3 & -- & 2 & -- & 3 \\
| & (0,4) & | & (1,4) & | & (2,4) & | & (3,4) & | & (4,4) & | & (5,4) & | & (6,4) & | \\
0 & -- & 1 & -- & 0 & -- & 1 & -- & 0 & -- & 1 & -- & 0 & -- & 1 \\
| & (0,3) & | & (1,3) & | & (2,3) & | & (3,3) & | & (4,3) & | & (5,3) & | & (6,3) & | \\
2 & -- & 3 & -- & 2 & -- & 3 & -- & 2 & -- & 3 & -- & 2 & -- & 3 \\
| & (0,2) & | & (1,2) & | & (2,2) & | & (3,2) & | & (4,2) & | & (5,2) & | & (6,2) & | \\
0 & -- & 1 & -- & 0 & -- & 1 & -- & 0 & -- & 1 & -- & 0 & -- & 1 \\
| & (0,1) & | & (1,1) & | & (2,1) & | & (3,1) & | & (4,1) & | & (5,1) & | & (6,1) & | \\
2 & -- & 3 & -- & 2 & -- & 3 & -- & 2 & -- & 3 & -- & 2 & -- & 3 \\
| & (0,0) & | & (1,0) & | & (2,0) & | & (3,0) & | & (4,0) & | & (5,0) & | & (6,0) & | \\
0 & -- & 1 & -- & 0 & -- & 1 & -- & 0 & -- & 1 & -- & 0 & -- & 1
\end{array}$$

Figura 4.1: Perturbação de uma malha em $\mathbb{R}^2$ a partir da reflexão de um bloco na posição $g = (0, 0)$. Nos centros os rótulos dos hipercubos e nos cantos os rótulos dos vértices.

Para perturbar toda a malha o procedimento é como segue:

1. Para cada vértice v do hipercubo unitário I^n, seja $w = rand(v, \epsilon)$ tal que $\| w \| < \epsilon$ (ϵ tão pequeno quanto se queira) o vértice correspondente do cubo unitário perturbado I^n_ϵ.

2. Considere a transformação que define uma reflexão na malha para os vértices de I^n, $R_g : \{0, 1\}^n \to \{0, 1\}^n$ definida por $R_g(v) = u$, onde

$$u_i = \begin{cases} v_i & \text{se } g_i \text{ é par} \\ 1 - v_i & \text{se } g_i \text{ é impar} \end{cases}$$

3. Então para perturbar um vértice v do hipercubo na posição g da malha, basta somar a perturbação do cubo I^n_ϵ refletido $v_\epsilon = v + rand(R_g(v), \epsilon)$.

No Apêndice A Seção A.2 é apresentado um trecho de código para o cálculo dos vétices dos hipercubos já pertubados com a perturbação descrita nesta Seção.

4.4 Regras de Pivoteamento

Dado um simplexo $\sigma = [v_0, \ldots, v_i, \ldots, v_n]$, para cada faceta $\tau_i = [v_0, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n]$, o simplexo $\sigma_i = [v_0, \ldots, \tilde{v}_i, \ldots, v_n]$ que também contém a faceta τ_i é denominado i-ésimo pivoteamento de σ ou pivoteado de σ pelo vértice v_i.

Regras de pivoteamento de uma triangulação são regras que permitem encontrar o i-ésimo pivoteado de um simplexo qualquer da triangulação. Este procedimento permite passear pelos simplexos vizinhos a um dado simplexo que tem uma faceta em comum apenas pivoteando cada vértice deste simplexo.

Para a triangulação K_1 de $\mathbb{R}^n$, a regra de pivoteamento é a seguinte :

Se $\sigma = K_1(v_0, \pi) = [v_0, v_1, \ldots, v_i, \ldots, v_n]$, o i-ésimo pivoteado de σ é o simplexo $\sigma_i = K_1(v_0', \pi')$ dado por v_0' e π' como a seguir :

$$i = 0 \quad \Rightarrow \begin{cases} v_0' &= v_0 + e_{\pi_1} \\ \pi' &= (\pi_2, \ldots, \pi_n, \pi_1) \end{cases}$$

$$0 < i < n \quad \Rightarrow \begin{cases} v_0' &= v_0 \\ \pi' &= (\pi_1, \ldots, \pi_{i-1}, \pi_{i+1}, \pi_i, \pi_{i+2}, \ldots, \pi_n) \end{cases}$$

$$i = n \quad \Rightarrow \begin{cases} v_0' &= v_0 - e_{\pi_n} \\ \pi' &= (\pi_n, \pi_1, \ldots, \pi_{n-1}) \end{cases}$$

Observe-se que para $0 < i < n$,

$$\begin{aligned} v_i' &= v_0 + e_{\pi_1'} + \cdots + e_{\pi_i'} = \\ &= v_0 + e_{\pi_1} + \cdots + e_{\pi_{i-1}} + e_{\pi_{i+1}} = \\ &= (v_0 + e_{\pi_1} + \cdots + e_{\pi_{i-1}}) - (v_0 + e_{\pi_1} + \cdots + e_{\pi_i}) + (v_0 + e_{\pi_1} + \cdots + e_{\pi_{i+1}}) = \\ &= v_{i+1} - v_i + v_{i-1}. \end{aligned}$$

Para a triangulação J_1 de $\mathbb{R}^n$, a regra de pivoteamento é a seguinte :

Se $\sigma = J_1(v_0, \pi, s) = [v_0, v_1, \ldots, v_i, \ldots, v_n]$ o i-ésimo pivoteado de σ é o simplexo $\sigma_i = J_1(v_0', \pi', s')$ dado por v_0' e π' como a seguir :

$$i = 0 \quad \Rightarrow \begin{cases} v_0' &= v_0 + 2s_{\pi_1} e_{\pi_1} \\ \pi' &= (\pi_1, \ldots, \pi_n) \\ s' &= s - 2s_{\pi_1} e_{\pi_1} \end{cases}$$

$$0 < i < n \quad \Rightarrow \begin{cases} v_0' &= v_0 \\ \pi' &= (\pi_1, \ldots, \pi_{i-1}, \pi_{i+1}, \pi_i, \pi_{i+2}, \ldots, \pi_n) \\ s' &= s \end{cases}$$

$$i = n \quad \Rightarrow \begin{cases} v_0' &= v_0 - 2s_{\pi_n} e_{\pi_n} \\ \pi' &= (\pi_n, \pi_1, \ldots, \pi_{n-1}) \\ s' &= s - 2s_{\pi_n} e_{\pi_n} \end{cases}$$

Considere as triangulações $v^f + \delta K_1$ e $v^f + \delta J_1$, com v^f e v^l pontos inicial e final de um paralelepípedo em $\mathbb{R}^n$, $N = (N_1, \ldots, N_n)$ e $\delta_i = (v_i^l - v_i^f)/N_i$. Cada simplexo de $v^f + \delta K_1$ e $v^f + \delta J_1$, $\sigma = K_1(g, \pi)$ e $\sigma = J_1(g, \pi)$ respectivamente, pode ser unicamente identificado pelo par (g, π), onde g define um bloco da malha e π uma permutação.

Para a triangulação $v^f + \delta K_1$ temos 3 tipos de regra de pivoteamento, como mostra o exemplo na figura 4.2.

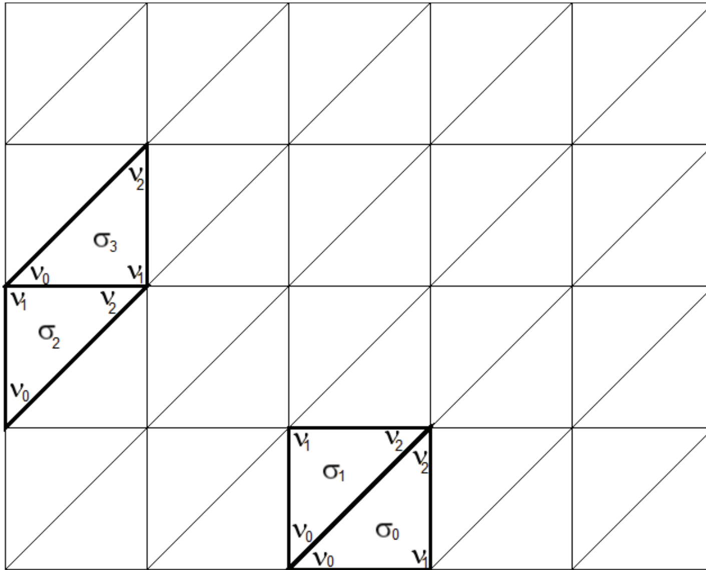

Figura 4.2: Triangulação K_1 de $\mathbb{R}^2$ com destaque a simplexos e seus pivoteados.

Para este exemplo temos :

$$
\begin{array}{llll}
\sigma_0 = K_1(g_0, \pi_0) & g_0 = (2,0) & \pi_0 = (1,2) & \ell(\sigma_0) = (0,1,3) \\
\sigma_1 = K_1(g_1, \pi_1) & g_1 = (2,0) & \pi_1 = (2,1) & \ell(\sigma_1) = (0,2,3) \\
\sigma_2 = K_1(g_2, \pi_2) & g_2 = (0,1) & \pi_2 = (2,1) & \ell(\sigma_2) = (0,2,3) \\
\sigma_3 = K_1(g_3, \pi_3) & g_3 = (0,2) & \pi_3 = (1,2) & \ell(\sigma_3) = (0,1,3)
\end{array}
$$

onde, σ_0 é o 1-ésimo pivoteado de σ_1, σ_1 é o 1-ésimo pivoteado de σ_0, σ_2 é o 2-ésimo pivoteado de σ_3 e σ_3 é o 0-ésimo pivoteado de σ_2.

Seja $\sigma = K_1(g, \pi) = [v_0, v_1, \ldots, v_n]$. O i-ésimo pivoteado de σ é o simplexo τ dado por:

Regra 1

$$
{\scriptstyle 0<i<n} \quad \Rightarrow \left\{ \begin{array}{lcl}
\tau & = & K_1(g, \pi') = [v_0, \ldots, v_{i-1}, v_{i-1} - v_i + v_{i+1}, v_{i+1}, \ldots, v_n] \\
\pi' & = & (\pi_1, \ldots, \pi_{i-1}, \pi_{i+1}, \pi_i, \pi_{i+2}, \ldots, \pi_n) \\
\ell(\tau) & = & (\ell(v_0), \ldots, \ell(v_{i-1}), \ell(v_i) - \ell(v_i) + \ell(v_{i+1}), \ell(v_{i+1}), \ldots \\
 & & \ldots, \ell(v_n))
\end{array} \right.
$$

Regra 2

$$\left.\begin{array}{r} i=0 \\ g_{\pi_1} < N_{\pi_1 - 1} \end{array}\right\} \Rightarrow \left\{\begin{array}{lcl} \tau & = & K_1(g', \pi') = [v_1, v_2, \ldots, v_n, v_0 + \delta_{\pi_1} e_{\pi_1}] \\ g' & = & g + e_{\pi_1} \\ \pi' & = & (\pi_2, \ldots, \pi_n, \pi_1) \\ \ell(\tau) & = & (\ell(v_1) - 2^{\pi_1 - 1}, \ldots, \ell(v_n) - 2^{\pi_1 - 1}, \ell(v_n)) \end{array}\right.$$

Regra 3

$$\left.\begin{array}{r} i=n \\ g_{\pi_n} > 0 \end{array}\right\} \Rightarrow \left\{\begin{array}{lcl} \tau & = & K_1(g', \pi') = [v_0 - \delta_{\pi_n} e_{\pi_n}, v_0, v_1, \ldots, v_{n-1}] \\ g' & = & g - e_{\pi_n} \\ \pi' & = & (\pi_n, \pi_1, \ldots, \pi_{n-1}) \\ \ell(\tau) & = & (\ell(v_0), \ell(v_0) + 2^{\pi_n - 1}, \ldots, \ell(v_{n-1}) + 2^{\pi_n - 1}) \end{array}\right.$$

Para a triangulação $v^f + \delta J_1$, temos 3 tipos de de pivoteamento, como mostra o exemplo na figura 4.3.

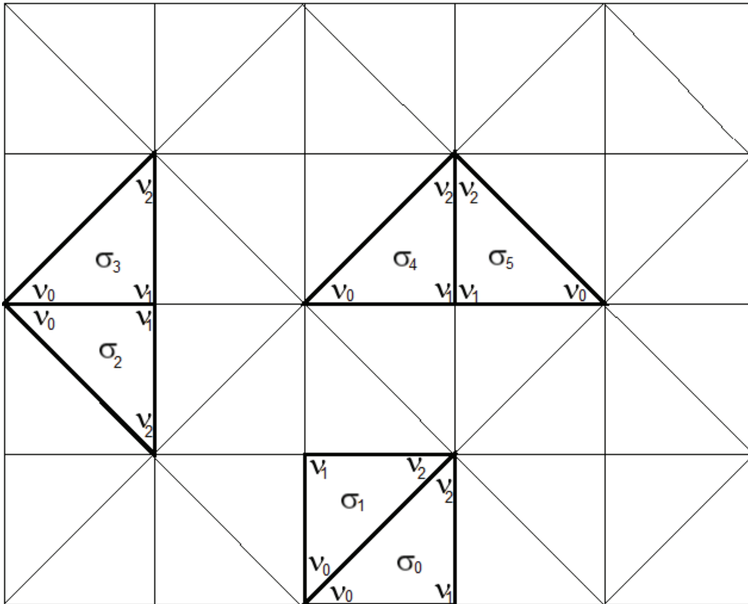

Figura 4.3: Triangulação J_1 de $\mathbb{R}^2$ com destaque a simplexos e seus pivoteados.

Para este exemplo temos :

$$\begin{array}{llll}
\sigma_0 = J_1(g_0,\pi_0) & g_0 = (2,0) & \pi_0 = (1,2) & \ell(\sigma_0) = (0,1,3) \\
\sigma_1 = J_1(g_1,\pi_1) & g_1 = (2,0) & \pi_1 = (2,1) & \ell(\sigma_1) = (0,2,3) \\
\sigma_2 = J_1(g_2,\pi_2) & g_2 = (0,1) & \pi_2 = (1,2) & \ell(\sigma_2) = (2,3,1) \\
\sigma_3 = J_1(g_3,\pi_3) & g_3 = (0,2) & \pi_3 = (1,2) & \ell(\sigma_3) = (0,1,3) \\
\sigma_4 = J_1(g_4,\pi_4) & g_4 = (2,2) & \pi_4 = (1,2) & \ell(\sigma_4) = (0,1,3) \\
\sigma_5 = J_1(g_5,\pi_5) & g_5 = (3,2) & \pi_5 = (1,2) & \ell(\sigma_5) = (1,0,2)
\end{array}$$

onde, σ_0 é o 1-ésimo pivoteado de σ_1, σ_1 é o 1-ésimo pivoteado de σ_0, σ_2 é o 2-ésimo pivoteado de σ_3, σ_2 é o 2-ésimo pivoteado de σ_3, σ_4 é o 0-ésimo pivoteado de σ_5 e σ_5 é o 0-ésimo pivoteado de σ_4.

Seja $\sigma = J_1(g,\pi) = [v_0, v_1, \ldots, v_n]$. O i-ésimo pivoteado de σ é o simplexo τ dado por:

Regra 1

$$_{0<i<n} \;\Rightarrow \left\{ \begin{array}{lll}
\tau & = & J_1(g,\pi') = [v_0, \ldots, v_{i-1}, v_{i-1} - v_i + v_{i+1}, v_{i+1}, \ldots, v_n] \\
\pi' & = & (\pi_1, \ldots, \pi_{i-1}, \pi_{i+1}, \pi_i, \pi_{i+2}, \ldots, \pi_n) \\
s' & = & s \\
\ell(\tau) & = & (\ell(v_0), \ldots, \ell(v_{i-1}), \ell(v_{i-1}) - \ell(v_i) + \ell(v_{i+1}), \ell(v_{i+1}), \ldots \\
 & & \ldots, \ell(v_n))
\end{array}\right.$$

Regra 2

$$\left.\begin{array}{r}
_{i=0} \\
_{g_{\pi_1} < N_{\pi_1 - 1}} \text{ ou} \\
_{g_{\pi_1} = K_{\pi_1 - 1}} \text{ é impar}
\end{array}\right\} \;\Rightarrow \left\{ \begin{array}{lll}
\tau & = & J_1(g',\pi') = [v_0 + 2s_{\pi_1}\delta_{\pi_1}e_{\pi_1}, v_1, \ldots, v_n] \\
g' & = & g + s_{\pi_1}e_{\pi_1} \\
\pi' & = & (\pi_1, \ldots, \pi_n) \\
s' & = & s - 2s_{\pi_1}e_{\pi_1} \\
\ell(\tau) & = & (\ell(v_0) + s_{\pi_1}2^{\pi_1 - 1}, \ell(v_1) - s_{\pi_1}2^{\pi_1 - 1}, \ldots \\
 & & \ldots, \ell(v_n) - s_{\pi_1}2^{\pi_1 - 1})
\end{array}\right.$$

Regra 3

$$\left.\begin{array}{r}
_{i=n} \\
_{g_{\pi_n} > 0}
\end{array}\right\} \;\Rightarrow \left\{ \begin{array}{lll}
\tau & = & J_1(g',\pi') = [v_0, v_1, \ldots, v_{n-1}, v_m - 2s_{\pi_n}\delta_{\pi_n}e_{\pi_n}] \\
g' & = & g - s_{\pi_n}e_{\pi_n} \\
\pi' & = & (\pi_n, \pi_1, \ldots, \pi_{n-1}) \\
s' & = & s - 2s_{\pi_n}e_{\pi_n} \\
\ell(\tau) & = & (\ell(v_0) + s_{\pi_n}2^{\pi_n - 1}, \ldots, \ell(v_{n-1}) + s_{\pi_n}2^{\pi_n - 1}, \\
 & & \ell(v_n) - s_{\pi_n}2^{\pi_n - 1})
\end{array}\right.$$

Observe-se que, se $\sigma = K_1(g,\pi) = [v_0, v_1, \ldots, v_n]$, então, se $g_{\pi_1} = N_{\pi_1 - 1}$, a face $[v_1, \ldots, v_n]$ pertence a fronteira de S e, se $g_{\pi_m} = 0$, a face $[v_0, \ldots, v_{n-1}]$ também pertence a fronteira de S.

