View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Repositorio da Producao Cientifica e Intelectual da Unicamp

Construção de um software de Elementos Finitos usando Programação Genérica/Generativa: considerações sobre C++, performance e generalidade

Este exemplar corresponde à redação final da tese devidamente corrigida e defendida por **Renato Fernandes Cantão** e aprovada pela comissão julgadora.

Campinas, 26 de novembro de 2004

Prof. Dr. João Frederico C. A. Meyer
Orientador

Banca examinadora

Prof. Dr. João Luiz Filgueiras de Azevedo
Prof. Dr. Luis Carlos de Castro Santos
Prof. Dr. Francisco A. M. Gomes Neto
Prof. Dr. Aurélio Ribeiro Leite de Oliveira

Tese apresentada ao Instituto de Matemática, Estatística e Computação Científica, UNICAMP, como requisito parcial para obtenção do título de **Doutor em Matemática Aplicada**.

Resumo

Este trabalho descreve a construção de uma biblioteca computacional (chamada Osiris) que implementa as funcionalidades necessárias para a execução de simulações baseadas no método dos elementos finitos. Os objetivos de Osiris são a flexibilidade, o desempenho e a extensibilidade. Estes quesitos são satisfeitos não apenas pela escolha da orientação por objetos (C++), mas também pelo uso e extensão do estado da arte em Programação Genérica e Programação Generativa. Vários exemplos numéricos são apresentados, alguns de interesse em Biomatemática.

Abstract

This work describes the creation of a computational library (named Osiris) that implements the basic functionalities needed in usual Finite Element simulations. Osiris' objectives are flexibility, performance and extensibility. These requirements are fulfilled not only by the choice of an object oriented language (C++), but also by the use and extension of the state-of-the-art in Generic Programming and Generative Programming. Several numerical examples are discussed, some of interest from a Biomathematics point of view.

Agradecimentos

Meu Doutorado foi longo e teve alguns momentos muito sofridos. Devo muito a várias pessoas o fato de ter conseguido chegar quase inteiro até a seção de agradecimentos, mas sem dúvida alguma, a pessoa que mais compartilhou comigo este "parto" foi minha esposa Luiza. Mesmo sendo do ramo (Doutora em Engenharia Elétrica) e jurando pra mim que o padre disse "...na saúde e na doença, na riqueza e na pobreza, programando e escrevendo a tese ...", sei que ela fez mais do que seria razoável nesta situação difícil, mesmo para minha companheira de vida. Sem ela e sua "reza brava" este trabalho não existiria. A você, Luiza, meu mais sincero e amoroso obrigado!

Ao meu orientador, o Joni, sou grato pelos longos anos de trabalho conjunto que tornaram-se amizade. Ele me ensinou a pensar 112 vezes antes de falar, a tomar café sem açúcar e a ser independente. Valeu Chefinho!

Ao professor Petrônio Pulino vai um agradecimento especial pelos muitos cursos, pelas horas de discussão, pelas listagens, pelos conselhos e pelas dicas de Elementos Finitos.

Agradeço também a minha família (mãe, irmã e sobrinhos) pelo apoio incondicional, pela torcida e pela paciência. Às vezes também pela compreensão com a minha ausência.

Ao Akiles pelas horas intermináveis de discussão e pelas excelentes idéias em programação. Ao Sérgio pelo bom humor, pela vasta cultura, pelos micros e pela guarida. Aos colegas de empresa Alberto e Cláudio por acreditarem mais nas minhas idéias do que eu mesmo.

Às professoras Vera, Sandra e Margarida, por contribuirem tanto com minha formação e por me trazerem um pouco de luz em períodos sombrios.

Agradecimentos gerais ao povo do projeto CFD da Embraer, a todos do IMECC que me acolheram bem durante mais de uma década, aos amigos do passado e do presente.

Aos meus "bebês", Costelinha e Pateta, pela companhia incansável durante as madrugadas.

Quero fazer um agradecimento especial a todas as pessoas que de alguma forma se envolvem ou se envolveram na construção do maravilhoso mundo do Software Livre. Sem o esforço por trás de GCC, make, gdb, ddd, emacs, Mozilla, Linux, OpenDX, LaTeX, POOMA e outras dezenas de softwares livres de altíssima qualidade, não conseguiríamos hoje enxergar além de certas janelas.

À Capes sou grato pelo apoio financeiro durante os quatro anos regulamentares de bolsa.

E para aqueles que passaram 6 anos me perguntando "quando você vai terminar a tese" minha resposta: está pronta.

"Everything that has a beginning, has an end."
The Oracle

Sumário

I História 12

1 Introdução 13

2 C++ e Computação Científica: uma combinação viável? 15

2.1 Introdução a C++ 15

2.1.1 Orientação por objetos 15

2.1.2 Modelagem 15

2.1.3 Herança e reutilização 16

2.1.4 Polimorfismo 16

2.1.5 Sobrecarga de operadores 18

2.2 Gargalos inerentes a C++ 18

2.2.1 Introdução 18

2.2.2 Avaliação de expressões aos pares 19

2.2.3 Polimorfismo: o custo da abstração 21

2.2.4 Aliasing 22

3 C++ e templates: estendendo as capacidades da linguagem 24

3.1 Introdução 24

3.2 Templates [DD01, Eck95] 24

3.3 Metaprogramas templates 26

3.4 Expressões templates [Vel00, Vel95a, CCH+98a, HC00] 27

3.4.1 Templates recursivos 27

3.4.2 A construção de uma expressão template 28

3.4.3 Uma implementação mínima do mecanismo de expressões templates 29

3.5 Especialização 31

3.5.1 Especialização explícita 32

3.6 Classes *traits* [Mye95] 32

II Forja 35

4 A forja de objetos 36
4.1 Introdução 36
4.2 Programação Orientada a Aspectos (OA) 36
4.3 Classes mixin 38
4.3.1 Introdução 38
4.3.2 Classes mixin como modificadoras 39
4.3.3 Classes mixin para a construção de objetos 43
4.4 Meta construções 45
4.4.1 Introdução 45
4.4.2 Meta `IF` 45
4.4.3 Meta comparações 50
4.4.4 Meta `SWITCH/CASE` 50
4.4.5 Outras meta construções 52

5 Repositórios de configuração: objetos com DNA 54
5.1 Introdução 54
5.1.1 Repositórios de configuração 54
5.1.2 Um exemplo inicial 55
5.2 Componentes para geração de configurações 55
5.2.1 Diagramas de características 57
5.2.2 DSL para Elementos Finitos 57
5.3 Detalhes de implementação do configurador 60
5.3.1 A descrição da DSL 60
5.3.2 Processamento de configurações (parsing) 62
5.3.3 O gerador de elementos 63
5.4 Outros configuradores 64
5.4.1 Configurador da base 65
5.4.2 Configurador da transformação padrão ↔ real 66
5.4.3 Configuradores dos operadores diferenciais 67

III Amálgama 71

6 Operadores e expressões templates 72
6.1 Introdução 72
6.2 Pooma 72
6.2.1 A biblioteca 72
6.2.2 Expressões templates em Pooma 73
6.2.3 Referenciando os operandos 75
6.2.4 Operadores unários 75
6.2.5 Operadores binários 76

6.2.6 O engine de expressões . . . 76
6.2.7 Árvore de expressões . . . 76
6.2.8 Tags de operadores . . . 77
6.3 Operadores diferenciais e POOMA: bases . . . 79
6.3.1 Geometria, integração numérica e POOMA . . . 79
6.3.2 Avaliação da base . . . 82
6.3.3 Transformações padrão $\leftrightarrow$ real . . . 86
6.4 Configurando os operadores . . . 90
6.4.1 DSL para operadores . . . 90
6.5 Condições de fronteira . . . 92
6.5.1 DSL para condições de fronteira . . . 92

7 Operadores essenciais **94**
7.1 Introdução . . . 94
7.2 Operadores do tipo $\nabla \cdot (-\alpha \nabla u)$ . . . 95
7.2.1 Difusão constante . . . 95
7.2.2 Difusão espacial . . . 100
7.3 Operadores do tipo σu . . . 104
7.3.1 Decaimento constante . . . 106
7.3.2 Decaimento espacial . . . 107
7.4 Operadores do tipo $\nabla \cdot (\mathbf{V} u)$ . . . 108
7.4.1 Advecção com velocidade constante . . . 109
7.4.2 Advecção com variação espacial do campo de velocidade . . . 109
7.5 Operador de interpolação . . . 109

IV Galeria **112**

8 Diversos exemplos e aplicações **113**
8.1 A equação de Poisson . . . 113
8.1.1 Cenário 1 . . . 114
8.1.2 Cenário 2 . . . 120
8.1.3 Cenário 3 . . . 121
8.1.4 Cenário 4 . . . 122
8.2 Difusão espacial . . . 126
8.2.1 Cenário 1 . . . 126
8.2.2 Cenário 2 . . . 128
8.2.3 Cenário 3 . . . 130
8.3 Interação poluente $\leftrightarrow$ espécie animal . . . 135
8.3.1 O poluente . . . 135
8.3.2 A população . . . 135
8.3.3 O domínio . . . 136

8.3.4 Cenário 1 . . . 136
8.3.5 Cenário 2 . . . 138

Conclusões, faltas e trabalhos futuros **145**
Conclusões gerais . . . 145
Faltas . . . 145
Trabalhos futuros . . . 147

Referências Bibliográficas **148**

A Elementos de Referência **151**
A.1 Triângulo de referência . . . 151
A.2 Quadrilátero de referência . . . 152
A.3 Tetraedro de referência . . . 153

B Listagens **155**
B.1 Meta SWITCH/CASE . . . 155
B.2 Meta comparações . . . 157

C Listagens para os exemplos de Poisson **158**
C.1 SolversCommon.h . . . 158
C.2 ElementDef.h . . . 159
C.3 Rhs_1.h e Rhs_2.h . . . 160
C.3.1 Rhs_1.h . . . 160
C.3.2 Rhs_2.h . . . 161
C.4 PoissonCteCoeff.h . . . 162
C.5 PoissonCteCoeff.cc . . . 165

D Listagens para os exemplos com difusão espacial **167**
D.1 SolversCommon.h . . . 167
D.2 ElementDef.h . . . 167
D.3 Rhs_1.h e BallSrc.h . . . 167
D.3.1 BallSrc.h . . . 167
D.4 Corner.h e HalfCube.h . . . 168
D.4.1 Corner.h . . . 168
D.4.2 HalfCube.h . . . 169
D.5 DoubleArch.h . . . 170
D.6 Diffusion.h . . . 171
D.7 Diffusion.cc . . . 175

E Listagem para os exemplos de Interação Espécie/Poluente **177**
E.1 PollutantSpecies.cc . . . 177

Lista de Figuras

2.1 Hierarquia de classes de matrizes. 16
2.2 Comparação de tempos de execução para operação de soma vetorial $w = x + y + z$, com vários tamanhos de vetores. 20

3.1 Templates recursivos. 28
3.2 Árvore de tipos construída com templates. 28

4.1 Componentes e aspectos. 37
4.2 Vários níveis mixin. 39
4.3 Nova configuração de classes com introdução da classe mixin. 41
4.4 Esquematização do meta `IF`. 46

5.1 Exemplo de diagrama de características. 57
5.2 DSL para elementos finitos. 59
5.3 Instância de `FE_Family< lagrange< 1 > >`. 61
5.4 Relações entre as diversas classes que formam um elemento. 64
5.5 DSL para o conceito "base de elementos finitos". 65
5.6 DSL para transformação elemento padrão $\leftrightarrow$ elemento real. 67

6.1 Árvore binária representando a expressão $y = \sin(x) + x$. 75
6.2 Árvore de expressão usando templates. 77
6.3 Representações gráficas para pontos e pesos de integração bidimensionais. 82
6.4 Representações gráfica para pontos e pesos de integração tridimensionais. 82
6.5 DSL para operadores diferenciais. 91
6.6 DSL para condições de fronteira. 93

7.1 Gráfico de (7.17). 103

8.1 Geometria para os Cenários 1, 2 e 3. 115
8.2 Curvas de nível e densidade para a solução aproximada dos Cenários 1, 2 e 3. 119
8.3 Geometria para o Cenário 4. 123
8.4 Corte nas superfícies de nível e densidade para a solução aproximada do Cenário 4. 125
8.5 Curvas de nível e densidade para a solução aproximada do Cenário 1 com α espacialmente variável. 129

8.6 Curvas de nível e densidade para a solução aproximada do Cenário 2. 133
8.7 Representação 3D para a solução aproximada do Cenário 2. 134
8.8 Superfícies de nível com corte para a solução aproximada do Cenário 3. 134
8.9 Domínio $[0, 5] \times [0, 1]$ para os cenários 1 e 2 (interação poluente-população). 136
8.10 Curvas de nível e densidade para a solução aproximada do Cenário 1 com $k = 0$. 139
8.11 Gráfico de densidade para a solução aproximada da espécie do Cenário 1. 140
8.12 Gráficos de nível para a solução aproximada da espécie do Cenário 1. 141
8.13 Curvas de nível e densidade para a solução aproximada do Cenário 2 com $k = 0$. 142
8.14 Gráfico de densidade para a solução aproximada da espécie do Cenário 2. 143
8.15 Gráficos de nível para a solução aproximada da espécie do Cenário 2. 144

A.1 Triângulo de referência. 151
A.2 Quadrilátero de referência. 152
A.3 Tetraedro de referência. 154

Lista de Tabelas

2.1 Comparação de tempos de execução entre um método virtual e um não virtual. 22

5.1 Repositório de configuração para `TriangleP1_t`. 56
5.2 Diagramas de características mais comuns. 58

6.1 Representação gráfica de estruturas de dados. 81
6.2 Resultados do cálculo do jacobiano de uma transformação padrão ↔ real. 88
6.3 Resultados da aplicação do jacobiano uma transformação padrão ↔ real. 88

8.1 Máquinas usadas nas simulações. 113
8.2 Cenário 1. 114
8.3 Tempos de execução para o Cenário 1 (máquina `honey`). 118
8.4 Cenário 2. 120
8.5 Tempos de execução para o Cenário 2 (máquina `gambrinus`). 121
8.6 Cenário 3. 121
8.7 Tempos de execução para o Cenário 3 (máquina `honey`). 122
8.8 Cenário 4. 123
8.9 Tempos de execução para o Cenário 4 (máquina `honey`). 124
8.10 Resumo dos Cenários de 1 a 4. 124
8.11 Cenário 1. 126
8.12 Tempos de execução para o Cenário 1 (máquina `honey`). 128
8.13 Cenário 2. 128
8.14 Tempos de execução para o Cenário 2 (máquina `honey`). 130
8.15 Cenário 3. 131
8.16 Tempos de execução para o Cenário 3 (máquina `honey`). 131
8.17 Resumo dos Cenários de 1 a 3. 131
8.18 Cenário 1. 137
8.19 Tempos de execução para o Cenário 1 (máquina `crucio`). 137
8.20 Tempos de execução para o Cenário 2 (máquina `crucio`). 138

Parte I

História

CAPÍTULO 1

Introdução

Desde seu gradativo aparecimento como parte do grupo de Biomatemática do IMECC, na Unicamp, um grupo de pesquisadores vem dedicando seus principais esforços em termos da modelagem matemática, da aproximação numérica, da formulação algorítmica e na execução de simulações computacionais de problemas geralmente classificados como sendo da Ecologia Matemática.