Da mesma forma, se $\sigma = J_1(g, \pi) = [v_0, v_1, \ldots, v_n]$, então, se $g_{\pi_1} = N_{\pi_1} - 1$ é par, então a face $[v_1, \ldots, v_n]$ pertence a fronteira de S, e se $g_{\pi_n} = 0$, a face $[v_0, \ldots, v_{n-1}]$ também pertence a fronteira de S.

No Apêndice A Seção A.3 é apresentado um trecho de código para o pivoteamento de simplexos da triangulação K_1.

4.5 Esqueleto Combinatório

Dado um simplexo de dimensão n, σ, cada vértice de $\mathcal{M}_\sigma = \sigma \cap \mathcal{M}$ pertence a uma face τ, de dimensão k, de σ.

Cada face de dimensão i de $\mathcal{M}_\sigma$ está em uma face de dimensão $k+i$ de σ, para $i = 0, \ldots, n-k$, pela observação da completude da dimensão. Além disso cada face de dimensão $k+i$ de σ pertence a exatamente duas faces de dimensão $k+i+1$ de σ.

Assim, é possível gerar a partir dos rótulos dos simplexos toda relação de adjacência e incidência de $\mathcal{M}_\sigma$.

Por exemplo, considere o caso de $n = 4$ e $k = 1$ com um simplexo σ com a seguinte rotulagem de seus vértices $\ell(\sigma) = (0, 8, 12, 14, 15)$. Considere também que as seguintes arestas do simplexo sejam transversais à variedade, que representaremos apenas seus rótulos:

1. Vértices de $\mathcal{M}_\sigma$: número total 6
   (a) $\ell(\tau_1) = (0, 14)$, que está na aresta que une os vértices 0 e 14 de σ
   (b) $\ell(\tau_2) = (8, 14)$, que está na aresta que une os vértices 8 e 14 de σ
   (c) $\ell(\tau_3) = (12, 14)$, que está na aresta que une os vértices 12 e 14 de σ
   (d) $\ell(\tau_4) = (0, 15)$, que está na aresta que une os vértices 0 e 15 de σ
   (e) $\ell(\tau_5) = (8, 15)$, que está na aresta que une os vértices 8 e 15 de σ
   (f) $\ell(\tau_6) = (12, 15)$, que está na aresta que une os vértices 12 e 14 de σ
2. Arestas de $\mathcal{M}_\sigma$: número total 9
   (a) $\ell(\nu_1) = (0, 8, 14)$, que contém $\ell(\tau_1) = (0, 14)$ e $\ell(\tau_2) = (8, 14)$
   (b) $\ell(\nu_2) = (0, 12, 14)$, que contém $\ell(\tau_1) = (0, 14)$ e $\ell(\tau_3) = (12, 14)$
   (c) $\ell(\nu_3) = (8, 12, 14)$, que contém $\ell(\tau_2) = (8, 14)$ e $\ell(\tau_3) = (12, 14)$
   (d) $\ell(\nu_4) = (0, 8, 15)$, que contém $\ell(\tau_4) = (0, 15)$ e $\ell(\tau_5) = (8, 15)$
   (e) $\ell(\nu_5) = (0, 12, 15)$, que contém $\ell(\tau_4) = (0, 15)$ e $\ell(\tau_6) = (12, 15)$
   (f) $\ell(\nu_6) = (8, 12, 15)$, que contém $\ell(\tau_5) = (8, 15)$ e $\ell(\tau_6) = (12, 15)$
   (g) $\ell(\nu_7) = (0, 14, 15)$, que contém $\ell(\tau_1) = (0, 14)$ e $\ell(\tau_4) = (0, 15)$
   (h) $\ell(\nu_8) = (8, 14, 15)$, que contém $\ell(\tau_2) = (8, 14)$ e $\ell(\tau_5) = (8, 15)$
   (i) $\ell(\nu_9) = (12, 14, 15)$, que contém $\ell(\tau_3) = (12, 14)$ e $\ell(\tau_6) = (12, 15)$
3. Faces de $\mathcal{M}_\sigma$: número total 5
   (a) $\ell(\mu_1) = (0, 8, 12, 14)$, que contém $\ell(\nu_1) = (0, 8, 14)$, $\ell(\nu_2) = (0, 12, 14)$ e $\ell(\nu_3) = (8, 12, 14)$
   (b) $\ell(\mu_2) = (0, 8, 12, 15)$, que contém $\ell(\nu_4) = (0, 8, 15)$, $\ell(\nu_5) = (0, 12, 15)$ e $\ell(\nu_6) = (8, 12, 15)$

(c) $\ell(\mu_3) = (0, 8, 14, 15)$, que contém $\ell(\nu_1) = (0, 8, 14)$, $\ell(\nu_4) = (0, 8, 15)$, $\ell(\nu_7) = (0, 14, 15)$ e $\ell(\nu_8) = (8, 14, 15)$

(d) $\ell(\mu_4) = (0, 12, 14, 15)$, que contém $\ell(\nu_2) = (0, 12, 14)$, $\ell(\nu_5) = (0, 12, 15)$, $\ell(\nu_7) = (0, 14, 15)$ e $\ell(\nu_8) = (8, 14, 15)$

(e) $\ell(\mu_5) = (8, 12, 14, 15)$, que contém $\ell(\nu_3) = (8, 12, 14)$, $\ell(\nu_6) = (8, 12, 15)$, $\ell(\nu_8) = (8, 14, 15)$ e $\ell(\nu_9) = (12, 14, 15)$

4. Sólido de $\mathcal{M}_\sigma$: número total 1

   (a) $\ell(\sigma_1) = (0, 8, 12, 14, 15)$, que contém $\ell(\mu_1) = (0, 8, 12, 14)$, $\ell(\mu_2) = (0, 8, 12, 15)$, $\ell(\mu_3) = (0, 8, 14, 15)$, $\ell(\mu_4) = (0, 12, 14, 15)$ e $\ell(\mu_5) = (8, 12, 14, 15)$

Pode-se observar que este é um sólido contendo duas faces triangulares e três faces quadrangulares, isto é, um prisma de base triangular.

No Apêndice A Seção A.4 é apresentado um trecho de código para gerar o esqueleto combinatório.

4.6 Marching Simplex

O programa Marching Simplex percorre todos os simplexos da triangulação de uma malha cartesiana em n dimensões para obter uma aproximação de uma variedade $\mathcal{M}$ definida implicitamente, de dimensão $n - k$.

Neste livro utilizaremos a triangulação K_1. Será gerado todos os politopos que aproximam $\mathcal{M} \cap \sigma$ para cada simplexo σ da triangulação. Cada politopo de dimensão $n - k$ é a aproximação da inteseção da variedade com um simplexo da malha de entrada.

Os métodos utilizados são os descritos anteriormente neste Capítulo, os códigos no Apêndice A e o código Marching Simplex é apresentado no Apêndice B.

À medida que a dimensão n cresce, a quantidade de simplexos processados cresce de forma exponencial, ao ponto de ser inviável guardar todos os dados do programa simulataneamente em memória. Para resolver esse problema, usamos as técnicas explicadas nos capítulos anteriores.

A Tabela 4.1 mostra a economia de memória resultatante da implementação do Marching Simplex usando técnicas de contagem e enumeração, em contraste com uma implementação que guarda todos elementos processados em memória. Para que a economia seja efetiva, alguns vetores auxiliares são usados: um vetor Base que armazena a base das dimensões 1 a n da malha cartesiana , um vetor BaseP que armazena a base das permutações para cálculo da triangulação K_1 e um vetor BaseC que armazena a base de combinações para cálculo de faces de dimensão $n-k$ dos simplexos de dimensão n.

Tabela 4.1: Número de elementos processados pelo Marching Simplex e máximo de cada elemento alocado em memória para qualquer instante durante sua execução.

Tipo de elemento	Número total processado	Número máximo em memória
Hipercubos da Malha	$H = \prod_{i=1}^{n} Division(i)$	1
Vértices da Malha	$\prod_{i=1}^{n}\{Division(i) + 1\}$	$2^n = \prod_{i=1}^{n} 2$
Simplexos de dimensão n	$S = Hn!$	1
Faces de dimensão $n - k$	$S\binom{n}{k}$	1

Em contrapartida, por conta da economia de memória, alguns dados são calculados mais de uma vez como, por exemplo, o valor da função implícita nos vértices do domínio. Ainda assim, o Marching Simplex executa em poucos minutos (menos que dez) para os exemplos encontrados neste livro em um notebook com processador Intel Core i7 de 3.1 GHz, com 256KB de L2 Cache e 8MB de L3 Cache, também com 16GB de memória Ram. A economia de dados se dá por conta do uso de operações combinatórias que, por serem feitas com números inteiros, são rápidas.

4.6.1 Parâmetros de Entrada e Saída

Os dados de entrada para o código Marching Simplex são os seguintes:

1. Dimensões do domínio e contradomínio n e k respectivamente;
2. *First*: Ponto inicial da malha em $\mathbb{R}^n$;
3. *Last*: Ponto final da malha em $\mathbb{R}^n$;
4. *Division*: número de divisões em cada dimensão da malha;
5. Função que define a variedade implícita $\mathcal{M}$;
6. Nome do arquivo de saída em que será escrito o esqueleto combinatório que aproxima a variedade $\mathcal{M}$ em cada simplexo σ.

A saída possui a seguinte estrutura:

```
n   k
[Division]

<esqueleto combinatório de politopos na célula 0 da malha>
<esqueleto combinatório de politopos na célula 1 da malha>

 ...

<esqueleto combinatório de politopos na célula Base(n+1)-1 da malha>

-1
```

Considerando sua definição no final do Capítulo 2, o esqueleto combinatório de politopos em uma célula da malha é escrito da seguinte forma:

```
número da célula [rótulos das coordenadas da célula]
1
número de vértices da aproximação
<lista de vértices>
número de arestas da aproximação
<lista de arestas>
...
número de politopos de dimensão n-k da aproximação
<lista de politopos>
```

Os rótulos das coordenadas de uma célula i tem valores inteiros definidos pela função $GridCoords(n, i, Base)$, descrita no Capítulo 3.

A saída do programa escreve um vértice, aresta ou politopo de qualquer dimensão por linha.

O rótulo de um vértice, aresta ou politopo de qualquer dimensão é igual à linha em que ele aparece em sua respectiva lista (começando por 1).

Para cada vértice v da aproximação, as seguintes informações são escritas em sequência:

1. rótulos dos dois vértices da malha que são extremidades da aresta na qual v se encontra;

2. coordenadas cartesianas de v.

Para cada célula de dimensão $j > 0$, escreve-se os rótulos das células de dimensão $j - 1$ de sua fronteira.

4.7 Continuation Simplex

Com a finalidade de economizar ainda mais tempo de processamento, o programa Continuation Simplex, modifica o Marching Simplex para processar apenas hipercubos da malha cartesiana que são transversais à variedade $\mathcal{M}$. Em contrapartida, um pré-processamento, que explicaremos adiante, se faz necessário. O Continuation Simplex é especialmente vantajoso quando a variedade $\mathcal{M}$ é esparsa, ou seja, ocupa uma fração diminuta do domínio. Nestes casos, a economia de tempo ao longo da execução é bem maior que o custo do pré processamento.

4.7.1 Métodos de Continuação

Métodos de continuação e métodos de homotopia têm sido utilizados à bastante tempo na matemática moderna, e por matemáticos de renome, tais como Poincaré (1881-1886) [49], Klein (1882-1883) [31] e Bernstein (1910) [3].

Métodos de homotopia consistem no seguinte. Suponha que queremos a solução de um sistema de equações não lineares,

$$F(x) = 0,$$

onde $F : \mathbb{R}^n \to \mathbb{R}^n$ é uma aplicação suave, para simplificar.

Definimos uma homotopia ou deformação $H : \mathbb{R}^n \times \mathbb{R} \to \mathbb{R}^n$ tal que

$$H(x, 1) = G(x), \qquad H(x, 0) = F(x),$$

onde $G : \mathbb{R}^n \to \mathbb{R}^n$ é uma aplicação suave e trivial (que conhecemos seus zeros) e H também suave. Uma homotopia típica é a convexa, tal que

$$H(x, \lambda) := \lambda\, G(x) + (1 - \lambda)\, F(x).$$

Considere uma curva $c(s) \in H^{-1}(0)$ que passa pelo ponto $(x_1, 1)$ $(x_1 \in G^{-1}(0))$ a um ponto $(\bar{x}, 0)$. Pela homotopia implica que $\bar{x} \in F^{-1}(0)$ e portanto uma solução do problema inicial.

Existem algumas técnicas para seguir esta curva, e uma delas é fazendo uma parametrização por comprimento de arco s da curva c e resolvendo a equação:

$$H(c(s)) = 0, \tag{4.7.2}$$

e um problema de valor inicial da seguinte equação diferencial:

$$DH(c)\,\dot{c} = 0, \qquad \| \dot{c} \| = 1, \qquad c(0) = (x_1, 1). \tag{4.7.3}$$

Desde os anos 70, vários autores contribuiram para o crescimento da literatura relativa a métodos denominados métodos de continuação do tipo preditor-corretor, que consite em predizer uma solução da equação 4.7.2 utilizando um método numérico para o problema de valor inicial e posteriormente corrigir (equação 4.7.3) com um método do tipo quase Newton.

Eaves (1972) [16] observou que uma classe de algoritmos para obter soluções podem obtidos por considerar uma aproximação linear por partes de H. Esta classe de métodos que emergiu em paralelo com os métodos de continuação do tipo preditor-corretor denominou-se métodos de continuação simplicial.

Os métodos de continuação simplicial para o problema de homotopia consiste em, a partir de uma triangulação de $\mathbb{R}^n \times [0,1]$, considerar uma aproximação linear por partes em cada simplexo e seguir a curva linear por partes que aproxima $c(s)$ a partir do ponto $(x_1, 1)$. Posteriormente estes métodos foram generalizados para aproximar $c(s) \in F^{-1}(0)$, com $F : \mathbb{R}^{n+1} \to \mathbb{R}^n$, a partir de uma condição inicial $c(0)$.

Para seguir a curva a partir de uma condição inicial, o princípio *door-in door-out* que diz que se a curva entra por uma face de dimensão n de um simplexo de dimensão $n+1$, ela tem que sair por apenas uma outra face de dimensão n. Este princípio juntamente com as regras de pivoteamento da triangulação de $\mathbb{R}^{n+1}$ permite passar de um simplexo para outro da triangulação até obter a aproximação de $c(s)$ no domínio de interesse a partir da condição inicial.

Posteriormente este método foi generalizado para aproximar componentes conexas de variedades $\mathcal{M} \in F^{-1}(0)$, com $F : \mathbb{R}^n \to \mathbb{R}^k$, $n - k > 1$, a partir de uma condição inicial $x_0 \in \mathcal{M}$. Assim como no caso de curvas, as regras de pivoteamento da triangulação são utilizadas para saber quais simplexos vizinhos a um dado simplexo transversal também são transversais, e o princípio *door-in door-out* para gerar o esqueleto combinatório de cada politopo que aproxima a variedade em um dado simplexo.

4.7.2 Condição Inicial

Como descrito na seção anterior, um método de continuação simplicial necessita de uma dada condição inicial $x_0 \in \mathcal{M} \in F^{-1}(0)$, com $F : \mathbb{R}^n \to \mathbb{R}^k$, $n > k$, encontrar um simplexo de dimensão n, σ, que contenha x_0 em seu interior.

A seguir vamos descrever um procedimento que encontra um simplexo de uma triangulação que contenha um determinado ponto.

Dado um simplexo $\sigma = [v_0, \ldots, v_n]$ de $\mathbb{R}^n$ e um ponto $x \in \mathbb{R}^n$, saber se x pertence ao interior de σ, a fronteira de σ, ou ao exterior de σ pode ser respondido com o cálculo das coordenadas baricêntricas de x relativo ao simplexo σ. O cálculo das coordenadas baricêntricas de x é dado por:

$$\begin{pmatrix} 1 & \cdots & 1 \\ v_0 & \cdots & v_n \end{pmatrix} \begin{pmatrix} \lambda_0 \\ \vdots \\ \lambda_n \end{pmatrix} = \begin{pmatrix} 1 \\ x \end{pmatrix}$$

Em outras palavras, utilizando Cramer

$$\lambda_i = \frac{det\begin{pmatrix} 1 & \cdots & 1 & \cdots & 1 \\ v_0 & \cdots & x & \cdots & v_n \end{pmatrix}}{det\begin{pmatrix} 1 & \cdots & 1 & \cdots & 1 \\ v_0 & \cdots & v_i & \cdots & v_n \end{pmatrix}}$$

que é o volume orientado do simplexo $\tau = [v_0, \ldots, x, \ldots, v_n]$ dividido pelo volume orientado do simplexo $\sigma = [v_0, \ldots, v_i, \ldots, v_n]$.

Caso algum $\lambda_i < 0$, o ponto x pertence ao exterior do simplexo σ, caso todos $\lambda_i > 0$, o ponto x pertence ao interior do simplexo σ e caso todos $\lambda_i \leq 0$ e algum $\lambda_i = 0$, o ponto x pertence a fronteira de σ.

A coordenada baricêntrica de um ponto $x \in \mathbb{R}^n$ com relação a um simplexo de $\mathbb{R}^n$, $\sigma = [v_0, \ldots, v_n]$, λ_i para $i = 0, \ldots, n$, é a menor distância com sinal do ponto x até o hiperplano definido pelos vértices do simplexo retirando o vértice v_i, $aff(v_0, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n)$, dividida pela menor distância com sinal do vértice v_i até o mesmo hiperplano. Esta observação segue, desde que o volume orientado de σ é igual a metade do volume orientado da faceta $[v_0, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n]$ vezes a altura definida pela distância orientada do vértice v_i até o hiperplano $aff(v_0, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n)$.

Se o ponto x estiver sobre o hiperplano, o valor da coordenada baricêntrica λ_i é zero, caso x estiver do lado oposto ao vértice v_i, λ_i terá sinal negativo e se x estiver do mesmo lado do vértice v_i com relação ao hiperplano, λ_i terá sinal positivo.

A Figura 4.4 ilustra o significado geométrico das coordenadas baricêntricas para o caso bidimensional. Nesta figura, para simplexo $\sigma = [v_1, v_2, v_3]$, temos $\lambda_1 = dx1/dv1$, $\lambda_2 = dx2/dv2$ e $\lambda_3 = -dx3/dv3$.

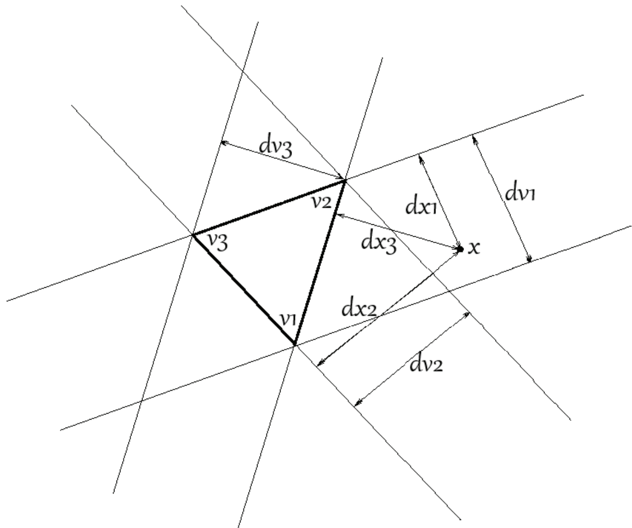

Figura 4.4: Interpretação geométrica das coordenadas baricêntricas

Um algoritimo para dado um ponto $x \in \mathbb{R}^n$ e uma triangulação T de um paralelepípedo de $\mathbb{R}^n$, encontrar um simplexo σ de T que contenha x é definido pelos

seguintes passos:

1. Escolha um simplexo σ_0 de T como passo inicial. Caso a triangulação T forneça uma boa estimativa, tal como em qual hipercubo da malha o ponto x esteja, escolha um simplexo deste hipercubo;
2. Faça $\sigma = \sigma_0$;
3. Calcule as coordenadas baricêntricas de x com relação a σ;
4. Se todas as coordenadas baricêntricas de x forem positivas, retorne σ como o simplexo que contém x;
5. Caso contrário, encontre a coordenada baricêntrica λ_i negativa com menor valor;
6. Encontre o simplexo de T, σ_i que é o pivoteado de σ pelo vértice v_i;
7. Faça $\sigma = \sigma_i$ e retorne ao passo 3;

Este processo pode ser visto na sequência de simplexos gerado pelo algoritimo acima para uma caso bidimensional na Figura 4.5. Nesta Figura podemos ver a sequência de simplexos da triangulação K_1, $\sigma_0, \sigma_1, \ldots, \sigma_{10}$, com x no interior de σ_{10}.

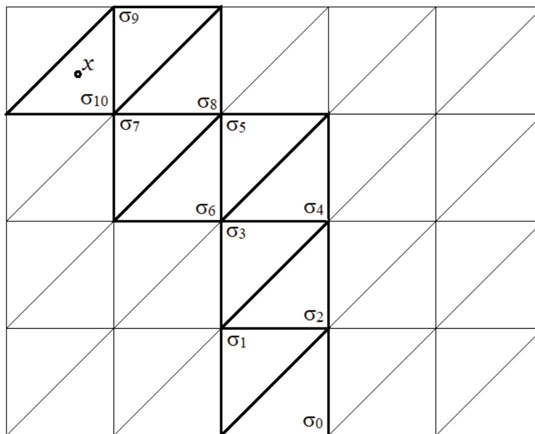

Figura 4.5: Sequência de simplexos até encontrar o simplexo σ_{10} contendo x

No Apêndice A Seção A.5 é apresentado um trecho de código que encontra um simplexo inicial utilizando as regras de pivoteamento com a triangulação K_1.

4.7.3 Estruturas de Dados

Os métodos de continuação simplicial, consistem em a partir de um ponto na variedade, encontrar um simplexo da triangulação que contenha este ponto e, a partir deste simplexo transversal, criar uma lista de simplexos adjacentes a este que

também sejam transversais. Daí para cada simplexo transversal que ainda não foi processado, deve-se processá-lo e colocá-lo em uma lista de simplexos já processados. Para isto são necessárias duas listas, uma dos simplexos que ainda tem que ser processados e outra dos simplexos que já foram processados (para que não sejam novamente processados).

No caso das triangulações K_1 ou J_1, podemos criar listas de pares (g, s), como o rótulo do hipercubo g, e o rótulo do simplexo s. Para estas listas precisamos de apenas três funções: uma para inserir elemento na lista, outra para eliminar elementos da lista e outra para pesquisar elementos em uma lista.

Vamos processar primeiramente os simplexos de um mesmo hipercubo antes de passar para o próximo hipercubo, para aproveitar as rotinas já implementadas do Marching Simplex.

No Apêndice A Seção A.6 é apresentado um trecho de código responsável pela pesquisa, inserção e remoção de simplexos das listas de simplexos processados e da lista dos simplexos que ainda estão por ser processados.

4.7.4 Parâmetros de Entrada e Saída

O código Continuation Simplex percorre todos os simplexos de dimensão n da triangulação K_1 que são transversais a variedade $\mathcal{M}$ e que estão dentro do domínio definido. Esta varredura é feita a partir de um ponto inicial em $\mathcal{M}$.

O código Continuation Simplex é apresentado no Apêndice C e as rotinas utilizadas são as descritas anteriormente no Capítulo 3 e nas seções iniciais deste capítulo, e os códigos são apresentados no Apêndice A.

Os dados de entrada do código Continuation Simplex são os seguintes:

1. Dimensões do domínio e contradomínio n e k respectivamente;
2. *Fisrt*: Ponto inicial da malha em $\mathbb{R}^n$;
3. *Last*: Ponto final da malha em $\mathbb{R}^n$;
4. *Division*: número de divisões em cada dimensão da malha;
5. *FirstPoint*: Ponto inicial da variedade $\mathcal{M}$;
6. Função que define a variedade implícita $\mathcal{M}$;
7. Nome do arquivo de saída em que será escrito o esqueleto combinatório que aproxima a variedade $\mathcal{M}$ em cada simplexo σ.

A saída possui a mesma estrutura do código MarchingSimplex:

```
n   k
[Division]

<esqueleto combinatório de politopos na célula 0 da malha>
<esqueleto combinatório de politopos na célula 1 da malha>

 ...

<esqueleto combinatório de politopos na célula Base(n+1)-1 da malha>

-1
```

4.8 Exemplos

Os exemplos das próximas subseções podem ser utilizados tanto com o código Marching Simplex quanto com o código Continuation Simplex.

4.8.1 Exemplos da Utilização do Marching Simplex

A seguir exibe-se um programa para gerar uma aproximação da esfera de dimensão 3, S^3 em $\mathbb{R}^4$ com centro na origem e raio 1 utilizando o programa Marching Simplex:

O programa a seguir que utiliza o código Marching Simplex gera os dados de entrada que do paralelepípedo que define o domínio, o número de divisões deste paralelepípedo para gerar o grid a ser varrido, a função que gera a esfera e o arquivo onde será escrita a saída de politopos da aproximação da estera, e finalmente aciona a função que implementa o método Marching Simplex.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Função para gerar uma aproximação de uma esfera de dimensão 3
% em R^4
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function Esf4
    % First: ponto extremo inicial do paralelepípedo que será gerado
    % o grid
    First    = [-1.1 -1.1 -1.1 -1.1];
    % Last: ponto extremo final do paralelepípedo que será gerado
    % o grid
    Last     = [1.1 1.1 1.1 1.1];
    % Division: número de divisões em cada direção para gerar o grid
    Division = [10 10 10 10];
    % Função que gera a aproximação da esfera utilizando o método
    % Marching Simplex
    MarchingSimplex(4,1,First,Last,Division,@esfera4,'esfera4_MS.pol');
    return

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Função cujo conjunto dos zeros é a esfera unitária S^3 em R^4
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    function [f] = esfera4(n,v)
      f(1) = v(1)*v(1) + v(2)*v(2) + v(3)*v(3) + v(4)*v(4) - 1;
      return
    end
end
```

Um trecho da saída, escrita no arquivo *esfera4_MS.pol*, é dado a seguir:

```
  4   1
  3   3   3   3

  0   0   0   0   0
  1
  4
  0  15     -0.44545473     -0.44545564     -0.44545416     -0.44545351
  8  15     -0.47171715     -0.47171840     -0.47171697     -0.36666583
 12  15     -0.52424256     -0.52424339     -0.36666636     -0.36666604
```

```
14  15     -0.68181883     -0.36666694     -0.36666638     -0.36666580
 6
 1   2
 1   3
 2   3
 1   4
 2   4
 3   4
 4
 1   2   3
 1   4   5
 2   4   6
 3   5   6
 1
 1   2   3   4

 0   0   0   0   0
 1
 4
 0  15     -0.44545473     -0.44545564     -0.44545416     -0.44545351
 8  15     -0.47171715     -0.47171840     -0.47171697     -0.36666583
12  15     -0.52424256     -0.52424339     -0.36666636     -0.36666604
13  15     -0.36666681     -0.68181876     -0.36666632     -0.36666603
 6
 1   2
 1   3
 2   3
 1   4
 2   4
 3   4
 4
 1   2   3
 1   4   5
 2   4   6
 3   5   6
 1
 1   2   3   4

 ...
 ...

 1   1   0   0   0
 1
 6
 0  14     -0.36666658     -0.47171817     -0.47171698     -0.47171630
 8  14     -0.36666623     -0.52424347     -0.52424235     -0.36666572
12  14     -0.36666681     -0.68181876     -0.36666632     -0.36666603
 0  15      0.26161678     -0.47171645     -0.47171768     -0.47171679
 8  15      0.20909166     -0.52424191     -0.52424301     -0.36666615
12  15      0.05151589     -0.68181765     -0.36666677     -0.36666634
 9
 1   2
 1   3
 2   3
 4   5
```

```
4   6
5   6
1   4
2   5
3   6
5
1   2   3
4   5   6
1   4   7   8
2   5   7   9
3   6   8   9
1
1   2   3   4   5

...
...
```

A Figura 4.6 mostra as células de saída que foram escritas acima usando o software de visualização mencionado no início do Capítulo 1. Observe que todas as células convexas definidas por 4 pontos são tetraedros e por 6 pontos são prismas com base triangular, mas poderiam ser tetraedros, prismas, piramides, hexaedros, entre outros. A Figura 4.7 mostra a esfera S^3 em $\mathbb{R}^4$ completa, e um corte por um hiperplano.

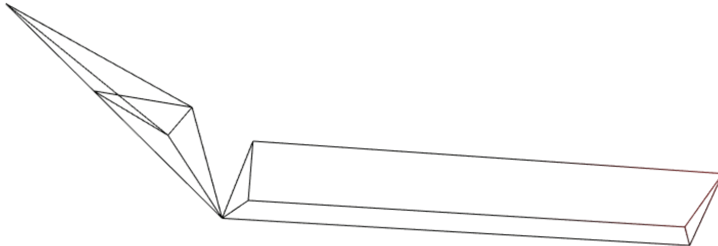

Figura 4.6: Subconjunto de células da aproximação de uma esfera 3D.

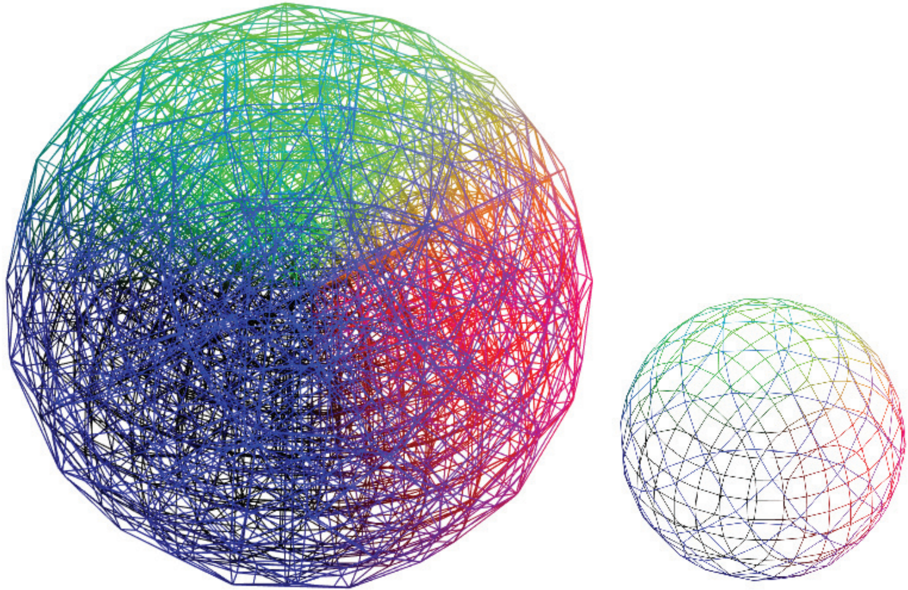

Figura 4.7: Esquerda: Projeção da aproximação de esfera tridimensional S^3 definida em $\mathbb{R}^4$; Direita: Corte da aproximação da esfera a esquerda por um hiperplano.

Um outro exemplo é o toro $S^1 \times S^1$ de dimensão 2 em $\mathbb{R}^4$, definido por

$$\begin{cases} x^2 + y^2 & = 1 \\ u^2 + v^2 & = \frac{1}{4} \end{cases}$$

O seguinte programa aproxima o toro usando Marching Simplex:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Função para gerar uma aproximação de um toro de dimensão 2
% em R^4
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function Tor4
    % First: ponto extremo inicial do paralelepípedo que será gerado
    % o grid
    First    = [-1.1 -1.1 -0.51 -0.51];
    % Last: ponto extremo final do paralelepípedo que será gerado
    % o grid
    Last     = [1.1 1.1 0.51 0.51];
    % Division: número de divisões em cada direção para gerar o grid
    Division = [10 10 10 10];
    % Função que gera a aproximação do toro utilizando o método
    % Marching Simplex
    MarchingSimplex(4,2,First,Last,Division,@toro,toro.pol');
    return
```

```
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Função cujo conjunto dos zeros é toro S^1 x S^1 em R^4
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    function [f] = toro(n,v)
      f(1) = v(1)*v(1) + v(2)*v(2) - 1;
      f(2) = v(3)*v(3) + v(4)*v(4) - 0.25;
      return
    end
end
```

Um trecho da saída, escrita no arquivo *toro.pol*, é dado a seguir:

```
  4   2
 10  10  10  10

302   2   0   3   0
  1
  3
  0   8  15     -0.45038738     -0.89039056     -0.12566715     -0.48292561
  0  12  15     -0.45038736     -0.89039058     -0.12361153     -0.48361082
  0  14  15     -0.45717100     -0.88662188     -0.12361150     -0.48361083
  3
  1   2
  1   3
  2   3
  1
  1   2   3

302   2   0   3   0
  1
  3
  0   8  15     -0.45038738     -0.89039056     -0.12566715     -0.48292561
  0  12  15     -0.45038736     -0.89039058     -0.12361153     -0.48361082
  0  13  15     -0.44661876     -0.89248424     -0.12361154     -0.48361081
  3
  1   2
  1   3
  2   3
  1
  1   2   3

...
...

302   2   0   3   0
  1
  4
  0   8  15     -0.45038738     -0.89039056     -0.12566715     -0.48292561
  0   9  15     -0.44422070     -0.89381650     -0.12753581     -0.48230273
  8  11  15     -0.45038750     -0.89039053     -0.13444705     -0.47999899
  9  11  15     -0.43999734     -0.89616283     -0.13444713     -0.47999896
  4
  1   2
  1   3
```

```
  2   4
  3   4
  1
  1   2   3   4

302   2   0   3   0
  1
  4
  0  12  15     -0.45038736    -0.89039058    -0.12361153    -0.48361082
  4  12  15     -0.45038735    -0.89039058    -0.11999975    -0.48481476
  0  14  15     -0.45717100    -0.88662188    -0.12361150    -0.48361083
  4  14  15     -0.45319759    -0.88882933    -0.11999974    -0.48481476
  4
  1   2
  3   4
  1   3
  2   4
  1
  1   2   3   4

302   2   0   3   0
  1
  4
  0  12  15     -0.45038736    -0.89039058    -0.12361153    -0.48361082
  4  12  15     -0.45038735    -0.89039058    -0.11999975    -0.48481476
  0  13  15     -0.44661876    -0.89248424    -0.12361154    -0.48361081
  4  13  15     -0.44882615    -0.89125791    -0.11999975    -0.48481476
  4
  1   2
  3   4
  1   3
  2   4
  1
  1   2   3   4

302   2   0   3   0
  1
  3
  0  14  15     -0.45717100    -0.88662188    -0.12361150    -0.48361083
  2  14  15     -0.46909008    -0.88000017    -0.12361146    -0.48361086
 10  14  15     -0.46909016    -0.88000015    -0.13444710    -0.47999897
  3
  1   2
  1   3
  2   3
  1
  1   2   3

...
...
```

A Figura 4.8 mostra as células de saída que foram escritas acima. Observe que todas as células convexas desta saída são triângulos, quadriláteros e pentágonos. A Figura 4.9 mostra o toro completo.