Foi nessa linha que se fizeram suas respectivas teses de mestrado, os hoje professores doutores:

- Diomar Cristina Mistro, abordando o problema da poluição por mercúrio no rio Madeira;
- Sonia Elena Palomino-Castro, estudando o projeto de instalação de uma unidade de geração de eletricidade por meios termomecânicos na região de Campinas/Paulínia;
- Renato Fernandes Cantão, na modelagem do movimento de manchas de óleo no canal de S. Sebastião, SP;
- Renata Cristina Sossae, na análise numérica e na simulação de sistemas não-lineares de convívios intra e inter-específicos;
- Mateus Bernardes, no estudo do movimento de manchas superficiais de poluentes em corpos aquáticos de baixa circulação, e
- Geraldo Lúcio Diniz, na compreensão de certos fenômenos de migrações de espécies de peixes em função de mudanças bruscas de habitats.

Um continuação destes trabalhos resultou em alguns doutorados, como os de:

- Profª Drª Maria Conceição Perez Young-Pessoa, na análise da presença do bicudo do algodoeiro e adequadas técnicas de manejo dessa praga;
- Profª Drª Tânia Maria Salgado Vilela Lacaz, estudando a dispersão espaço-temporal desse mesmo bicudo do algodoeiro;

- Prof. Dr. Sílvio de Alencastro Pregnolatto, na modelagem e na simulação de certa epizootia em ambiente de pantanal

- Prof. Dr. Geraldo Lúcio Diniz, na simulação algorítmica e computacional da dispersão de poluentes em sistemas ar-água com características de baixa circulação;

- Profª Drª Renata Cristina Sossae, no estudo, na aproximação numérica e na simulação computacional dos efeitos de um produto tóxico sobre uma parte de cadeia trófica, também em região de pantanal, e

- Profª Drª Rosane Ferreira de Oliveira, modelando a circulação e o resultante movimento de manchas de óleo na baía da Ilha Grande, RJ.

Ainda, o doutorando Júlio César Saavedra Vásquez, na fase final de sua dissertação de doutorado, vem trabalhando com a análise numérica e a simulação computacional de plumas de efluentes tóxicos em domínios costeiros, considerado tri-dimensionalmente.

Em todos estes casos foi necessário discretizar o domínio em estudo, algumas vezes chegando a detalhes bastante descritivos dessa região, seguindo-se a aproximação de uma Equação Diferencial Parcial ou Sistemas lineares ou não-lineares de EDP's ou, ainda, um conjunto de EDP's acopladas por condições de contorno numa interface do domínio. Ora, em todos estes casos, as aproximações espaciais envolveram, de algum modo, uma aproximação com o Método de Galerkin via Elementos Finitos, embora tivessem sido diversas as opções de ordem, grau, algoritmo, e geometria. Em outras palavras, todos estes trabalhos envolveram de algum modo a discretização de um domínio, a aproximação numérica de operadores de difusão efetiva dependente ou não do espaço, do tempo ou da própria solução, de um transporte advectivo ou convectivo, também podendo variar espacial ou temporalmente, além de fenômenos de decaimento, de fonte ou de sorvedouro. E todos estes trabalhos envolveram a criação de códigos computacionais de resolução e de visualização de resultados.

A rigor, uma categorização dessas pesquisas poderia separá-las genericamente em dois grupos principais, sendo o primeiro o do comportamento espaço-temporal de poluentes num meio e o segundo o do estudo de fenômenos de dispersão populacional. No trabalho de Sossae [Sos03], porém, reúnem-se estes dois enfoques, de modo a identificar os diversos e sucessivos passos usados na modelagem da presença de poluentes num determinado meio e sua influência sobre populações interativas desse meio. Este modo construtivo de cooperação acadêmica, porém, embora tenha incluído uma intensa e constante troca de informações e experiências na parte de concepções e execuções algorítmicas, não se estendeu aos programas em si, refeitos para cada novo caso, cada nova quadratura, cada nova discretização de domínio. Numa dessas renovadas definições de algoritmo, nasceu a idéia de se constituir um recurso de alto nível e de total confiabilidade para a continuação dos trabalhos do grupo, tanto em termos de potenciais cooperações e novos esforços conjuntos de estudo, quanto relativamente a novos pesquisadores e seus respectivos trabalhos.

Diversas tentativas de unificar as experiências e recursos foram se cristalizando – e o resultado pode ser identificado como o Netuno ,um programa para simulação de dispersão de poluentes em meios aquosos usando elementos finitos, e seu sucessor, o Osiris, voltado a simulações mais genéricas, mas ainda no campo dos elementos finitos.

CAPÍTULO 2

C++ e Computação Científica: uma combinação viável?

Apresentaremos aqui os motivos que fazem com que a linguagem C++, quando usada de forma tradicional, perca em termos de desempenho em relação a linguagens procedurais como FORTRAN 77 ou C, por exemplo, aparentemente invalidando-a como linguagem para o desenvolvimento de aplicações em Computação Científica.

2.1 Introdução a C++

2.1.1 Orientação por objetos

A *programação orientada por objetos*[1], aqui num contexto totalmente ligado à C++ e à Computação Científica, tem aspectos muito positivos do ponto de vista de Engenharia de *Software*. De fato, esta filosofia de programação facilita a manutenção e a extensão do código e, ainda, a representação de problemas de forma muito mais eficaz e rápida do que no caso das linguagens procedurais. Para basear nossa discussão, apresentamos os elementos fundamentais da OO a seguir (ver [Eck95, DD01] para um aprofundamento maior sobre a linguagem).

2.1.2 Modelagem

Um objeto nada mais é do que um *representante funcional* de uma idéia modelada pelo programador e que em C++ recebe o nome de *classe*. A classe compreende *características* e *ações*. Num contexto mais amplo em OO, as características recebem o nome de *dados da classe* e as ações de *métodos*. Em C++, os métodos são chamadas *funções membro* [Eck95]. Um objeto representante de uma determinada classe é denominado *instância* desta classe. O processo de modelagem de conceitos por OO é bastante poderoso, pois permite isolar saudavelmente as partes constituintes de um problema.

[1] Abreviada por “OO”.

2.1.3 Herança e reutilização

Herança é a capacidade de construção de novas classes (chamadas *classes herdeiras*) que absorvem características e ações de classes pré-existentes (*classes ancestrais*). Este processo estimula a *reutilização de software*, pois se a classe A modela um determinado conceito, e desejamos modelar um conceito similar através da classe B, não é necessária a construção de B do zero. Dentro de determinados limites impostos pelo projeto ou pelo problema, pode-se construir B como uma nova classe, porém aproveitando de A aquilo que for comum a ambas. A notação comum para herança é $\boxed{\text{A}} \longleftarrow \boxed{\text{B}}$ significando que a classe B é herdeira de A (ou é derivada de A). A herança pertence a um conjunto de técnicas de construção de objetos chamada *composição*.

Para exemplificarmos melhor, tomemos a hierarquia de classes representada na figura 2.1. Lá temos a classe ancestral `Matrix`, que será *especializada* através da herança, ou seja, nela serão incluídos métodos e características que melhor a adaptem a outra tarefa. Duas especializações diretas são as classes herdeiras `SymmMatrix` e `BandMatrix`. Parte da implementação da classe `Matrix` pode ser aproveitada — por exemplo, contabilidade de número de linhas e colunas — porém as operações de acesso (obter elemento a_{ij}) devem ser especializadas para que sejam *aproveitadas as características de cada matriz*. No caso de `SymmMatrix`, necessitamos armazenar apenas a parte triangular superior da matriz e devemos garantir que o acesso aos elementos a_{ij} e a_{ji} correspondam à mesma posição de memória. Para `BandMatrix`, devemos armazenar bandas suficientes e verificar se o elemento requisitado a_{ij} está dentro da banda (e retorná-lo) ou não (retornando zero). Vemos que o processo se estende, pois matrizes tridiagonais — representadas por `TridMatrix` — podem ser derivadas diretamente de `BandMatrix`, já que são um caso especial desta última.