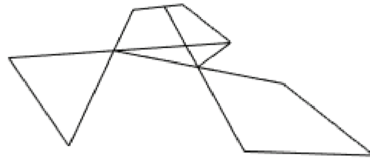

Figura 4.8: Subconjunto de células da aproximação de um toro 2D.

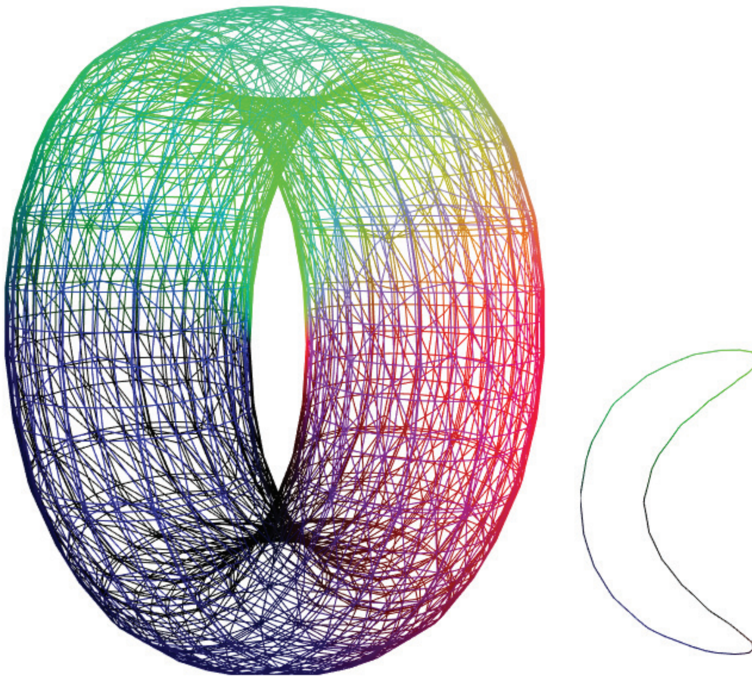

Figura 4.9: Esquerda: Projeção de uma aproximação do toro bidimensional $S^1 \times S^1$ definido em $\mathbb{R}^4$; Direita: Corte da aproximação do toro a esquerda por um hiperplano.

Na figura 4.9, lado direito, observe que o corte efetuado pelo hiperplano no toro gerou uma curva que é homeomorfa a S^1, para outras escolhas poderíamos ter duas curvas homeomorfas a S^1.

Observe que o Marching Simplex varre todos os simplexos de dimensão n da triangulação, sendo tranversais ou não a variedade. É interessante uma alternativa que somente varra os simplexos transversais a variedade, só que neste caso uma

componente conexa por vez.

Um programa que responde a esta espectativa é o que será apresentado na próxima Seção sobre o Método de Continuação Simplicial que a partir de uma semente (um ponto pertencente a uma componente conexa da variedade) percorra a partir do simplexo que contém este ponto, todos os simplexos tranversais desta componente.

4.8.2 Exemplos da Utilização do Continuation Simplex

Como exemplo da utilização do código *Continuation Simplex* vamos aproximar curvas no plano complexo, isto é, $F : \mathbb{C}^2 \to \mathbb{C}$, $F(z, w) = 0$. Tais curvas também podem ser aproximadas pelo código *MarchingSimplex*, vamos utilizar aqui o código *ContinuationSimplex* para diversificar, e também pelo motivo que o tempo de execução do código *ContinuationSimplex* é inferior ao do código *MarchingSimplex* devido a somente processar simplexos transversais a variedade.

Como primeiro exemplo, vamos aproximar o gráfico da função cosseno, $z = cos(w)$, com $z, w \in \mathbb{C}$. Reescrevendo as coordenadas reais e imaginárias, esta temos

$$\begin{cases} x &= \frac{e^{-v}cos(u) + e^{v}cos(-u)}{2} \\ y &= \frac{e^{-v}sen(u) + e^{v}sen(-u)}{2} \end{cases} \quad \text{com} \quad \begin{cases} z &= x + iy \\ w &= u + iv \end{cases}$$

O código a seguir gera esta aproximação e a Figura 4.10 apresenta uma projeção de $z = cos(w)$ gerado por este código.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Função Complex1
%     Esta função retorna o esqueleto combinatório da aproximação
%     de z = cos(w), com z e w complexo, utilizando o método de
%     continuação simplicial com a triangulação K_1.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function Complex1
    % Dimensão do domínio
    n          = 4;
    % Dimensão do contradomínio
    k          = 2;
    % Ponto inicial da malha em R^n
    First      = [-pi -pi -pi -pi];
    % Ponto final da malha em R^n
    Last       = [pi pi pi pi];
    % Número de divisões em cada dimensão da malha
    Division   = [10 10 10 10];
    % Ponto inicial da variedade
    FirstPoint = [0.0 0.0 1.0 0.0];

    % Metodo de continuação simplicial para a trinagulação K_1
    ContinuationSimplex(n,k,First,Last,Division,FirstPoint,@zcosw,'zcosw.pol');
    return

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % z = cos(w), w = x(1) + i x(2), z = x(3) + i x(4)
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
    function [f] = zcosw(n,x)
       % Parte real
       f(1) = x(3)-0.5*exp(-x(2))*cos(x(1))-0.5*exp(x(2))*cos(-x(1));
       % Parte imaginária
       f(2) = x(4)-0.5*exp(-x(2))*sin(x(1))-0.5*exp(x(2))*sin(-x(1));
       return
    end
end
```

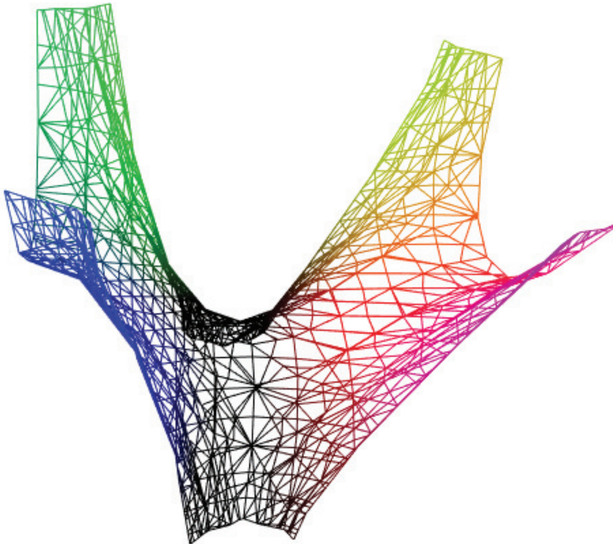

Figura 4.10: Projeção da aproximação de $z = cos(w)$.

No próximo exemplo vamos aproximar uma curva complexa um pouco mais sofisticada , $z^2 = cos(w)$, $z, w \in \mathbb{C}$. Reescrevendo as coordenadas reais e imaginárias desta equação temos

$$\begin{cases} x^2 - y^2 & = & \frac{e^{-v}cos(u) + e^{v}cos(-u)}{2} \\ 2xy & = & \frac{e^{-v}sen(u) + e^{v}sen(-u)}{2} \end{cases} \quad \text{com} \quad \begin{cases} z & = & x + iy \\ w & = & u + iv \end{cases}$$

O código a seguir gera esta aproximação e a Figura 4.11 apresenta uma projeção da curva complexa $z^2 = cos(w)$ gerado por este código.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Função Complex2
%      Esta função retorna o esqueleto combinatório da aproximação
```

```
%     de z^2 = cos(w), com z e w complexo, utilizando o método de
%     continuação simplicial com a triangulação K_1.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function Complex2
    % Dimensão do domínio
    n          = 4;
    % Dimensão do contradomínio
    k          = 2;
    % Ponto inicial da malha em R^n
    First      = [-pi -pi -pi -pi];
    % Ponto final da malha em R^n
    Last       = [pi pi pi pi];
    % Número de divisões em cada dimensão da malha
    Division   = [10 10 10 10];
    % Ponto inicial da variedade
    FirstPoint = [0.0 0.0 1.0 0.0];

    % Metodo de continuação simplicial para a trinagulação K_1
    ContinuationSimplex(n,k,First,Last,Division,FirstPoint,@z2cosw,'z2cosw.pol');
    return

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % z^2 = cos(w), w = x(1) + i x(2), z = x(3) + i x(4)
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    function [f] = z2cosw(n,x)
       % Parte real
       f(1) = x(3)*x(3)-x(4)*x(4)-0.5*exp(-x(2))*cos(x(1))-0.5*exp(x(2))*cos(-x(1));
       % Parte imaginária
       f(2) = 2.0*x(3)*x(4)-0.5*exp(-x(2))*sin(x(1))-0.5*exp(x(2))*sin(-x(1));
       return
    end
end
```

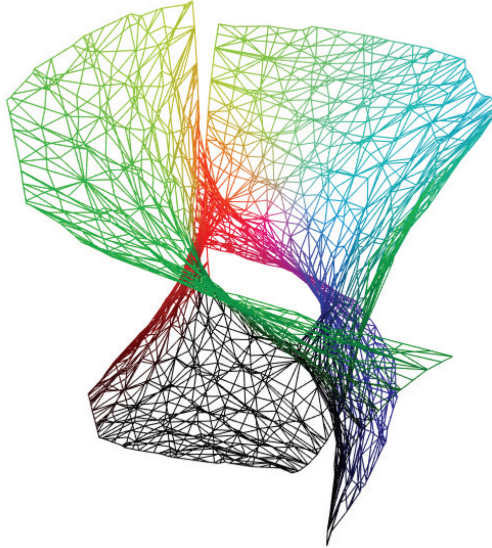

Figura 4.11: Projeção da aproximação de $z^2 = cos(w)$.

Para encerrar exemplos de curvas complexas, vamos aproximar uma curva polinomial , $z^2 = w^5 + 1 + i$, $z, w \in \mathbb{C}$. Reescrevendo as coordenadas reais e imaginárias desta equação temos

$$\left\{ \begin{array}{lcl} x^2 - y^2 & = & u^5 + 10u^3v^2 + 5uv^4 + 1 \\ 2xy & = & v^5 + 10u^2v^3 + 5u^4v + 1 \end{array} \right. \quad \text{com} \quad \left\{ \begin{array}{lcl} z & = & x + iy \\ w & = & u + iv \end{array} \right.$$

O código a seguir gera esta aproximação e a Figura 4.12 apresenta uma projeção da curva complexa $z^2 = w^5 + 1 + i$ gerada por este código.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Função Complex3
%     Esta função retorna o esqueleto combinatório da aproximação
%     de z^2 = w^5 + (1+i), com z e w complexo, utilizando o método
%     de continuação simplicial com a triangulação K_1.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function Complex3
    % Dimensão do domínio
    n          = 4;
    % Dimensão do contradomínio
    k          = 2;
    % Ponto inicial da malha em R^n
    First      = [-4.0 -4.0 -4.0 -4.0];
    % Ponto final da malha em R^n
    Last       = [4.0 4.0 4.0 4.0];
    % Número de divisões em cada dimensão da malha
```

```
    Division    = [12 12 12 12];
    % Ponto inicial da variedade
    FirstPoint = [0.0 -1.0 1.0 0.0];

    % Metodo de continuação simplicial para a trinagulação K_1
    ContinuationSimplex(n,k,First,Last,Division,FirstPoint,@z2w5,'z2w5.pol');
    return

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % z^2 = w^5 + (t + i s),  w = x(1) + i x(2), z = x(3) + i x(4)
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    function [f] = z2w5(n,x)
       t     = 1.0;
       s     = 1.0;
       % Parte real
       f(1) = x(1)^2-x(2)^2-x(3)^5-10.0*x(3)^3*x(4)^2-5.0*x(3)*x(4)^4-t;
       % Parte imaginária
       f(2) = 2.0*x(1)*x(2)-5.0*x(3)^4*x(4)-10.0*x(3)^2*x(4)^3-x(4)^5-s;
       return
    end
end
```

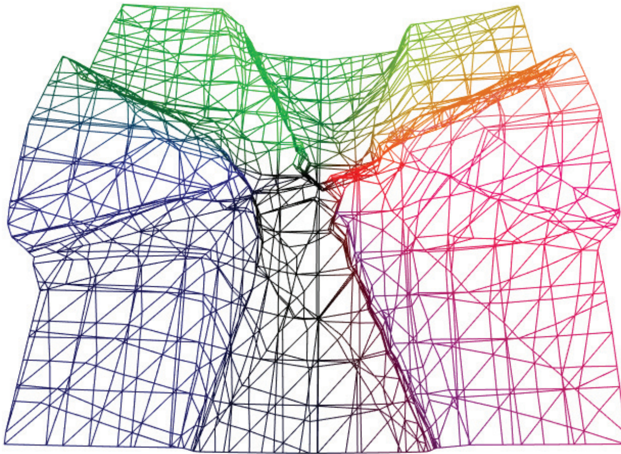

Figura 4.12: Projeção da aproximação de $z^2 = w^5 + 1 + i$.

4.9 Exercícios

1. Prove a regra de pivoteamento **Regra 1** de $K_1(g, \pi)$.
2. Prove a regra de pivoteamento **Regra 2** de $K_1(g, \pi)$.
3. Prove a regra de pivoteamento **Regra 3** de $K_1(g, \pi)$.
4. Prove a regra de pivoteamento **Regra 1** de $J_1(g, \pi)$.
5. Prove a regra de pivoteamento **Regra 2** de $J_1(g, \pi)$.
6. Prove a regra de pivoteamento **Regra 3** de $J_1(g, \pi)$.
7. Utilizando *MarchingSimplex* gere uma aproximação para a esfera S^4 de dimensão 4 em $\mathbb{R}^5$, definida por
$$x^2 + y^2 + z^2 + u^2 + v^2 = 1$$
8. Utilizando *MarchingSimplex* gere uma aproximação para o toro $S^2 \times S^1$ de dimensão 3 em $\mathbb{R}^5$, definido por
$$\begin{cases} x^2 + y^2 + z^2 & = 1 \\ u^2 + v^2 & = \frac{1}{4} \end{cases}$$
9. Pode-se representar uma superfície parametrizada
$$\begin{cases} x & = & f(s,t) \\ y & = & g(s,t) \\ z & = & h(s,t) \end{cases}$$
na forma implícita definida por $F : \mathbb{R}^5 \to \mathbb{R}^3$, com
$$\begin{cases} x - f(s,t) & = & 0 \\ y - g(s,t) & = & 0 \\ z - h(s,t) & = & 0 \end{cases}$$
A seguir desconsiderar as coordenadas s e t. Seguindo este procedimento, gere uma aproximação para a faixa de Möbius.
10. Com o mesmo procedimento do exercício anterior, gere uma aproximação para a Garrafa de Klein.
11. Utilizando *ContinuationSimplex* gere uma aproximação para a superfície em $\mathbb{C}^3$, definida por
$$u^2 + v^2 + w^2 = 1,$$
com $u, v, w \in \mathbb{C}$.
12. Utilizando *ContinuationSimplex* gere uma aproximação para a superfície em $\mathbb{C}^4$, definido por
$$\begin{cases} x^2 + y^2 & = 1 \\ u^2 + v^2 & = \frac{1}{4} \end{cases}$$
13. Utilizando *ContinuationSimplex* gere uma aproximação para cada variedade dos exercícios 7 a 10 e compare o tempo de execução.

4.10 Projetos

1. Modifique a geração de simplexos do programa *MarchingSimplex* para trocar a triangulação K_1 pela triangulação J_1.

2. Modifique a geração de simplexos do programa *MarchingSimplex* para trocar a triangulação K_1 pela triangulação definida por duas translações e uma reflexão em cada direção da triangulação $\mathcal{CFK}$ do hipercubo unitário I^n.

3. Modifique a geração de simplexos do programa *ContinuationSimplex* para trocar a triangulação K_1 pela triangulação J_1.

4. Modifique a geração de simplexos do programa *ContinuationSimplex* para trocar a triangulação K_1 pela triangulação definida por duas translações e uma reflexão em cada direção da triangulação $\mathcal{CFK}$ do hipercubo unitário I^n.

Capítulo 5

Modelagem Implícita

5.1 Combinação de Hipersuperfícies

Uma técnica muito usada em modelagem geométrica é a de se combinar objetos geométricos relativamente simples para se obter objetos geométricos mais complexos. Um exemplo desta técnica é utilizado no sistema de modelagem semelhante ao CSG (Construtive Solid Geometry, veja Requicha [50]) cujas operações para combinar primitivas geométricas vamos expor nesta seção.

Denominaremos hipersuperfície uma variedade definida implicitamente de dimensão $n-1$ em $\mathbb{R}^n$, isto é, quando a dimensão do contradomínio da função que a define é 1.

Sejam as aplicações $f_1, \ldots, f_k : \mathbb{R}^n \to \mathbb{R}$ tendo 1 como valor regular de $f_i, i = 1, \ldots, k$. Então, cada hipersuperfície $\mathcal{M}_i = f_i^{-1}(1)$ divide $\mathbb{R}^n$ em duas partes $\mathcal{M}_i^+ = \{x \in \mathbb{R}^n \mid f_i(x) > 1\}$ e $\mathcal{M}_i^- = \{x \in \mathbb{R}^n \mid f_i(x) < 1\}$, $i = 1, \ldots, k$.

O seguinte resultado pode ser visto em [55]: Sejam $f_{max}, f_{min} : \mathbb{R}^n \to \mathbb{R}$ dadas por $f_{max}(x) = max\{f_1(x), \ldots, f_k(x)\}$ e $f_{min}(x) = min\{f_1(x), \ldots, f_k(x)\}$. Então $\mathcal{M}_\cap = f_{max}^{-1}(1) = \partial(\mathcal{M}_1^- \cap \cdots \cap \mathcal{M}_k^-)$ e $\mathcal{M}_\cup = f_{min}^{-1}(1) = \partial(\mathcal{M}_1^- \cup \cdots \cup \mathcal{M}_k^-)$.

Uma observação importante é que $\mathcal{M}_\cup$ e $\mathcal{M}_\cap$ podem não ser variedades diferenciáveis. Mas se $\mathcal{M}_1, \ldots, \mathcal{M}_k$ se intercectam transversalmente duas a duas, então $\mathcal{M}_\cup$ e $\mathcal{M}_\cap$ são diferenciáveis, exceto, possivelmente, na união de um número finito de subvariedades de dimensão $n-2$.

Para evitar a falta de diferenciabilidade pode-se usar uma aproximação diferenciável para $\mathcal{M}_\cup$ e $\mathcal{M}_\cap$. Os dois resultados a seguir podem ser vistos em [55].

Se $f_1, \ldots, f_k : \mathbb{R}^n \to \mathbb{R}$ são aplicações diferenciáveis com $f_i(x) > 0$, $i = 1, \ldots, k$, então $\lim_{p\to\infty}(f_1^p + \cdots + f_k^p)^{\frac{1}{p}} = max\{f_1, \ldots, f_k\}$ e $\lim_{p\to\infty}(f_1^{-p} + \cdots + f_k^{-p})^{-\frac{1}{p}} = min\{f_1, \ldots, f_k\}$.

Sejam $f_{max,p}, f_{min,p} : \mathbb{R}^n \to \mathbb{R}$ dadas por $f_{max,p}(x) = (f_1^p(x) + \cdots + f_k^p(x))^{\frac{1}{p}}$ e $f_{min,p}(x) = (f_1^{-p}(x) + \cdots + f_k^{-p}(x))^{-\frac{1}{p}}$. Então, $f_{max,p}$ e $f_{min,p}$ são aproximações pontuais de classe C^1 de f_{max} e f_{min}, respectivamente.

Deste último resultado segue que, $\mathcal{M}_{\cap,p} = f_{max,p}^{-1}(1)$ é uma variedade diferenciável que aproxima $\mathcal{M}_\cap$, e $\mathcal{M}_{\cup,p} = f_{min,p}^{-1}(1)$ é uma variedade diferenciável que aproxima $\mathcal{M}_\cup$.

Para $f_1, f_2 : \mathbb{R}^2 \to \mathbb{R}$ dadas por $f_1(x, y) = x^2 + y^2$ e $f_2(x) = (x-1)^2 + y^2$, $\mathcal{M}_1, \mathcal{M}_1^+$ e $\mathcal{M}_1^-$ são mostrados na Figura 5.1 à esquerda e $\mathcal{M}_2, \mathcal{M}_2^+$ e $\mathcal{M}_2^-$ na Figura 5.1 à direita.

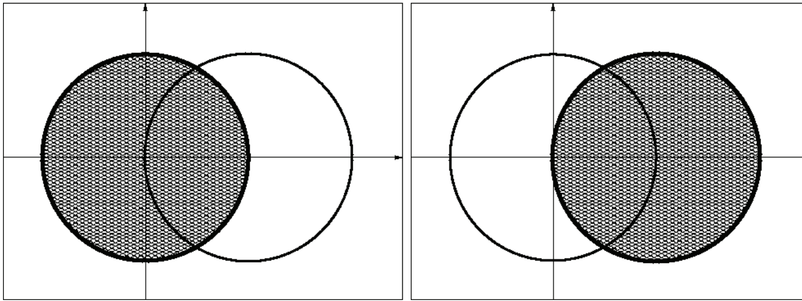

Figura 5.1: Esquerda: $\mathcal{M}_1, \mathcal{M}_1^+$ e $\mathcal{M}_1^-$; Direita: $\mathcal{M}_2, \mathcal{M}_2^+$ e $\mathcal{M}_2^-$.

$\mathcal{M}_\cup = f_{min}^{-1}(1) = \partial(\mathcal{M}_1^- \cup \mathcal{M}_2^-)$ é mostrado Figura 5.2 à direita e $\mathcal{M}_\cap = f_{max}^{-1}(1) = \partial(\mathcal{M}_1^- \cap \mathcal{M}_2^-)$ é mostrado Figura 5.2 à esquerda.

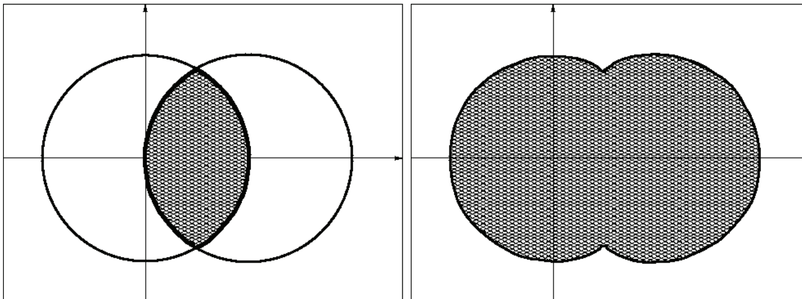

Figura 5.2: Esquerda: $\mathcal{M}_\cap = \partial(\mathcal{M}_1^- \cap \mathcal{M}_2^-)$; Direita: $\mathcal{M}_\cup = \partial(\mathcal{M}_1^- \cup \mathcal{M}_2^-)$.

Um exemplo utilizando o código *MarchingSimplex* ou *ContinuationSimplex* é dado pela união de duas esferas $f_1, f_2 : \mathbb{R}^3 \to \mathbb{R}$ dadas por $f_1(x,y,z) = x^2 + y^2 + z^2$ e $f_2(x,y,z) = (x-1)^2 + y^2 + z^2$. Então $\mathcal{M}_\cup = f_{min}^{-1}(1)$ onde $f_{min}(x,y,z) = min\{x^2 + y^2 + z^2, (x-1)^2 + y^2 + z^2\}$ é mostrado na Figura 5.3

Na Figura 5.4 temos a união de duas esferas aproximada por $\mathcal{M}_{\cup,p} = f_{min,p}^{-1}(1)$ com $f_{min,p}(x,y,z) = ((x^2 + y^2 + z^2)^{-p} + ((x-1)^2 + y^2 + z^2)^{-p})^{-\frac{1}{p}}$ para diversos valores de p.

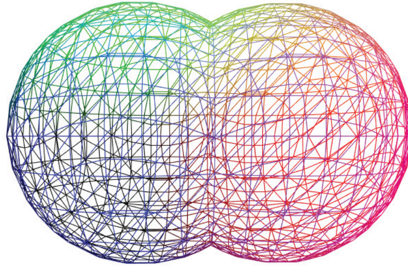

Figura 5.3: $\mathcal{M}_{\cup}$ de duas esferas.

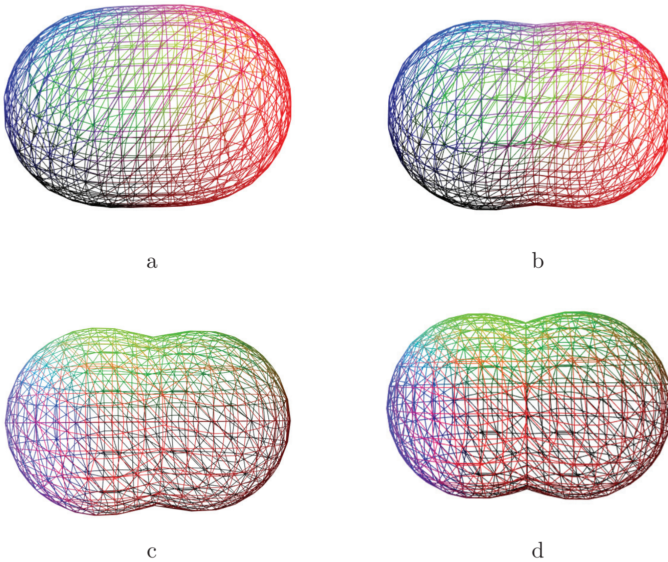

Figura 5.4: $\mathcal{M}_{\cup,p}$ de duas esferas para $p = 2$ (a), $p = 4$ (b), $p = 8$ (c) e $p = 16$ (d).

5.2 Exemplos

O seguinte programa usa *MarchingSimplex* para gerar a união de oito toros cujos centros estão ao longo de um ciclo. Neste caso:

$$f_{min,p}(x, y, z) = (\sum_{i=1}^{8} (t(i)^2 + z^2)^{-p})^{-1/p} - 1$$

tal que

$$t(i) = \sqrt{(x - x_0(i))^2 + (y - y_0(i))^2} - 2$$

$x_0(i)$ e $y_0(i)$ são os centros do toro $t(i)$, que definimos manualmente no código abaixo. Usamos $p = 10$.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Função oito_toros
%     Esta função retorna o esqueleto combinatório da aproximação
%     da união de oito toros utilizando a normna p, com p = 10,
%     utilizando o método Marching Simplex com a triangulação K_1.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function oito_toros
    % Dimensão do domínio
    n        = 3;
    % Dimensão do contradomínio
    k        = 1;
    % Ponto inicial da malha em R^n
    First    = [-9.1 -9.1 -2.1];
    % Ponto final da malha em R^n
    Last     = [9.1 9.1 2.1];
    % Número de divisões em cada dimensão da malha
    Division = [30 30 8];

    % Metodo Marching Simplex para a trinagulação K_1
    MarchingSimplex(n,k,First,Last,Division,@toro8_p,'toro8_10_MS.pol');

    rerturn

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % União de oito toros
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    function [f] = toro8_p(n,x)
       p    = 10.0;
       t(1) = sqrt((x(1)-6.0)*(x(1)-6.0)+(x(2)-0.0)*(x(2)-0.0))-2.0;
       t(2) = sqrt((x(1)+6.0)*(x(1)+6.0)+(x(2)-0.0)*(x(2)-0.0))-2.0;
       t(3) = sqrt((x(1)-0.0)*(x(1)-0.0)+(x(2)-6.0)*(x(2)-6.0))-2.0;
       t(4) = sqrt((x(1)-0.0)*(x(1)-0.0)+(x(2)+6.0)*(x(2)+6.0))-2.0;
       t(5) = sqrt((x(1)-4.0)*(x(1)-4.0)+(x(2)-4.0)*(x(2)-4.0))-2.0;
       t(6) = sqrt((x(1)-4.0)*(x(1)-4.0)+(x(2)+4.0)*(x(2)+4.0))-2.0;
       t(7) = sqrt((x(1)+4.0)*(x(1)+4.0)+(x(2)-4.0)*(x(2)-4.0))-2.0;
       t(8) = sqrt((x(1)+4.0)*(x(1)+4.0)+(x(2)+4.0)*(x(2)+4.0))-2.0;
       f(1) = 0.0;
       for i = 1:8
           f(1) = f(1) + (t(i)*t(i)+x(3)*x(3))^(-p);
       end
       f(1) = f(1)^(-1.0/p) - 1.0;
       return
    end
end
```

A Figura 5.5 representa a saída do código acima, escrita no arquivo *toro8_10_MS.pol*.

Figura 5.5: $\mathcal{M}_{\cup,p}$, para $p = 10$.

O próximo programa usa *MarchingSimplex* para gerar a união de 150 esferas cujos centros estão ao longo de um nó parametrizado.

Neste caso:

$$f_{min,p}(x,y,z) = (\sum_{i=1}^{m}((x-x_0(i))^2 + (y-y_0(i))^2 + (z-z_0(i))^2 - r^2 + 1)^{-p})^{-1/p} - 1$$

tal que

$$\begin{aligned} x_0(i) &= (1 + 0.4\, cos(\tfrac{4\pi i}{m}))\; cos(\tfrac{4\pi i}{m}), \\ y_0(i) &= (1 + 0.4\, cos(\tfrac{4\pi i}{m}))\; sen(\tfrac{4\pi i}{m}), \\ z_0(i) &= 0.5\; sen(\tfrac{4\pi i}{m}). \end{aligned}$$

Usamos $p = 30$, $r = 0.24$ e $m = 150$.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Função no_de_esferas
%     Esta função retorna o esqueleto combinatório da aproximação
%     da união de esferas com centro em um nó parametrizado,
%     utilizando a normna p, com p = 30.
%     Foi utilizando o método Marching Simplex com a triangulação K_1.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function no_de_esferas
```

```
% Dimensão do domínio
n        = 3;
% Dimensão do contradomínio
k        = 1;
% Ponto inicial da malha em R^n
First    = [-1.7 -1.7 -1.1];
% Ponto final da malha em R^n
Last     = [1.7 1.7 1.1];
% Número de divisões em cada dimensão da malha
Division = [28 28 22];

% Metodo Marching Simplex para a trinagulação K_1
MarchingSimplex(n, k, First, Last, Division, @no_p, 'no_30_MS.pol');

return

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Esfera com centro em x0 e raio r
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [f] = ESFERA(x,x0,r)
    f  = (x(1)-x0(1))*(x(1)-x0(1))+(x(2)-x0(2))*(x(2)-x0(2))
    +(x(3)-x0(3))*(x(3)-x0(3))-r*r;
    return
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Parametrização de um nó
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [x0] = PARAMETRIZACAO(t)
   x0(1) = (1.0+0.4*cos(1.5*t))*cos(t);
   x0(2) = (1.0+0.4*cos(1.5*t))*sin(t);
   x0(3) = 0.5*sin(1.5*t);
   return
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% União de esferas com centro no nó parametrizado e raio r
% utilizando a norma p, com p = 30.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [f] = no_p(n,x)
   p    = 30.0;
   m    = 150;
   r    = 0.24;
   f(1) = 0.0;
   for i = 0:m
      t    = 4.0*pi*i/m;
      [x0] = PARAMETRIZACAO(t);
      f(1) = f(1) + (ESFERA(x,x0,r)+1.0)^(-p);
   end
   f(1) = (f(1))^(-1.0/p)-1.0;
end
```

```
end
```

A Figura 5.6 representa a saída do código acima, escrita no arquivo *no_30_MS.pol*.

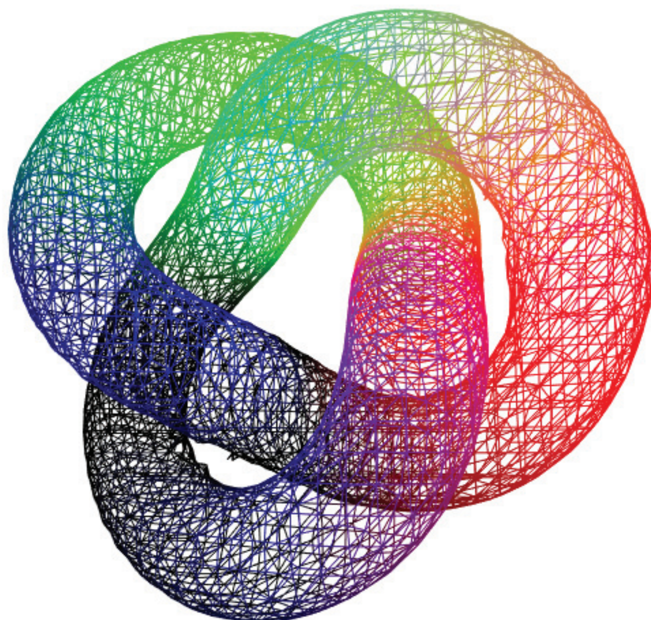

Figura 5.6: $\mathcal{M}_{\cup,p}$, para $p = 30$.

O último exemplo usa o programa *MarchingSimplex* para gerar a interseção de três esferas com centros equidistantes no plano (x, y) e mesmo raio:

$$\begin{cases} f_1(x,y,z) & = & x^2 + y^2 + z^2 - 1 \\ f_2(x,y,z) & = & x^2 + (y-1)^2 + z^2 - 1 \\ f_3(x,y,z) & = & (x - \frac{\sqrt{3}}{2})^2 + (y - \frac{1}{2})^2 + z^2 - 1, \end{cases}$$

assim, a interseção é dada pela função:

$$f_{max,p}(x,y,z) = ((f_1(x,y,z)+1)^p + (f_2(x,y,z)+1)^p + (f_3(x,y,z)+1)^p)^{1/p} - 1.$$

O código a seguir gera uma aproximação para a interseção das três esferas com $p = 10$.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Função intersecao_esferas
%     Esta função retorna o esqueleto combinatório da aproximação
%     da interseção de três esferas utilizando a normna p, com p = 10.
%     Foi utilizando o método Marching Simplex com a triangulação K_1.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function intersecao_esferas
    % Dimensão do domínio
    n        = 3;
    % Dimensão do contradomínio
    k        = 1;
    % Ponto inicial da malha em R^n
    First    = [-1.1 -1.1 -1.1];
    % Ponto final da malha em R^n
    Last     = [2.1 2.1 1.1];
    % Número de divisões em cada dimensão da malha
    Division = [30 30 20];

    % Metodo Marching Simplex para a trinagulação K_1
    MarchingSimplex(n,k,First,Last,Division,@intesf_p,'intesf_10_MS.pol');

    rerturn

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Interseção de três esferas utilizando a norma p, com p = 10.
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    function [f] = intesf_p(n,x)
       p      = 10.0;
       esf(1) = x(1)^2+x(2)^2+x(3)^2-1.0;
       esf(2) = x(1)^2+(x(2)-1.0)^2+x(3)^2-1.0;
       esf(3) = (x(1)-sqrt(3.0)/2.0)^2+(x(2)-0.5)^2+x(3)^2-1.0;
       f(1)   = ((esf(1)+1.0)^p+(esf(2)+1.0)^p+(esf(3)+1.0)^p)^(1.0/p)-1.0;
       return
    end
end
```

A Figura 5.7 representa a saída do código acima, escrita no arquivo *intesf_10_MS.pol*.