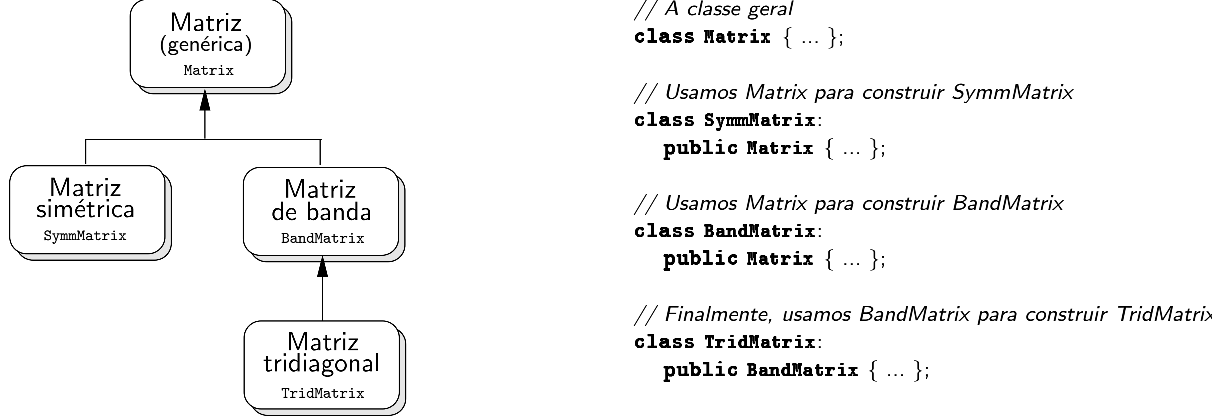


Figura 2.1: Hierarquia de classes de matrizes.

2.1.4 Polimorfismo

Polimorfismo é a capacidade de fazer com que o curso da ação seja decidido *em tempo de execução*. Um exemplo de uso do polimorfismo, que é típico em Computação Científica usando C++, é a multiplicação de matrizes por vetores. Se a utilização do operador produto $*$ for polimórfica, a estrutura

interna da matriz (simétrica, banda, tridiagonal, por exemplo) fica invisível a quem a usa. Desta forma podemos construir métodos genéricos que funcionem para qualquer tipo de matriz[2]. Consideremos o exemplo da listagem 2.1.

```
    // Método dos Gradientes conjugados.
    Vector ConjugateGradient( const Matrix& A, const Vector& b, const Vector& x0 )
    {
      // Em algum ponto do algoritmo, aparece uma multiplicação de matriz por vetor:
 5
      r = b - A * x;   // Cálculo do residuo e continua...
    };

    main()  // No programa principal.
10  {
      SymmMatrix A( 20, 20 ); // A ∈ ℝ^{20×20}, simétrica.
      BandMatrix B( 20, 20, 5 ); // B ∈ ℝ^{20×20}, banda 5.
      TridMatrix C( 20, 20 ); // C ∈ ℝ^{20×20}, tridiagonal.

15    Vector x0( 20 ), y( 20 ), b( 20 ); // x_0, y, b ∈ ℝ^{20}

      // Supor alguma inicialização nas matrizes e vetores...

      // Resolvendo A*x = b usando o mesmo GC, com matrizes estruturalmente diferentes.
20    y = ConjugateGradient( A, b, x0 );
      y = ConjugateGradient( B, b, x0 );
      y = ConjugateGradient( C, b, x0 );
    }
```

Listagem 2.1: Exemplo de polimorfismo.

Neste exemplo temos um trecho de uma rotina que implementa o método dos Gradientes Conjugados (GC) [BBC+94]. Notemos que, além dos vetores usuais de entrada (chute inicial e lado direito), a rotina recebe também um objeto `Matrix`, que está na raiz da árvore de herança matricial (figura 2.1).

Notemos que, para cada multiplicação matriz/vetor presente no método, usamos implicitamente um tipo diferente de matriz (linhas 11–13). Quando fazemos a chamada com a matriz A, as operações de multiplicação de matriz por vetor do GC são aquelas relacionadas com as matrizes simétricas. Se usamos B, são aquelas relacionadas às matrizes de banda e assim por diante. Assim, se um novo tipo de matriz for definido, pertencendo à mesma estrutura derivada de `Matrix`, a implementação do GC pode ser usada prontamente, *sem alterações*.

É claro que ainda é *responsabilidade do implementador* das classes matriciais providenciar o operador produto $*$ apropriado para cada uma delas, fazendo o possível, é claro, para usufruir das características especiais das matrizes em questão. O polimorfismo é realizado em C++ através da herança e da palavra chave **`virtual`**.

[2] Qualquer tipo de matriz no sentido de estruturas de dados e não no matemático.

2.1.5 Sobrecarga de operadores

Como já intuimos acima, operadores como $+$, $-$, $*$ (e outros) podem ser sobrecarregados[3], ou "*ensinados*" a operar com objetos definidos pelo usuário; de fato, qualquer função pode ser sobrecarregada, dentro das regras semânticas da linguagem. Junto com o polimorfismo, esta sobrecarga permite uma forma natural de representação das operações entre classes representativas de conceitos matemáticos.

Estas capacidades da OO — modelagem, herança, polimorfismo e sobrecarga de operadores — causaram uma revolução nos conceitos de Engenharia de Software e este entusiasmo rapidamente estendeu-se à comunidade de Computação Científica.

Porém a grande generalidade e a facilidade de manutenção de código oferecidas pela OO em C++ resultaram em uma penalização[4] significativa do ponto de vista de eficiência computacional. Os programas científicos escritos em C++ são genéricos, confiáveis, fáceis de modificar e estender, porém são também, em alguns casos, intoleravelmente lentos! Havia perdas de desempenho de até 10 vezes entre aplicações semelhantes escritas em C++ e FORTRAN[5] ou C [VJ97, Vel97].

Parte do problema devia-se ao fato de C++ ser uma linguagem relativamente nova, fazendo com que os compiladores disponíveis não fizessem um bom trabalho de otimização de código e também não conseguissem acompanhar a evolução da linguagem. À medida que a estrutura da linguagem foi se estabilizando, os compiladores foram acrescidos de algoritmos para otimização de código cada vez mais agressivos, diminuindo consideravelmente seu peso na aparente lentidão da linguagem.

Mas mesmo com este aumento de qualidade dos compiladores, C++ ainda perdia em desempenho para linguagens procedurais. Chegou-se ao ponto de relegar C++ a uma linguagem tipicamente de construção de interfaces com o usuário (tarefa, inclusive, a que ela se presta muito bem), uma noção errônea, como veremos.

2.2 Gargalos inerentes a C++

2.2.1 Introdução

Em Computação Científica, chamamos de *gargalos* aqueles pontos onde um determinado algoritmo ou processo gasta uma parte considerável de seu tempo de execução, em comparação com o resto das operações. Normalmente são os gargalos que merecem maior atenção quando do aprimoramento de um algoritmo.

Veremos na seqüência quais são os principais gargalos em C++ e como eles aparecem naturalmente dentro da linguagem, quando utilizada de forma tradicional, fora de um contexto de Computação Científica.

[3] Termo técnico. Do Inglês, *overloaded*.

[4] Termo técnico. Do Inglês, *penalty*. Usado aqui no contexto do aumento de tempo de processamento de uma determinada computação em virtude da adição de características não inerentes ao algoritmo, como a linguagem utilizada, o sistema operacional, a arquitetura de *hardware*, etc.

[5] O FORTRAN é usado aqui como comparação por ser uma das primeiras linguagens de uso geral amplamente utilizada em computação científica.

2.2.2 Avaliação de expressões aos pares

A sobrecarga de operadores, aliada ao polimorfismo, resulta em uma maneira elegante e natural de escrever expressões matemáticas envolvendo objetos. Porém operadores sobrecarregados são *sempre avaliados aos pares* [VJ97, Vel95a, HC00], pelo menos em C++ padrão. Mas antes do exemplo, apresentemos uma definição.