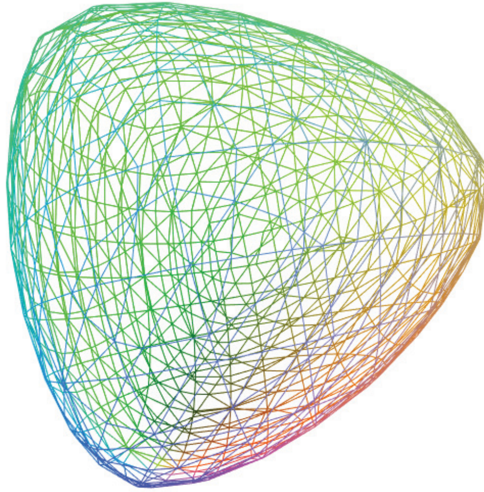

Figura 5.7: $\mathcal{M}_{\cap,p}$, para $p = 10$.

5.3 Exercícios

1. Sejam $f_{max}, f_{min} : \mathbb{R}^n \to \mathbb{R}$ dadas por $f_{max}(x) = max\{f_1(x), \ldots, f_k(x)\}$ e $f_{min}(x) = min\{f_1(x), \ldots, f_k(x)\}$. Mostre que $\mathcal{M}_\cap = f_{max}^{-1}(1) = \partial(\mathcal{M}_1^- \cap \cdots \cap \mathcal{M}_k^-)$ e $\mathcal{M}_\cup = f_{min}^{-1}(1) = \partial(\mathcal{M}_1^- \cup \cdots \cup \mathcal{M}_k^-)$.

2. Defina uma função que gere a diferença entre duas hipersuperfícies com interseção não nula.

3. Sejam $f_1, \ldots, f_k : \mathbb{R}^n \to \mathbb{R}$ aplicações diferenciáveis com $f_i(x) > 0$, $i = 1, \ldots, k$. Mostre que $\lim_{p\to\infty}(f_1^p + \cdots + f_k^p)^{\frac{1}{p}} = max\{f_1, \ldots, f_k\}$.

4. Sejam $f_1, \ldots, f_k : \mathbb{R}^n \to \mathbb{R}$ aplicações diferenciáveis com $f_i(x) > 0$, $i = 1, \ldots, k$. Mostre que $\lim_{p\to\infty}(f_1^{-p} + \cdots + f_k^{-p})^{-\frac{1}{p}} = min\{f_1, \ldots, f_k\}$.

5.4 Projetos

1. Utilize o código *MarchingSimplex* para aproximar um toro em $\mathbb{R}^3$ gerado a patir de uma união de esferas em $\mathbb{R}^3$.

2. Utilize o código *MarchingSimplex* para aproximar superfícies em $\mathbb{R}^3$ geradas a partir da união, interseção e diferença de superfícies mais simples tais como esferas, toros, cilindros, hiperbolóides em $\mathbb{R}^3$.

Bibliografia

[1] ALLGOWER, E.; GEORG, K. *Numerical Continuation Methods*. [S.l.]: Sringer Verlag, 1990.

[2] ALLGOWER E.; GEORG, K. Simplicial and continuation methods for approximating fixed points and solutions to systems of equations. *SIAM REVIEW*, v. 22, n. 1, 1980.

[3] BERNSTEIN, S. Sur la généralization du problème de dirichlet. *Math. Ann.*, v. 69, p. 82–136, 1910.

[4] BHANIRAMKA, P.; WENGER, R.; CRAWFIS, R. Isosurface construction in any dimension using convex hulls. *Transactions on Visualization and Computer Graphics*, v. 10, n. 2, p. 130–141, 2004.

[5] BLOOMENTHAL, J. Polygonization of implicit surfaces. *CAGD 5*, p. 341–355, 1988.

[6] CALABI, E. On kähler manifolds with vanishing canonical class. *Algebraic Geometry and Topology: A Symposium in Honor of Solomon Lefschetz*, p. 78–89, 1957.

[7] CALVES, L. F. G. et al. Four-dimensional ultrasonography of the fetal heart with spatiotemporal image correlation. *American Journal of Obstetrics & Gynecology (AJOG)*, v. 189, n. 6, p. 1792–1802, 2003.

[8] CANDELAS, P. et al. Vacuum configurations for superstrings. *Nuclear Physics B*, v. 258, p. 46–74, 1985.

[9] CASTELO, A. *Aproximações Adaptativas de Variedades Implícitas e Aplcações em Modelagem Implícita e Equações Algébrico-Diferenciais*. Tese (Doutorado) — PUC-Rio, Rio, 1992.

[10] CASTELO, A. et al. The j1a triangulation: an adaptive triangulation in any dimension. *Computers & Graphics*, v. 30, n. 5, p. 737–753, 2006.

[11] CHERNYAEV, E. V. Marching cubes 33: Construction of topologically correct isosurfaces. *Technical Report CERN CN 95-17*, 1995.

[12] CHOQUET-BRUHAT, Y.; DEWITT-MORETTE, C.; M., D.-B. *Analysis, Manifolds, and Physics*. [S.l.]: Gulf Professional Publishing, 1982.

[13] COXETER, H. S. M. Discrete groups generated by reflections. *Annals of Mathematics*, v. 35, p. 588–621, 1934.

[14] CREMMER, E.; JULIA, B.; SCHERK, J. Supergravity theory in 11 dimensions. *Physics Letters B*, v. 76, p. 409–412, 1978.

[15] DOI, A.; KOIDE, A. An efficient method of triangulating equi-valued surfaces by using tetrahedral cells. *Transactions on Information and Systems*, v. 1, p. 214–224, 1991.

[16] EAVES, B. C. Homotopies for computation of fixed points. *Mathematical Programming*, v. 3, n. 1, p. 1–22, 1972.

[17] EAVES, B. C. Homotopies for computation of fixed points on unbounded regions. *Mathematical Programming*, v. 3, n. 1, p. 225–237, 1972.

[18] EAVES, B. C. A short course in equations with pl homotopies. In: PUBLICATIONS, S. (Ed.). *Proc. SIAM - AMS*. [S.l.: s.n.], 1976. v. 9.

[19] EVEN, S. *Algorithmic Combinatorics*. [S.l.]: The Macmillan Company, 1973.

[20] FREITAS, S. R. *Aproximações Simpliciais de Variedades Implicitas e Equações Algébrico-Diferenciais*. Tese (Doutorado) — PUC-Rio, Rio, 1991.

[21] FREUDENTHAL, H. Simplizalzerlegungen von beschränkter flachheit. *Annals of Mathematics*, v. 43, p. 580–582, 1942.

[22] GARCIA C. B.; ZANGWILL, W. I. *The Flex Algorithm, Numerical Solution of Highly Nonlinear Problems*. [S.l.]: W. Forster (ed.) North - Holland Publishing Company, 1990.

[23] GLASSNER, A. S. *An Introduction to Ray Tracing*. [S.l.]: Academic Press, 1989.

[24] GREEN, M. B.; SCHWARZ, J. H. Anomaly cancellations in supersymmetric d = 10 gauge theory and superstring theory. *Physics Letters B*, v. 149, p. 117–122, 1984.

[25] GUCKENHEIMER, J.; VLADIMIRSKY, A. A fast method for approximating invariant manifolds. *SIAM Journal on Applied Dynamical Systems*, v. 3, n. 3, p. 232–260, 2004.

[26] GUÉZIEC, A.; HUMMEL, R. Exploiting triangulated surface extraction using tetrahedral decomposition. *Transactions on Visualization and Computer Graphics*, v. 1, n. 4, p. 328–342, 1995.

[27] HALL, M.; WARREN, J. Adaptive polygonalization of implicitly defined surfaces. *IEEE Computer Graphics and Applications*, v. 10, n. 6, p. 33–42, 1990.

[28] HENDERSON, M. E. Multiple parameter continuation: Computing implicitly defined k-manifolds. *International Journal of Bifurcation and Chaos*, v. 12, n. 3, p. 451–476, 2002.

[29] J., T.; V., S.; J., L. A global geometric framework for non-linear dimension reduction. *Science*, v. 290, n. 5500, p. 2319–2323, 2000.

[30] KALUZA, T. Zum unitätsproblem der physik. *Sitzungsberichte der Königlich Preubischen Akademie der Wissenschaften*, p. 966–972, 1921.

[31] KLEIN, F. Neue beiträge zur riemannchen funktionentheorie. *Math. Ann.*, v. 21, 1882–1883.

[32] KLEIN, O. Quantum theory and five-dimensional theory of relativity (in german and english). *Zeitschrift für Physik*, v. 37, n. 12, p. 895–906, 1926.

[33] KUHN, H. W. Simplicial approximation of fixed points. *Proceedings of National Academy of Sciences of United States of America*, v. 61, p. 1238–1242, 1968.

[34] LEWINER, T. et al. Efficient implementation of marching cubes' cases with topological guarantees. *Journal of Graphics Tools*, v. 8, n. 2, p. 1–15, 2003.

[35] LIMA, E. L. *Variedades Diferenciáveis*. [S.l.]: IMPA, CNPq, 1981.

[36] LIMA, E. L. *Curso de Análise, Vol 2*. [S.l.]: IMPA, CNPq, 2011.

[37] LORENSEN, W. E.; CLINE, H. E. Marching cubes: A high resolution 3d surface construction algorithm. *SIGGRAPH*, p. 163–169, 1987.

[38] MICHAEL, M. et al. 4d flow mri. *Journal of Magnetic Resonance Imaging (JRMI)*, v. 36, n. 5, p. 1015–1036, 2012.

[39] MILNOR, J. *Topology form the Differentiable Viewpoint*. [S.l.]: The University of Virginia Press, 1965.

[40] NIJENHUIS, A.; WILF, H. S. *Combinatorial Algorithms*. [S.l.]: Academic Press, 1978.

[41] NILSSON, B. E. W.; POPE, C. N. Hopf fibration of eleven-dimensional supergravity. *Supergravities in Diverse Dimensions*, p. 1478–1494, 1989.

[42] OHTAKE, Y. et al. Multi-level partition of unity implicits. *ACM Transactions on Graphics*, v. 22, n. 3, p. 463–470, 2003.

[43] O'NEILL, B. *Semi-Riemannian Geometry With Applications to Relativity*. [S.l.]: Academic Press, 1983.

[44] PAGE, E. S.; WILSON, L. B. *An Introduction to Computational Combinatorics*. [S.l.]: Cambridge Computer Science Text - 9, 1979.

[45] PAIVA, A.; LOPES, H.; LEWINER, T. Robust adaptive meshes for implicit surfaces. *19th Brazilian Symposium on Computer Graphics and Image Processing (SIBGRAPI)*, p. 205–212, 2006.

[46] PALIS-JR. J.; MELO, W. *Introdução aos Sistemas Dinâmicos*. [S.l.]: IMPA, CNPq, 1978.

[47] PAN, T. et al. 4d-ct imaging of a volume influenced by respiratory motion on multi-slice ct. *Medical Physics*, v. 31, n. 2, p. 333–340, 2004.

[48] PLANTINGA, S.; VEGTER, G. Isotopic approximation of implicit curves and surfaces. *ACM SIGGRAPH Symposium on Geometry Processing*, p. 245–254, 2004.

[49] POINCARÉ, H. Sur les courbes definé par une équation differencielle. *Gauthier-Villars, Paris*, 1881–1886.

[50] REQUICHA, A. A. G. Representations for rigid solids: Theory, methods and systems. *ACM Comp. Surveys*, v. 12, n. 4, p. 437–464, 1980.

[51] ROWEIS, S.; SAUL, L. Nonlinear dimensionality reduction by locally linear embedding. *Science*, v. 290, n. 5500, p. 2323–2326, 2000.

[52] SCARF, H. The approximation of fixed points of a continuous mapping. *SIAM Journal on Applied Mathematics*, v. 15, n. 5, p. 1328–1343, 1967.

[53] SHARF, A. et al. Space-time surface reconstruction using incompressible flow. *ACM Transaction on Graphics*, v. 27, n. 5, p. 110–120, 2008.

[54] SIMO, C. On the analytical and numerical approximation of invariant manifolds. *13. École de Printemps d'Astrophysique de Goutelas: Les méthodes modernes de la mécanique céleste*, p. 285–329, 1990.

[55] TAVARES G.; MIRANDA, J. Concordance operations for implicitly-defined manifolds. In: PUBLICATIONS, S. (Ed.). *Proc. SIAM Conference on Geometric Design*. [S.l.: s.n.], 1989.

[56] TREECE, G. M.; PRAGER, R. W.; GEE, A. H. Regularised marching tetrahedra: Improved iso-surface extraction. *Computers & Graphics*, v. 23, n. 4, p. 583–598, 1999.

[57] WEISS, K.; FLORIANI, L. Simplex and diamond hierarchies: Models and applications. *Computer Graphics Forum*, v. 30, n. 8, p. 2127–2155, 2011.

[58] WITTEN, E. String theory dynamics in various dimensions. *Nuclear Physics B*, v. 443, p. 85–126, 1995.

[59] YAU, S. Calabi's conjecture and some new results in algebraic geometry. *Proceedings of the National Academy of Sciences*, v. 74, n. 5, p. 1798–1799, 1977.

Apêndice A

Códigos Auxiliares

A.1 Os Vértices da Aproximação

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Calcula as coordenadas (Vertex) dos vértices da aproximação de M
% na face Face do simplexo s.
% Vert são as coordenadas dos k vértices de um simplexo s do domíno,
% rotulado por Face
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [Vertex trans] = GetVertexManifold(n,k,Face,Vert,FVert)
   % FVert eh a imagem de Vert para uma função Func
   % GetVertexManifold aproxima f por uma interpolação linear de FVert
   % e calcula o zero da aproximação
   for i = 1:k+1
      A(1,i) = 1;
      A(2:k+1,i) = FVert(Face(i)+1,:);
   end
   b = [1; zeros(k,1)];
   lamb = A\b;
   trans = 0;
   Vertex = zeros(1,n);
   % Se lamb >= 0, então o zero da aproximação esta no interior ou na
   % fronteira da face Face.
   % trans = 1 significa que a aproximação é transversal à face Face e
   % que Vertex será incluso na aproximação de M
   if lamb >= 0
       for i = 1:k+1
          Vertex = Vertex + lamb(i)*Vert(Face(i)+1,:);
       end
       trans = 1;
   end
   return
end
```

A.2 Perturbação

O código abaixo gera os vértices de todos os hipercubos em uma malha cartesiana com pontos inicial *First* e final *Last*, e número de divisões *Division* em $\mathbb{R}^n$ com $n = 4$. Observe que os vértices da malha foram pertubados utilizando uma distribuição normal para gerar um hipercubo de perturbações *Pert*, que é adicionado aos vértices dos hipercubos da malha depois de uma reflexão adequada de acordo com a posição do hipercubo na malha para preservar a continuidade. Exceto a parte da pertubação, o código abaixo é basicamente o código visto no Capítulo 3.