Chamamos de *temporário*[6] uma variável totalmente sob controle do compilador. Ela não é explicitamente declarada pelo programador, sendo criada e destruída pelo compilador de acordo com a necessidade. Seu tipo é definido de forma coerente com as variáveis com as quais ela opera.

Suponhamos que `Vector` seja uma classe que modele o conceito de vetor, com as respectivas operações de soma e atribuição. Se implementada de maneira usual, a expressão `w = x + y + z` resultará em 3 *loops*[7] e o uso de dois objetos temporários do tipo `Vector`. A listagem 2.2 mostra como o compilador traduz este trecho de código.

```
Vector temp1(N);   // Primeiro temporário.

for( int i = 0; i < N; i++ ) { // Primeiro loop.
   temp1(i) = x(i) + y(i);
}

Vector temp2(N);    // Segundo temporário.

for( int i = 0; i < N; i++ ) { // Segundo loop.
   temp2(i) = temp1(i) + z(i);
}

for( int i = 0; i < N; i++ ) { // Atribuição a w (o terceiro loop).
   w(i) = temp2(i);
}
```

Listagem 2.2: Soma de vetores: versão do compilador.

Obviamente, em uma linguagem procedural, como C padrão e em FORTRAN de modo totalmente análogo, escreveríamos de forma muito mais simples, como na listagem 2.3.

```
double x[N], y[N], z[N], w[N];

for( int i = 0; i < N; i++ ) {
   w[i] = x[i] + y[i] + z[i];
}
```

Listagem 2.3: Soma de vetores: versão procedural.

A formulação da listagem 2.3 possui claramente um desempenho superior. A título de comparação, foram feitos três programas com aritmética de ponto flutuante de precisão dupla, para somar três

[6]Comumente usa-se a forma masculina.
[7]Termo técnico. Laço de repetição.

vetores e atribuir o resultado desta soma a um quarto vetor. A versão em Fortran usa vetores estaticamente alocados em memória, sem blocos de `common` e faz a soma de forma similar à da listagem 2.3. A versão C usa alocação dinâmica de memória e também segue o padrão da listagem 2.3 para a operação de soma. Já a versão C++ usa uma classe que simula o comportamento de um vetor, definindo para tanto os operadores de atribuição (**`operator=`**) e de soma (**`operator+`**). A figura 2.2 resume os resultados para vetores entre cem mil e quatro milhões de componentes. As versões C e Fortran possuem desempenho similar, enquanto que a versão em C++ é aproximadamente 60% mais lenta devido às avaliações aos pares. Há um quarto resultado no gráfico correspondente à utilização de Pooma, biblioteca computacional que será formalmente apresentada na seção 6.2.

Os programas teste foram compilados com a suite GCC[8] 3.3.4, com otimização padrão `-O2` e executados em um máquina AMD Athlon-XP 1800+ com 256Mb de memória. Os testes foram repetidos 3 vezes. O tempo associado a cada teste é a média de tempo das três execuções. O tamanho máximo dos vetores foi escolhido de forma a todos caberem na memória RAM.

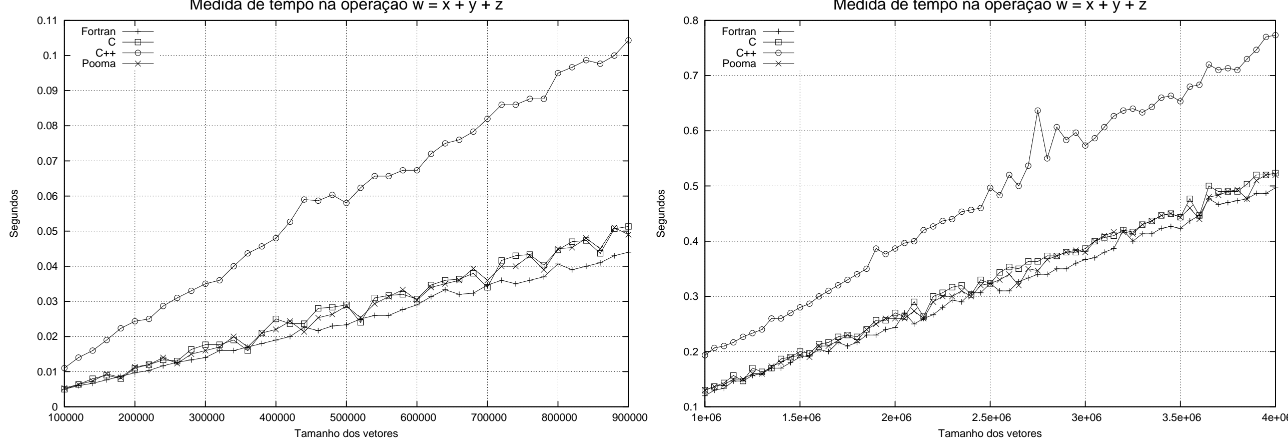


Figura 2.2: Comparação de tempos de execução para operação de soma vetorial $w = x + y + z$, com vários tamanhos de vetores.

De acordo com [Vel00], a manipulação de vetores pequenos é afetada pelo gasto de tempo extra dos operadores **`new`** (alocação de memória) e **`delete`** (desalocação) aplicados aos temporários, resultando em uma perda de desempenho médio de 90% em relação ao mesmo algoritmo codificado diretamente em C. Para vetores médios (ainda dentro da memória cache), o overhead concentra-se nos loops extras e nos temporários. O uso massivo de temporários pode ainda provocar o esgotamento do cache, forçando o uso da memória convencional. Finalmente, temos os grandes vetores, que concentram todo o custo computacional nos temporários que, novamente, sobrecarregam o cache, recaindo no uso da memória convencional.

[8] GNU Compiler Collection, `http://www.gnu.org/gcc`.

2.2.3 Polimorfismo: o custo da abstração

O polimorfismo, como dissemos na introdução, é a capacidade do programa decidir, em tempo de execução, qual função deve ser chamada, *dependendo do objeto presente no argumento de entrada*. Este mecanismo recebe o nome de *late binding*. Em C++ as funções membro polimórficas são chamadas de *virtuais*.

Voltando ao exemplo da introdução, suponhamos três tipos de matrizes, com um diagrama de herança como o da figura 2.1. Suponhamos também que estejamos usando o método dos Gradientes Conjugados (GC), cujo custo computacional principal é a operação de multiplicação matriz por vetor. Notemos que, se o operador produto $*$ for polimórfico, basta escrevermos uma única versão do GC, pois, graças ao polimorfismo, ela funcionará para todas as matrizes. Se chamarmos a rotina GC com uma matriz simétrica, será usado o produto `SymmMatrix` por vetor, exatamente como no exemplo da listagem 2.1. Se for chamada com uma matriz de banda, então, em tempo de execução, será chamado o produto `BandMatrix` por vetor, e assim por diante.

O importante é que *se novos tipos de matrizes forem criados, estes podem ser instantaneamente usados com a implementação do GC*, desde que o operador produto $*$ esteja apropriadamente definido (ou herdado) para cada novo tipo de matriz.

Percebe-se então que o polimorfismo é um trunfo fantástico, pois permite a generalização de pontos chave em Computação Científica. E é justamente aí que reside sua maior fraqueza. A decisão em tempo de execução tem um custo, pois o programa precisa parar e consultar em uma tabela de funções possíveis chamada VTABLE, qual delas é a mais apropriada naquele momento. Voltando ao nosso exemplo, cada multiplicação matriz por vetor do GC gera uma busca na tabela de possíveis funções que fazem multiplicação, levando assim a um overhead de computação indesejado.

Pode-se argumentar que esta busca na VTABLE é rápida em comparação com a multiplicação de matriz por vetor, mas se levarmos em conta que o polimorfismo também é aplicado em larga escala — no caso de Computação Científica — em funções "pequenas" como o acesso a elementos de uma matriz, pequenas transformações e principalmente acesso a dados da classe, então teremos um gasto astronômico de tempo por causa de todas as buscas em tabelas implícitas nas operações polimórficas.