```
   % Dimensão do domínio
   n        = 4;
   % Ponto inicial do domínio
   First    = [0.0, 0.0, 0.0, 0.0];
   % Ponto final do domínio
   Last     = [1.0, 1.0, 1.0, 1.0];
   % Número de divisões da malha
   Division = [10, 10, 10, 10];
   % Delta: comprimento dos lados de cada célula da malha
   Delta = (Last-First)./Division;
   % Pert: função de pertubação com distribuição normal
   Pert  = random('Normal',0,1,2^n,n)*0.000001;
   % Base: base de cada dimensão da malha (ver definição de base no
   % cap. contagem e enumeração)
   Base  = InitGrid(n, Division);
   % Repete para cada célula g da malha
   for g = 0:Base(n+1)-1
      % Calcula os rótulos das coordenadas de g na malha a partir de
      % sua enumeração
      [Grid] = GridCoords(n,g,Base);
      % Calcula as coordenadas dos vértices de g e o valor da função
      % Func nesses vértices
      [Vert] = GenVert(n, Grid, Pert, First, Delta);
   end
   return


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Calcula as coordenadas (Grid) enumerada por i de uma malha de
% n dimensões, a partir da base de cada dimensão.
% Usa formula do cap. contagem e enumeração para obter Grid
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [Grid] = GridCoords(n,i,Base)
   copy = i;
   for j = n:-1:2
      aux     = mod(copy,Base(j));
      Grid(j) = (copy-aux)/Base(j);
      copy    = aux;
   end
   Grid(1) = copy;
   return
```

```
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Calcula as coordenadas Vert dos vértices de uma célula (hipercubo)
% da malha rotulada por Grid
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [Vert] = GenVert(n, Grid, Pert, First, Delta, Func)
   Vert  = [];
   % Repete para cada vértice i do hipercubo
   for i = 0:2^n-1
      % Calcula as coordenadas do vértice no hipercubo unitário
      % de dimensão n
      [Coords]    = HyperCubeCoords(n,i);
      % Perturba o vértice pela função Pert
      [CoordPert] = HyperCubePert(n,Grid,Coords,Pert);
      % Encontra a posição do vértice do hipercubo escalonado,
      % transladado e pertubado
      [VHC]       = HyperCube(n,First,Delta,Grid,Coords,CoordPert);
      Vert        = [Vert; VHC];
   end
   return
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Calcula as coordenadas (Coords) de um vértice enumerado por i em
% um hipercubo unitário de dimensão n.
% Usa fórmula do cap. contagem e enumeração para obter Coords
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [Coords] = HyperCubeCoords(n,i)
   copy = i;
   for j = 1:n-1
      Coords(j) = mod(copy,2);
      copy      = (copy-Coords(j))/2;
   end
   Coords(n) = copy;
   return
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Calcula as coordenadas perturbardas pela função Pert (CoordPert) de
% um vértice i com coordenadas Coords no hipercubo unitário de
% dimensão n
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [CoordPert] = HyperCubePert(n,Grid,Coords,Pert)
   pot = 1;
   label = 0;
   for i = 1:n
      % Reflete perturbação se coordenada de dimensão i do vértice
      % estiver em um hipercubo rotulado por Grid, tal que Grid(i)
      % seja ímpar
      p     = abs(Coords(i) - mod(Grid(i),2));
```

```
      label = label + pot*p;
      pot   = 2*pot;
   end
   CoordPert = Pert(label+1,:);
   return
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Calcula as coordenadas VHC de um vértice de um hipercubo, na
% malha de dimensão n
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [VHC] = HyperCube(n,First,Delta,I,Coords,Pert)
   % O vértice é transladado por First, escalonado por Delta, tem
   % coordenadas Coords no hipercubo unitário e é perturbado por Pert
   for i = 1:n
      VHC(i) = First(i) + (I(i)+Coords(i))*Delta(i) + Pert(i);
   end
   return
end
```

A.3 Regras de Pivoteamento

O código a seguir devolve o simplexo que é o i-ésimo pivoteado de um dado simplexo definido por *Grid* e P de acordo com as regras de pivoteamento definidos na seção 4.4.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Esta função devolve o simplexo que é o i-ésimo pivoteado de um
% dado simplexo definido por Grid e P.
% Entradas
%    n: dimensão do domínio
%    i: rótulo do vertice a ser pivoteado
%    Grid: posição do hipercubo na malha
%    P: permutação
%    Division: dimesões da malha
%    Base: base para gerar a malha
%    BaseP: base para gerar a permutação
% Saídas
%    g: rótulo da malha onde está o simplexo pivoteado
%    s: rótulo do simplexo pivoteado no hipercubo rotulado por g
%    value: se value = 1 o simplexo está no domínio e se
%           value = 0 o simplexo não está no domínio
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [value g s] = Pivoting(n, i, Grid, P, Division, Base, BaseP)
    % Vértice 0
    if i == 0
        if Grid(P(1)) < Division(P(1))-1
            % Pivoteado no domínio
            % De acordo com a regra 2 de pivoteamento da i
            %   triangulação K1
```

```
            G       = Grid;
            G(P(1)) = G(P(1)) + 1;
            g       = Get_Label_Prod(n,Base,G);
            Q       = [P(2:n) P(1)];
            s       = Get_Label_Perm(n,BaseP,Q);
            value   = 1;
            return
        else
            % Pivoteado fora do domínio
            g       = 0;
            s       = 0;
            value   = 0;
            return
        end
    end
    % Vértice n
    if i == n
        if Grid(P(n)) > 0
            % Pivoteado no domínio
            % De acordo com a regra 3 de pivoteamento da i
            %   triangulação K1
            G       = Grid;
            G(P(n)) = G(P(n)) - 1;
            g       = Get_Label_Prod(n,Base,G);
            Q       = [P(n) P(1:n-1)];
            s       = Get_Label_Perm(n,BaseP,Q);
            value   = 1;
            return
        else
            % Pivoteado fora do domínio
            g       = 0;
            s       = 0;
            value   = 0;
            return
        end
    end
    % Vértice i
    % Pivoteado no domínio
    % De acordo com a regra 1 de pivoteamento da triangulação K1
    g     = Get_Label_Prod(n,Base,Grid);
    Q     = [P(1:i-1) P(i+1) P(i) P(i+2:n)];
    s     = Get_Label_Perm(n,BaseP,Q);
    value = 1;
    return
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Esta função retorna o vertice de Simp que não está em Face,
% isto é, o vétice a ser pivoteado
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [i] = GetPivotSimplex(n, Simp, Face)
```

```
    for i = 1:n
        vertex = Simp(i);
        % verifica se vertex esta em Face
        [k]    = Include(1,vertex,n-1,Face);
        if k == 0
            % Retorna o vertice i a ser pivoteado
            return
        end
    end
    % Nenhum vertice de Simp está em Face
    i = 0;
    return
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Verifica se face y de dimensão m-1 contém face x de dimensão n-1
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [k] = Include(n,x,m,y)
   % k é o número de rótulos que estão em x e y ao mesmo tempo.
   % Se k = n, entao y contém x
   k = 0;
   for i = 1:n
       j = i;
       % Faz varredura supondo que os rótulos de x e y estão em
       % ordem lexicográfica
       while (j < m) && (y(j) < x(i))
           j = j+1;
       end
       if y(j) == x(i)
           k = k+1;
       end
   end
   return
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Gera o rótulo do produto cartesiano dado por f
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [label] = Get_Label_Prod(n, Base, f)
    % Cálculo do rotulo
    label = 0;
    for i = 1:n
        label = label + f(i)*Base(i);
    end
    return
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Gera o rótulo da permutação p
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [label] = Get_Label_Perm(n, Base, p)
```

```
    % Gera o mapeamento inverso da permutação para produto cartesiano
    [f]     = Map_Inv_Perm(n,p);
    % Gera o rótulo do produto cartesiano
    [label] = Get_Label_Prod(n,Base,f);
    % Inverte a ordem do rótulo
    label   = Base(n+1)-1 - label;
    return
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Gera o mapeamento inverso da permutação para produto cartesiano
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [f] = Map_Inv_Perm(n,p)
    % Copia p em q
    q = p;
    for i = n:-1:2
        % Encontra a posição j em x que está i
        [j]  = Find(i,i,q);
        % Define o valor da posição i do produto cartesiamo
        f(i) = j;
        % Retira a j-ésima coordenada de q
        q    = [q(1:j-1) q(j+1:i)];
    end
    % Define a primeira posição
    f(1) = 1;
    % Subtrai 1 de todas as coordenadas para que iniciem com 0
    f    = f - 1;
    return
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Encontra a posição j em x que está i
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [j] = Find(n,i,x)
    j = 0;
    for k = 1:n
        if x(k) == i
            j = k;
            return
        end
    end
    return
end
```

A.4 Esqueleto Combinatório

O código a seguir gera o esquleto combinatório de um politopo (aproximação linear da variedade $\mathcal{M}_\sigma = \sigma \cap \mathcal{M}$ em um simplexo σ).

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% Gera o esqueleto combinatório
% Entrada:
%    n: dimensão do domínio
%    Simp: rótulo do simplexo
%    k: dimensão do contradomínio
%    nVertex: número de vértices da aproximação da variedade
%    FaceVertex: rótulo das faces de dimensão k que contém os
%    vértices da aproximação da variedade
% Saída:
%    nSkel: número de faces de cada dimensão da aproximação
%    Skel: rótulos das faces dos simplexos que contém as faces de
%    cada dimensão da aproximação
%    AdjSkel: relação de adjacência das faces da aproximação
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [nSkel Skel AdjSkel] = Skeleton(n, Simp, k, nVertex, FaceVertex)
    % nSkel(i): número de faces da aproximação com dimensão i-1
    nSkel(1) = nVertex;
    % Skel{i}: rótulo das faces da aproximação com dimensão i-1
    Skel{1}  = FaceVertex;

    % Repete para cada dimensão s entre k+1 e n
    for s = k+1:n
        % Gera os rótulos das faces da aproximação de dimensão s
        for i = 1:s+1
            DivisionC(i) = n-s+i;
        end
        [BaseC]      = InitGrid(s+1,DivisionC);
        FacesSimp    = [];
        FacesAdj     = [];
        nSkel(s-k+1) = 0;

        % Repete para cada combinação j de vértices
        for j = 0:BaseC(s+2)-1
          % Gera as faces de dimensão s do simplexo a partir da
          % enumeração j
          [C]   = GridCoords(s+1,j,BaseC);
          C     = C+1;
          [lex] = Lexico(s+1,C);

          % Verifica unicidade de C
          if lex == 1
              for i = 1:s+1
                  % Rotula Face em função dos vértices
                  Face(i) = Simp(C(i));
              end
              cont = 0;
              Adj  = [];

              % Para cada face do esqueleto combinatório
              for i = 1:nSkel(s-k)
                  % Verifica se a face Face de dimensão s
```

```
                  % contém face de dimensão s-1 no esqueto combinatório
                  [np] = Include(s,Skel{s-k}(i,:),s+1,Face);
                  % Se positivo, atualiza adjacência
                  if np == s
                      cont      = cont+1;
                      Adj(cont) = i;
                  end
              end

              % Se Face contém alguma face de dimensão s-1 na saída:
              if cont > 0
                 % Atualiza número de faces de dimensão s-k
                 nSkel(s-k+1)           = nSkel(s-k+1)+1;
                 % Inclui os rótulos dos vértices de Face na saída
                 FacesSimp              = [FacesSimp; Face];
                 % Inclui faces de dimensão anterior que estão
                 % adjacentes a Face
                 FacesAdj{nSkel(s-k+1)} = Adj;
              end
           end
        end

        % Atualiza lista de faces da aproximação e suas faces adjacentes
        Skel{s-k+1}  = FacesSimp;
        AdjSkel{s-k} = FacesAdj;
    end
    return
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Verifica se face y de dimensão m-1 contém face x de dimensão n-1
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [k] = Include(n,x,m,y)
   % k é o número de rótulos que estão em x e y ao mesmo tempo.
   % Se k = n, entao y contém x
   k = 0;
   for i = 1:n
       j = i;
       % Faz varredura supondo que os rótulos de x e y estão em
       % ordem lexicográfica
       while (j < m) && (y(j) < x(i))
           j = j+1;
       end
       if y(j) == x(i)
           k = k+1;
       end
   end
   return
end
```

A.5 Condição Inicial

A seguir apresentamos um código que encontra um simplexo definido pelos rótulos g do hipercubo e s do simplexo que contém o ponto inicial *FirstPoint*. Parte do código que foi descrito nas seções 4.4 do capítulo 2 não será repetida.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Função GetFirstSimplex
%     Esta função percorre os simplexos da triangulação K_1
%     de acordo com o método da seção 5.2 do capítulo 5 para
%     encontrar o simplexo da triangulação que contenha o ponto
%     inicial.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Entradas:
%     n: Dimensão do domínio
%     k: Dimensão do contradomínio
%     Fisrt: Ponto inicial da malha em R^n
%     Last: Ponto final da malha em R^n
%     Division: número de divisões em cada dimensão da malha
%     FirstPoint: Ponto inicial da variedade
%     Func: Função que define a variedade implícita
% Saídas:
%     g: rótulo do hipercubo que contem o ponto inicial
%     s: rótulo do simplexo no hipercubo com rótulo g que contem
%        o ponto inicial
%     value: retorno de sucesso ou não da busca
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [g s value] = GetFirstSimplex(n,k,First,Last,Division,FirstPoint,Func)
   % Tolerancia para o erro do ponto inicial pertencer a variedade
   tol  = 0.000001;
   % Calculando a função no ponto inicial
   [F]  = Func(n,FirstPoint);
   % Verificando se o ponto pertence a variedade
   if norm(F(1:k),2) > tol
       % Ponto inicial longe da variedade, retorne
       g     = -1;
       s     = -1;
       value = -1;
       fprintf('Ponto nao pertence a variedade\n');
       return
   end
   % Verificando se o ponto inicial pertence ao domínio
   for i = 1:n
        if (FirstPoint(i) < First(i)) || (FirstPoint(i) > Last(i))
            % Ponto inicial fora do domínio, retorne
            g     = -1;
            s     = -1;
            value = -2;
            fprintf('Ponto fora do dominio\n');
            return
        end
```

```
end
% Calculando o espaçamento da malha
Delta = (Last-First)./Division;
% Criando o hipercubo de perturbações
Pert  = random('Normal',0,1,2^n,n)*tol;
% Calculando uma base para a malha
Base  = InitGrid(n, Division);
% Calculando uma base para as permutações
for i = 1:n
     DivisionP(i) = i;
end
[BaseP] = InitGrid(n,DivisionP);
% Calculando as coordenadas do hipercubo mais próximo do
% ponto inicial
Grid    = fix((FirstPoint-First)./Delta);
% Calculando o rótulo do hipercubo
g       = Get_Label_Prod(n,Base,Grid);
% Inicializando o rótulo do simplexo como o primeiro
s       = 0;
% Inicializando o rótulo do último hipercubo
gold    = -1;
% Repete enquanto não encontrou o hipercubo e o simplexo contendo
% o ponto inicial
while g > 0
     % Imprimindo o rótulo do hipercubo e do simplexo
     fprintf('Simplexo: g = %d  s = %d\n',g,s);
     % Verificando se o hipercubo é o mesmo que o último calculado
     if g ~= gold
         % Gerando as coordenadas do hipercubo
         [Grid]       = GridCoords(n,g,Base);
         % Calculnado os vértices e a função avalida nos vértices do
         % hipercubo
         [Vert FVert] = GenVert(n, Grid, Pert, First, Delta, Func);
     end
     % Calculando a permutação que gera o simplexo
     [P]      = Get_Perm(n,BaseP,s);
     % Calculando as coordenadas dos vértices do simplexo a partir
     % da permutação
     [Simp]  = GenLabelSimplex(n,P);
     % Calculando as coordenadas baricêntricas do ponto inicial com
     % respeito a este simplexo e verificando se está no simplexo ou
     % retornando o melhor vértice a ser pivoteado para chegar no
     % ponto inicial
     [trans] = GetBestSimplex(n,Simp,Vert,FirstPoint);
     if trans < 0
         % Ponto inicial no simplexo, retorne sucesso
         value = 1;
         return
     end
     % Calculando o rótulo do hipercubo e do simplexo pivoteado
     gold = g;
```

```
        [value g1 s1] = Pivoting(n,trans,Grid,P,Division,Base,BaseP);
        % Verificando se o hipercubo está no domínio
        if value == 0
            % Hipercubo fora do domínio
            g  = -1;
        else
            % Hipercubo e simplexo no domínio
            g  = g1;
            s  = s1;
        end
   end
   % Ponto inicial fora do domínio, retorne fracasso
   g     = -1;
   s     = -1;
   value = -2;
   fprintf('Ponto fora do dominio\n');
   return
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Função GetBestSimplex
%     Caso o ponto inicial esteja dentro do simplexo, retorna êxito,
%     caso contrário encontra o melhor simplexo (pivoteado do simplexo
%     de entrada) que define o caminho até o ponto inicial.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Entradas:
%     n: Dimensão do domínio
%     Simp: rótulos dos vértices do simplexo
%     Vert: vértices do hipercubo
%     FirstPoint: Ponto inicial
% Saídas:
%     trans: retorno de sucesso caso o ponto inicial esteja
%            no simplexo e caso contrário o melhor vértice
%            a ser pivoteado para chegar ao ponto inicial
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [trans] = GetBestSimplex(n,Simp,Vert,FirstPoint)
   % Montando a matriz
   for i = 1:n+1
      A(1,i) = 1;
      A(2:n+1,i) = Vert(Simp(i)+1,:);
   end
   % Montando o vetor independente
   b = [1; FirstPoint(1:n)'];
   % Calculando as coordenadas baricêntricas
   lamb = A\b;
   % Verificando as coordenadas baricêntricas
   trans = 0;
   if lamb >= 0.0
       % Coordenadas todas positivas, ponto no simplexo
       trans = -1;
   else
```

```
       % Alguma coordenada negativa, ponto fora do simplexo
       % Calculando a coordenada mais negativa
       min  = lamb(1);
       imin = 1;
       for i = 1:n+1
           if lamb(i) < min
               min  = lamb(i);
               imin = i;
           end
       end
       % retornando o vértice a ser pivoteado
       trans = imin-1;
   end
   return
end
```

A.6 Estrutura de Dados

A seguir apresentamos o código de pesquisa, inserção e remoção de elementos de uma lista armazenada na forma de um vetor cujos elementos possuem duas coordenadas (g, s) correspondendo ao rótulo do hipercubo e do simplexo no hipercubo.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Função InList
%     Esta função verifica se um elemento (g,s) está na lista.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Entradas:
%     NList: Número de elementos da lista
%     List: Lista
%     g: rótulo de um hipercubo da malha
%     s: rótulo de um simplexo no hipercubo com rótulo g
% Saídas:
%     value: retorna 0 se não está na lista e 1 se está na lista
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [value] = InList(NList, List, g, s)
    % Não está na lista é o usual
    value = 0;
    % Se a lista não é vazia
    if NList > 0
        % Percorrendo os elementos da lista
        for i = 1:NList
            % Se g é menor ou igual ao elemento da lista
            if List(i,1) >= g
                % Se g está na lista e s é menor ou igual ao
                % elemento da lista
                if (List(i,1) == g) && (List(i,2) >= s)
                    % Se s menor que o elemento da lista
                    if List(i,2) > s
                        % Não está na lista
                        return
```

```
                    else
                        % Está na lista
                        value = 1;
                        return
                    end
                else
                    % Se g menor que o elemento da lista
                    if List(i,1) > g
                        % Não está na lista
                        return
                    end
                end
            end
        end
    end
    return
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Função GetAndRemoveList
%     Esta função retorna um elemento da lista e remove este
%     elemento.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Entradas:
%     NList: Número de elementos da lista
%     List: Lista
%     g: rótulo de um hipercubo da malha
% Saídas:
%     g: rótulo de um hipercubo da malha
%     s: rótulo de um simplexo no hipercubo com rótulo g
%     NList: Número de elementos da lista atualizado
%     List: Lista atualizada
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [g s NList List] = GetAndRemoveList(NList, g, List)
    % Se a lista é vazia
    if NList == 0
        % retorne que não há este elemento na lista
        g  = -1;
        s  = -1;
        return
    end
    % Percorre a lista
    for i = 1:NList
        % Se o rótulo do hipercubo g está na lista
        if List(i,1) == g
            % Retorne (g s) e remova da lista
            s     = List(i,2);
            List  = [List(1:i-1,:); List(i+1:NList,:)];
            NList = NList - 1;
            return
        end
```

```
    end
    % Se g não está na lista, retorne (g s) do primeiro
    % elemento da lista
    g     = List(1,1);
    s     = List(1,2);
    List  = List(2:NList,:);
    NList = NList - 1;
    return
end


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Função InsertList
%     Esta função insere em uma list um elemento (g,s) composto
%     por um hipercubo com rótulo g e um simplexo neste hipercubo
%     com rótulo s.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Entradas:
%     NList: Número de elementos da lista
%     List: Lista
%     g: rótulo de um hipercubo da malha
%     s: rótulo de um simplexo no hipercubo com rótulo g
% Saídas:
%     NList: Número de elementos da lista atualizado
%     List: Lista atualizada
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [NList List] = InsertList(NList, List, g, s)
    % Se a lista não é vazia
    if NList > 0
        % Percorre a lista
        for i = 1:NList
            % Se o rótulo do hipercubo for menor que o rótulo do
            % hipercubo da lista
            if List(i,1) >= g
                % Se o rótulo do hipercubo for igual ao rótulo do
                % hipercubo da lista e o rótulo do simplexo for menor
                % que o rótulo do simplexo da lista
                if (List(i,1) == g) && (List(i,2) >= s)
                    % Se o rótulo do simplexo for menor que o da lista
                    if List(i,2) > s
                        % Insere (g s) na lista e retorna
                        List  = [List(1:i-1,:); g s; List(i:NList,:)];
                        NList = NList + 1;
                        return
                    else
                        % (g s) já está na lista, retorne
                        return
                    end
                else
                    % Se o rótulo do hipercubo for menor que o da lista
                    if List(i,1) > g
```

```
                        % Insere (g s) na lista, retorne
                        List  = [List(1:i-1,:); g s; List(i:NList,:)];
                        NList = NList + 1;
                        return
                    end
                end
            end
        end
    end
    % Lista vazia, insere (g s) na lista
    List  = [List; g s];
    NList = NList + 1;
    return
end
```

Apêndice B

Código Marching Simplex

Segue abaixo o código inteiro do programa, com comentários.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Função MarchingSimplex
%     Esta função percorre todos os simplexos da triangulação K_1
%     de R^n para obter uma aproximação de uma variedade definida
%     implicitamente, definida pela função Func: R^n -> R^k.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Entradas:
%     n: Dimensão do domínio
%     k: Dimensão do contradomínio
%     First: Ponto inicial da malha em R^n
%     Last: Ponto final da malha em R^n
%     Division: número de divisões em cada dimensão da malha
%     Func: Função que define a variedade implícita
%     filename: Nome do arquivo de saída em que será escrito o
%               esqueleto combinatório
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function MarchingSimplex(n,k,First,Last,Division,Func,filename)
   % Abre arquivo para escrita do esqueleto combinatrio e escreve
   % parâmetros de entrada
   file  = fopen(filename,'w');
   fprintf(file,'%3d %3d\n',n,k);
   fprintf(file,'%3d ',Division(1:n));
   fprintf(file,'\n\n');

   % Delta: comprimento dos lados de cada célula da malha
   Delta = (Last-First)./Division;
   % Pert: função de pertubação com distribuição normal
   Pert  = random('Normal',0,1,2^n,n)*0.000001;
   % Base: base de cada dimensão da malha (ver definição de base no
   % cap. contagem e enumeração)
   Base  = InitGrid(n, Division);

   % BaseP: base de permutações
   for i = 1:n
        DivisionP(i) = i;
```

```
end
[BaseP] = InitGrid(n,DivisionP);

% BaseC: base de combinações
for i = 1:k+1
     DivisionC(i) = n-k+i;
end
[BaseC] = InitGrid(k+1,DivisionC);

% Repete para cada célula g da malha
for g = 0:Base(n+1)-1
   % Calcula os rótulos das coordenadas de g na malha a partir de
   % sua enumeração
   [Grid] = GridCoords(n,g,Base);
   % Calcula as coordenadas dos vértices de g e o valor da função
   % Func nesses vértices
   [Vert FVert] = GenVert(n, Grid, Pert, First, Delta, Func);

   % Repete para cada simplexo s de g
   for s = 0:BaseP(n+1)-1
     fprintf('Simplexo: g = %d  s = %d\n',g,s);
     % Calcula n-upla f a partir da enumeração de s em g
     [f] = GridCoords(n,s,BaseP);
     % Mapeia f em uma permutação P
     f = f+1;
     [P] = Map_Perm(n,f);
     % Encontra os rótulos dos vértices de s a partir de P
     [Simp] = GenLabelSimplex(n,P);
     % VertexManifold: coordenadas dos vértices da aproximação
     % de M em s
     VertexManifold = [];
     % FaceVertex: rótulo das células da aproximação de M
     FaceVertex     = [];
     % NumberVertex: número de faces de s na aproximação
     NumberVertex   = 0;

     % Repete para cada face formada pela combinação de
     % k+1 vértices de s
     for j = 0:BaseC(k+2)-1
       % Calcula combinação C a partir da enumeração j
       [C]   = GridCoords(k+1,j,BaseC);
       C     = C+1;
       [lex] = Lexico(k+1,C);

       % Verifica unicidade de C (desconsidera-se todas permutações,
       % exceto uma)
       if lex == 1
           % Rotula a face em função dos rótulos dos vértices de s
           for i = 1:k+1
               Face(i) = Simp(C(i));
           end
```

```
            % Calcula as coordenadas dos vértices da aproximação (Vertex)
            [Vertex trans] = GetVertexManifold(n,k,Face,Vert,FVert);

            % Inclui Vertex na aproximação de M, assim como os rótulos
            % de Face, caso a variedade seja transversal ao simplexo
            if trans == 1
               VertexManifold = [VertexManifold; Vertex];
               FaceVertex     = [FaceVertex; Face];
               NumberVertex = NumberVertex + 1;
            end
        end
      end

      % Calcula o esqueleto combinatório da aproximação no simplexo s
      if NumberVertex > 0
         [nSkel Skel AdjSkel] = Skeleton(n,Simp,k,NumberVertex,FaceVertex);
         % Escreve a enumeração da célula g e os rótulos de suas
         % coordenadas na malha
         fprintf(file,'%3d ',g);
         fprintf(file,'%3d ',Grid);
         fprintf(file,'\n');
         fprintf(file,'  1\n');
         % Escreve o número de vértices de uma célula da aproximação
         fprintf(file,'%3d \n',nSkel(1));
         % Escreve os rótulos dos vértices extremos das arestas
         % da triangulação de entrada que contém vértices da
         % aproximação e as coordenadas desses vértices da aproximação
         for i = 1:nSkel(1)
            fprintf(file,'%3d ',Skel{1}(i,:));
            fprintf(file,'%15.8f ',VertexManifold(i,:));
            fprintf(file,'\n');
         end

         % Escreve a dimensão de cada face da aproximação, e o rótulo de
         % suas facetas
         for j = 1:n-k
            fprintf(file,'%3d\n',nSkel(j+1));
            for i = 1:nSkel(j+1)
               fprintf(file,'%3d ',AdjSkel{j}{i}(:));
               fprintf(file,'\n');
            end
         end
         fprintf(file,'\n');
       end
    end
  end
  fprintf(file,'-1\n');
  fclose(file);
  return
end
```

Apêndice C

Código Continuation Simplex

Serão apresentadas apenas as funções que ainda não foram descritas nas seções e capítulos anteriores.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Função ContinuationSimplex
%     Esta função percorre os simplexos da triangulação K_1
%     de acordo com o método da seção 5.2 do capítulo 5 para
%     encontrar o simplexo da triangulação que contenha o ponto
%     inicial.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Entradas:
%     n: Dimensão do domínio
%     k: Dimensão do contradomínio
%     Fisrt: Ponto inicial da malha em R^n
%     Last: Ponto final da malha em R^n
%     Division: número de divisões em cada dimensão da malha
%     FirstPoint: Ponto inicial da variedade
%     Func: Função que define a variedade implícita
%     filename: nome do arquivo que será escrito o esqueleto
%               combinatório
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function ContinuationSimplex(n,k,First,Last,Division,FirstPoint,
                             Func,filename)
   % Abrindo o arquivo de saida de dados que será escrito o
   % esqueleto combinatório e escrevendo as dimensões do domínio,
   % contra-domínio e número de divisões da malha
   file  = fopen(filename,'w');
   fprintf(file,'%3d %3d\n',n,k);
   fprintf(file,'%3d ',Division(1:n));
   fprintf(file,'\n\n');

   % Calculando o espaçamento da malha
   Delta = (Last-First)./Division;
   % Gerando o hipercubo de perturbações
   Pert  = random('Normal',0,1,2^n,n)*0.000001;
   % Gerando a base para a malha
```

```
Base  = InitGrid(n, Division);
% Gerando a base para as permutações
for i = 1:n
     DivisionP(i) = i;
end
[BaseP] = InitGrid(n,DivisionP);
% Gerando a base para as combinações
for i = 1:k+1
     DivisionC(i) = n-k+i;
end
[BaseC] = InitGrid(k+1,DivisionC);

% Inicializando as listas vazias
NSimpTrans = 0;
SimpTrans  = [];
NSimpProc  = 0;
SimpProc   = [];

% Encontrando o primeiro hipercubo com rótulo g e simplexo com
% rótulo s
[g s value] = GetFirstSimplex(n,k,First,Last,Division,FirstPoint,Func);
% Analizando se houve falha
if value < 0
    % Falha da pesquisa pelo simplexo inicial
    fprintf('Nenhum simplexo transversal\n');
    return
end

% Inserindo (g,s) na lista de simplexo transversais
% não processados
[NSimpTrans SimpTrans] = InsertList(NSimpTrans,SimpTrans,g,s);
gold                   = -1;

% Percorrendo a lista de simplexos transversais não processados
while NSimpTrans > 0
     % Removendo um simplexo transversal não processado da lista
     [g s NSimpTrans SimpTrans] = GetAndRemoveList(NSimpTrans,g,SimpTrans);
     % Inserindo (g,s) na lista de simplexo transversais
     % já processados
     [NSimpProc SimpProc]       = InsertList(NSimpProc,SimpProc,g,s);
     fprintf('Simplexo: g = %d  s = %d  N = %d\n',g,s,NSimpTrans);
     % Verificando se o simplexo pertence ao mesmo hipercubo do
     % último simplexo processado
     if g ~= gold
         % Novo hipercubo, gerando os vértices e a função avaliada nos
         % vértoces deste hipercubo
         [Grid]       = GridCoords(n,g,Base);
         [Vert FVert] = GenVert(n, Grid, Pert, First, Delta, Func);
     end
     % Gerando a permutação com rótulo s
     [P]              = Get_Perm(n,BaseP,s);
```

```
% Gerando os rótulos dos vértices do simplexo a partir da permutação
[Simp]           = GenLabelSimplex(n,P);
% Inicializando o esquelto combinatório deste simplexo
NumberVertex     = 0;
VertexManifold   = [];
FaceVertex       = [];
% Gerando os vérices do esqueleto combinatório deste simplexo
% percorrendo as faces de dimensão k
for j = 0:BaseC(k+2)-1
    % Calculando a combinação
    [C]   = GridCoords(k+1,j,BaseC);
    C     = C+1;
    % Verificando se é lexicográfica
    [lex] = Lexico(k+1,C);
    if lex == 1
        % Calculando os rótulos dos vértices da face
        for i = 1:k+1
            Face(i) = Simp(C(i));
        end
        % Verificando se a face é transversal e caso sim
        % armazenado o véritce do esqueleto combinatório
        [Vertex trans] = GetVertexManifold(n,k,Face,Vert,FVert);
        if trans == 1
            VertexManifold = [VertexManifold; Vertex];
            FaceVertex     = [FaceVertex; Face];
            NumberVertex   = NumberVertex + 1;
        end
    end
end
% Se o simplexo é transversal
if NumberVertex > 0
   % Gerando o esqueleto combinatório
   [nSkel Skel AdjSkel] = Skeleton(n,Simp,k,NumberVertex,FaceVertex);
   % Escrevendo o esqueleto combinatório
   fprintf(file,'%3d ',g);
   fprintf(file,'%3d ',Grid);
   fprintf(file,'\n');
   fprintf(file,'  1\n');
   fprintf(file,'%3d \n',nSkel(1));
   for i = 1:nSkel(1)
      fprintf(file,'%3d ',Skel{1}(i,:));
      fprintf(file,'%15.8f ',VertexManifold(i,:));
      fprintf(file,'\n');
   end
   for j = 1:n-k
      fprintf(file,'%3d\n',nSkel(j+1));
      for i = 1:nSkel(j+1)
         fprintf(file,'%3d ',AdjSkel{j}{i}(:));
         fprintf(file,'\n');
      end
   end
```

```
            fprintf(file,'\n');

            % Gerando os simplexos adjacentes que também são transversais
            % Varrendo as faces de dimensão n-k do esqueleto combinatório
            for i = 1:nSkel(n-k)
                % Calculando o vértice a ser pivoteado
                [r]           = GetPivotSimplex(n+1,Skel{n-k+1}(1,:),Skel{n-k}(i,:));
                % Pivotendo com relação a esta faceta
                [value g1 s1] = Pivoting(n,r-1,Grid,P,Division,Base,BaseP);
                % Se o pivoteado está no domínio
                if value == 1
                    % Verificando se este simplexo já foi processado
                    [value] = InList(NSimpProc, SimpProc, g1, s1);
                    if value == 0
                        % Insere na lista de simplexos a serem processado
                        [NSimpTrans SimpTrans] = InsertList(NSimpTrans,SimpTrans,g1,s1)
                    end
                end
            end
        end
        % Armazenando o último hipercubo processado
        gold = g;
    end
    % Finalizando o arquivo do esqueleto combinatório
    fprintf(file,'-1\n');
    % Fechando o arquivo com o esqueleto combinatório
    fclose(file);
    return
end
```