Um teste simples pode mostrar o peso relativo da virtualidade. Definimos duas classes com um único método chamado `get()`. Na primeira classe `get()` é virtual e na segunda, não. O teste consiste em chamar os dois métodos alguns milhões de vezes e medir o tempo de cada versão. A tabela 2.1 traz o resultado do teste para várias versões de `get()`. Como podemos notar, o peso relativo da virtualidade diminui conforme a complexidade do método aumenta. Por exemplo, para alocação e desalocação de memória, a diferença é mínima. Porém, para métodos simples (como incrementar um inteiro e retornar o novo valor) essa diferença é gritante.

Resumindo, *o polimorfismo torna-se prejudicial quando o tempo de execução de uma determinada função membro é comparável ao tempo de decisão da tabela de funções virtuais.*

`get()`	Δt virtual (s)	Δt direto (s)	$\Delta\%$
Incrementa um inteiro e retorna seu valor	9.52	1.52	526.31%
Eleva ao quadrado um número de ponto flutuante (precisão dupla), substituindo o valor original (operador `*=`)	16.55	3.27	406.11%
Avalia a função $\text{sen}(1.7\,x^3 - 3.7\,\sqrt{\lvert x\rvert})$, substituindo o valor original de x (precisão dupla)	108.56	56.13	93.40%
Aloca e desaloca 10 vezes um vetor de 5000 posições	23.86	23.21	2.8%

Tabela 2.1: Comparação de tempos de execução entre um método virtual e um não virtual.

2.2.4 Aliasing

O problema de *aliasing*[9] ocorre por causa do uso de ponteiros. Como sempre existe a possibilidade de dois ponteiros diferentes apontarem para um mesmo dado, certas otimizações não podem ser feitas a priori pelo compilador sob o risco da perda total de controle da memória pelo programa.

Tomemos como exemplo o código da listagem 2.4, para atualização de posto 1 de uma matriz, $\boldsymbol{A} \leftarrow \boldsymbol{A} + \boldsymbol{x}\boldsymbol{x}^T$ [Vel00].

```
void rank1Update( Matrix& A, const Vector& v )
{
   for( int i = 0; i < A.rows(); i++ )      // Loop nas linhas.
      for( int j = 0; j < A.cols(); j++ )   // Loop nas colunas.
         A(i,j) += x(i) * x(j);
}
```

Listagem 2.4: Correção de posto 1.

O ideal seria que o compilador mantivesse o valor de `x(i)` em um registrador do processador, ao invés de trazê-lo constantemente da memória dentro do loop em `j`, evitando o custo intrínseco do operador **`operator()`**. É *muito provável* — para não dizer razoável — que o projeto das classes `Matrix` e `Vector` impeça que acessos aos elementos da matriz `A` através do operador `A(i,j)`, modifiquem o vetor `x` (embora isto seja perfeitamente possível e receba o nome de *aliasing*). Mas não há como o compilador inferir isto sozinho. Mesmo que a possibilidade de aliasing seja remota, deve ser levada em conta, o que força o compilador a gerar código completamente ineficiente, embora correto, para a função acima.

Uma das tentativas para eliminar o problema de aliasing foi a criação da palavra chave denominada **`restrict`** por parte do NCEG (Numerical C Extensions Group). A idéia é que ponteiros declarados com **`restrict`** não podem sofrer aliasing, mas o comitê de regulamentação ANSI/ISO C++, orgão que define e regulamenta a linguagem, rejeitou o pedido de adoção de **`restrict`**. Mesmo assim alguns compiladores a implementam, e ela serve basicamente para informar que, para determinadas variáveis, o aliasing é impossível, deixando o compilador livre para otimizar o código.

Em suma, estas são as deficiências apresentadas por C++ quando usada como linguagem para desenvolvimento de rotinas para Computação Científica: *a avaliação de expressões aos pares*, *o polimor-*

[9] Termo técnico, sem equivalente em Português ainda. Será abolido o itálico.

fismo e o aliasing. Veremos na seção seguinte quais são as técnicas utilizadas para *"trapacear" a linguagem* e evitar essas deficiências. Essa tecnologia servirá de base para a construção do nosso software.

CAPÍTULO 3

C++ e templates: estendendo as capacidades da linguagem

O material apresentado no capítulo 2 seria mais do que suficiente para descartarmos completamente C++ como linguagem de desenvolvimento de aplicações científicas. No presente capítulo veremos que esse fato é verdadeiro somente se C++ for usada de maneira tradicional. Se usada da forma aqui proposta, reduzimos a diferença de desempenho, e mantemos — ou mesmo aumentamos — a generalidade oferecida pela linguagem.

3.1 Introdução

Existem várias técnicas para aumentar o desempenho de um determinado software, abrangendo desde a troca do compilador e a correta escolha de algoritmos até a paralelização do programa.

Aqui estamos interessados em outro tipo de alto desempenho, obtida através da *otimização de alta granularidade*. Neste tipo de otimização, partes pequenas — porém decisivas — do software são otimizadas, e não o pacote como um todo.

Vejamos agora quais são as técnicas que podem ser aplicadas na tentativa de tornar C++ competitiva enquanto linguagem para aplicações científicas usando otimizações de alta granularidade. Estas técnicas serão usadas nos capítulos seguintes para a construção das estruturas que minoram os problemas apresentados no capítulo 2.

3.2 Templates [DD01, Eck95]

A tecnologia de *templates* (gabaritos[1]) começou como uma simples curiosidade, uma espécie de macro inteligente da linguagem e que hoje evoluiu até se tornar a pedra fundamental das técnicas em

[1] Termo técnico. A tradução de template por gabarito consta de [DD01], porém não exprime corretamente o conceito em questão. Aqui "gabarito" deve ser entendido no sentido de moldes de corte-e-costura, que podem ser combinados de diversas formas, e não no sentido das réguas de gabarito do Desenho Técnico, que são rígidas e de forma pré-definida. Será abolido o itálico.

Computação Científica para C++. Tudo o que será visto daqui em diante *é baseado de alguma forma em templates*.

A idéia de template é simples: permitir ao programador criar *classes parametrizadas*. Assim, além da generalidade natural oferecida pela Orientação por Objetos, mais um nível de abstração é incluído, através de um parâmetro que permanece em aberto.

```
template< class T >
class Vector {
   public:
      Vector( int n ) { ... } // Construtor.

      Vector< T > add( const Vector< T >& v ); // Operação de soma vetorial.

   protected:
      T* data;   // Notemos que os dados são do tipo 'T'.
};

main()
{
   Vector< int > x( 10 ); // Vetor de inteiros.
   Vector< double > y( 10 ); // Vetor de números de ponto flutuante.
}
```

Listagem 3.1: Exemplo minimalista de template.

Do exemplo da listagem 3.1 observamos que a declaração de um objeto da classe `Vector` é feita por `Vector< int > x` ou `Vector< double > y`, por exemplo, resultando respectivamente, em um vetor de inteiros e um de pontos flutuantes de precisão dupla. Do ponto de vista do compilador, a declaração dos dois objetos acima é equivalente à construção de *duas classes completamente distintas*, uma para vetores com componentes inteiras e outra para vetores com componentes de ponto flutuante. Até mesmo semanticamente objetos dessas duas classes são considerados distintos.

Percebemos também que os templates são uma ferramenta extra para reaproveitamento de código: uma só versão da classe Vector, com apenas uma operação de soma (`add`), serve para vetores de qualquer tipo (inteiros, reais ou complexos). *Não é necessária a replicação de código* neste caso, pois a operação de soma é idêntica, independentemente do tipo dos elementos contidos no vetor; é claro que estes elementos devem possuir a operação de soma também definida (como é o caso para **`int`** e **`double`**). Obviamente a manutenção do software é também simplificada neste caso, pois em caso de erros na implementação, uma só classe necessita de atenção, e não várias.

Mas esses não são os únicos benefícios escondidos por trás dos templates. Sua aparente "simplicidade" resultou em um conjunto de técnicas poderosas, ao mesmo tempo mantendo as boas características da OO presentes em C++, e elevando consideravelmente o desempenho da linguagem quando usada em Computação Científica.

3.3 Metaprogramas templates

Também chamados de programas em tempo de compilação, os metaprogramas são projetados de forma a passar parte do processamento, que normalmente seria feito durante a execução, para a *compilação* [HC00, Vel95b, Vel00, VP96].

Um exemplo muito simples, porém ilustrativo, é o cálculo do fatorial [Vel95b, Vel00] (listagem 3.2).

```
template< int N > // Templates podem ser tipos específicos também!
class Factorial {
   public:
      // Uniões anônimas (enum) funcionam como inteiros constantes.
      enum { value = N * Factorial< N-1 >::value };
};

// Especialização para parar a recursão.
class Factorial< 1 > {
   public:
      enum { value = 1 };
};
```

Listagem 3.2: Metaprograma que calcula o fatorial durante a compilação.

Usando esta classe em um programa, a declaração `Factorial< 10 >::value` gera o fatorial de 10, *em tempo de compilação*. Do ponto de vista do processador, é como se `Factorial< 10 >::value` fosse uma constante pré-definida. No caso do GCC, uma única instrução de máquina é gerada carregando a constante (3628800) na memória.

Outro exemplo de metaprograma template é apresentado na listagem 3.3. A estrutura `Fibonacci` calcula em tempo de compilação o n-ésimo termo da seqüência de Fibonacci $a_n = a_{n-1} + a_{n-2}$, com $a_0 = a_1 = 1$. Portanto, `Fibonacci< 10 >::RET` é equivalente a 89, obtido em tempo de compilação.

```
template< int N >
struct Fibonacci {
   // Fibonacci:  a_n = a_{n-1} + a_{n-2}.
   enum { RET = Fibonacci< N-1 >::RET + Fibonacci< N-2 >::RET };
};

struct Fibonacci< 1 > {
   enum { RET = 1 }; // a_1 = 1
};

struct Fibonacci< 0 > {
   enum { RET = 1 }; // a_0 = 1
};
```

Listagem 3.3: Metaprograma que calcula o n-ésimo termo da seqüência de Fibonacci.

Em [Vel00, Vel95b] são dados outros exemplos interessantes de metaprogramas, como uma implementação do algoritmo de *Bubble Sort* ou de um programa que avalia a função seno em tempo de

compilação. No capítulo 4 veremos que *o conceito de metaprograma engloba também as estruturas de controle comuns em programação.*

3.4 Expressões templates [Vel00, Vel95a, CCH+98a, HC00]

A técnica de expressões templates (*expression templates*) merece ser bem analisada, pois ela ataca sozinha os problemas de avaliação em pares, o uso de temporários e a propagação de loops nas expressões usando os operadores +, −, etc. Esta técnica será um dos pontos chaves para a construção de operadores em nosso software, como explicado no capítulo 6.

3.4.1 Templates recursivos

Para melhor entendermos o conceito de expressões templates, devemos observar que os templates podem ser instanciados[2] de forma recursiva, fazendo com que classes possam ser parâmetros delas mesmas (algo como $T(T(x)) = (T \circ T)(x)$). Consideremos a classe `X`, definida pela listagem 3.4.

```
template< class Left, class Right >
class X {
   public:
      // Exportando os tipos necessários.
      typedef Left L;
      typedef Right R;
};

class END {};
```

Listagem 3.4: Classe com dois parâmetros.

Como os parâmetros `Left` e `Right` na declaração da classe `X` podem ser *qualquer tipo*, incluindo a própria classe `X`, a declaração

```
X< A, X< B, X< C, X<D, END> > > > a;
```

é equivalente à lista de tipos `[A, B, C, D]`. Por exemplo, para recuperarmos o segundo elemento da lista (B), usamos a sintaxe `a.R.L`. A figura 3.1 ilustra este tipo de construção. Notemos que A, B, C e D podem ser tipos pré-definidos ou mesmo classes. O tipo `END` é uma classe vazia, que serve apenas como marcadora de fim de lista.

Com este recurso podemos ainda representar árvores de tipos (vide figura 3.2):

```
X< X<A,B>, X<C,D> > a;
```

As expressões templates nada mais são que árvores de avaliação construídas de forma similar à acima, mas com operadores C++. Voltando ao exemplo da soma, se considerarmos uma classe `X` com três parâmetros template (listagem 3.5) o fragmento `w = x+y+z`, com `x`, `y`, `z` e `w` todos do tipo `Vector`. pode ser *representado* através de `X` por um tipo como

[2]Veja definição de *instância* na seção 2.1.2.

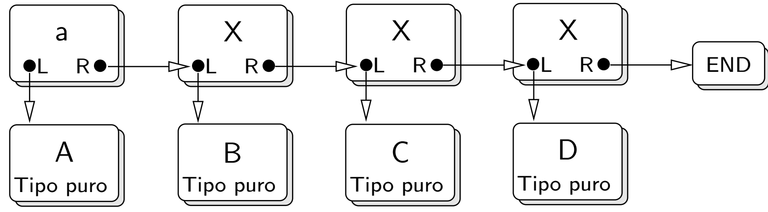


Figura 3.1: Templates recursivos.

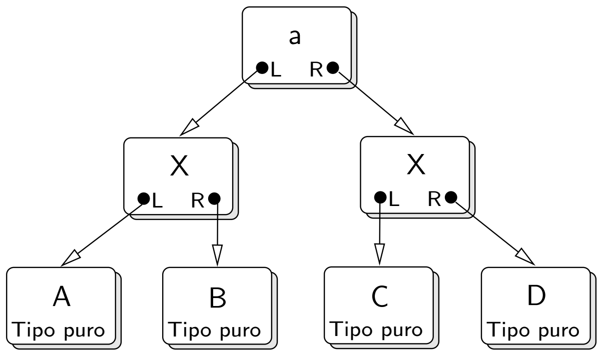


Figura 3.2: Árvore de tipos construída com templates.

```
template< class Left, class Op, class Right >
class X {
   public:
      typedef Left L;  // Exportando Left...
      typedef Right R; // ...Right e ...
      typedef Op Oper; // ...a operação.
};
```

Listagem 3.5: Classe com três parâmetros template.

```
X< Vector, assign, X< Vector, plus, X< Vector, plus, Vector > > >
```

com `plus` sendo um identificador relacionado à operação de soma e `assign` à atribuição.

3.4.2 A construção de uma expressão template

Talvez a parte mais complicada das expressões templates seja entender como são instanciados os tipos necessários para se chegar à representação desejada de uma expressão.

Acompanhemos o exemplo[3] da listagem 3.6.

Com o código da listagem 3.6 a expressão `x+y+z` é expandida para:

```
w = x + y + z;
  = X< Vector, plus, Vector >() + z;
```

[3]Para todos os efeitos, considere a construção **`template< typename T >`** completamente equivalente à construção **`template< class T >`**.

```
// Um identificador para a operação de soma.
struct plus { /* A ser implementada! */ };

class Vector { /* A mesma de sempre. */ }

// A classe X será um dos nós da árvore de avaliação.
template< typename Left, typename Op, typename Right >
class X { ... };

// O operador de soma.
X< T, plus, Vector > operator+( T, Vector ) {
  return X< T, plus, Vector >();
}
```

Listagem 3.6: Operador de soma com templates.

```
 = X< X< Vector, plus, Vector >, plus, Vector >();
```

Bem, chegamos à expressão final. Agora vamos colocar algum "estofo" na nossa implementação.

3.4.3 Uma implementação mínima do mecanismo de expressões templates

As próximas listagens mostram uma implementação minimamente funcional do conceito de expressões templates. O programa foi inteiramente retirado de [Vel00].

```
// A estrutura que encapsula o operador de soma
struct plus {
  static double apply( double a, double b ) {
    return a+b;
  };
};
```

A classe que "monta" a expressão durante a compilação.

```
template< typename Left, typename Op, typename Right >
class X {
  public:
    X( Left L, Right R ): LeftNode( L ), RightNode( R ) {};

    double operator[]( int i ) {
      return Op::apply( LeftNode[i], RightNode[i] );
    };

  protected:
    Left LeftNode;
    Right RightNode;
};
```

Nossos vetores...

```
class Vector {
  public:
```

```
      Vector( double* d, int nn ): data( d ), N( nn ) {};

      // Operador de atribuição.
      template< typename Left, typename Op, typename Right >
      void operator=( X< Left, Op, Right > expression ) {
         for( int i = 0; i < N; i++ )
            data[i] = expression[i];
      };

      // Operador de acesso a elemento.
      double operator[]( int i ) {
         return data[i];
      };

   protected:
      double* data;
      int N;
};
```

E finalmente o operador de soma:

```
template< typename Left >
X< Left, plus, Vector > operator+( Left a, Vector b ) {
   return X< Left, plus, Vector >( a, b );
}
```

Precisamos de um programa principal como palco para a ação.

```
int main() {
   // Dados numéricos.
   double x_data[] = { 2, 3, 5, 9 },
          y_data[] = { 1, 0, 0, 1 },
          z_data[] = { 3, 0, 2, 5 },
          w_data[4];

   Vector x( x_data, 4 ), y( y_data, 4 ), z( z_data, 4 ), w( w_data, 4 );

   // Somando.
   w = x + y + z;

   return 0;
}
```

Acompanhemos agora, passo a passo, como o compilador interpreta e como é feita a instanciação das classes quando de uma operação. Iniciaremos pelo *parsing* (expansão dos argumentos template).

```
w = x + y + z; // Primeiro expande 'x+y'.
  = X< Vector, plus, Vector >( x, y ) + z; // Agora expande '+z'.
  // Expressão enorme!
  = X< X< Vector, plus, Vector >, plus, Vector >( X< Vector, plus, Vector >( x, y ), z );
```

Agora devemos expandir o operador de atribuição (=):

```
w.operator=( X< X< Vector, plus, Vector >, plus, Vector >( X< Vector,
plus, Vector >( x, y ), z ) expression ) {
```

```
    for( int i = 0; i < N; i++ )
        data[i] = expression[i];
}
```

Notemos que `expression[i]` de fato refere-se ao operador `[ ]` da “expressão enorme”, que será expandida *inline*[4] de *cada nó da árvore da expressão*.

Acompanhemos:

```
data[i] = plus::apply( X< Vector, plus, Vector >( x, y )[i], z[i] );
= plus::apply( x[i], y[i] ) + z[i];
= x[i] + y[i] + z[i];
```

Portanto, o resultado final é:

```
for( int i = 0; i < w.N; i++ )
    w.data[i] = x.data[i] + y.data[i] + z.data[i];
```

Sem temporários, em um único loop!

3.5 Especialização

Outro uso interessante da metaprogramação template é na criação de código especializado para pequenas tarefas. O interesse advém da comparação de “tamanho” entre a tarefa a ser executada e o tempo gasto para decisão no polimorfismo. Tarefas pequenas tendem a perder mais desempenho com polimorfismo do que tarefas grandes (veja tabela 2.1).

Um exemplo de código especializado é o metaprograma abaixo (que imita o comportamento de um loop **for**):

```
template< int I > // A função geral.
inline double meta_dot( double* a, double* b ) {
    return meta_dot< I-1 >( a, b ) + a[I] * b[I];
}

// A especialização para terminar a recursão.
inline double meta_dot< 0 >( double* a, double* b ) {
    return a[0] * b[0];
}
```

Supondo dois vetores de 3 componentes cada, `double x[3]`, `double y[3]`, a chamada

```
double z = meta_dot< 2 >( x, y );
```

é expandida pelo compilador em:

```
z = x[0] * y[0] + x[1] * y[1] + x[2] * y[2];
```

Novamente, sem loops! O compilador consegue otimizar totalmente o cálculo da soma, resultando diretamente no valor da soma em tempo de compilação.

[4] Termo técnico. Uma função *inline* não provoca desvio no fluxo do programa. Seu código compilado é colocado diretamente no ponto de chamada. Desta forma economizamos não apenas o desvio do fluxo de execução, mas também o tempo perdido trocando argumentos com a função.

3.5.1 Especialização explícita

A maquinaria template presente em C++ também favorece a especialização explícita em certos casos, com controle do programador.

Imagine por exemplo, uma classe que, dado um elemento finito, calcule a aproximação do laplaciano neste elemento, retornando a submatriz de rigidez correspondente. Uma forma geral desta classe seria:

```
template< class Element >
class LocalLaplacian {
   // Função que aproxima o laplaciano.
   Submat eval() { ... }
};
```

Notemos que, se bem projetada, esta classe funciona para *qualquer* elemento finito que se encaixe no padrão de elemento esperado pelo template.

Suponhamos que, por motivos de desempenho, necessitemos de uma versão altamente especializada deste laplaciano, para elementos triangulares de Lagrange de segunda ordem [KN87]. Então, podemos gerar uma nova instância da classe `LocalLaplacian`, especial para este tipo de elemento:

```
class LocalLaplacian< LagrangeType2 > { // Especialização forçada.
   // Função que aproxima o Laplaciano, especial para Lagrange tipo 2.
   Submat eval() { ... }
};
```

3.6 Classes *traits* [Mye95]

A técnica de classes *traits*[5] não está diretamente associada ao desempenho, mas sim à manutenção da generalidade do código e à "colagem" das técnicas anteriores. De fato, podemos até dizer que a técnica de classes traits aumenta a generalidade inerente à OO em C++.

Suponhamos, a título de exemplo, que vamos criar uma classe que execute um determinado tipo de transformação matemática (há outras formas de se obter o mesmo resultado neste caso, mas exemplos são exemplos). Suponhamos também que esta transformação possa ser aplicada a um único ponto, ou a um conjunto de pontos, armazenados em um vetor. Respectivamente, a imagem será um único ponto, ou um conjunto de pontos.

Notemos que, neste caso, a entrada e a saída da transformação podem variar (ponto ou conjunto de pontos). Como lidar de forma geral com este problema? É importante observarmos que o polimorfismo e a sobrecarga, vistos na introdução (capítulo 2), resolvem apenas parte do problema (além das eventuais penalidades no tempo de execução). A maneira eficiente e geral de resolver este problema é usando o conceito de *classes traits*. Este tipo de classe nada mais é do que uma forma de se "carregar junto" um conjunto de definições *que dependem de um parâmetro template*. Vamos acompanhar o exemplo passo a passo.

Primeiramente, vamos definir a classe traits para nosso exemplo.

```
struct PontualTransformation {}; // Estrutura vazia usada como marcador.
struct VectorTransformation {}; // Idem.
```

[5] Termo técnico, sem uma boa tradução ainda. Será abolido o itálico.

```
// A classe traits.
template< class Structure >
class Transformation_traits {}; // Também vazia!

// E suas especializações, necessárias para decidir qual
// o tipo de dado útil em cada caso.

class Transformation_traits< PontualTransformation > { // Especialização 1.
  public:
    typedef double InputType; // Tipo de entrada.
    typedef double OutputType; // Tipo de saida.
};

class Transformation_traits< VectorTransformation > { // Especialização 2.
  public:
    typedef Vector InputType; // Tipo de entrada.
    typedef Vector OutputType; // Tipo de saida.
};
```

Uma observação muito delicada deve ser feita em relação aos diversos **typedef**'s encontrados na área pública da classe. O nome *metaprograma* mencionado anteriormente significa que estamos, em última instância, fazendo *programas que geram outros programas*. Para tanto tratamos a todo momento com *tipos* e não somente variáveis como num programa usual. Estes tipos necessitam — eventualmente — serem feitos visíveis para outras classes. Chamamos a esse processo de *exportação de tipos*.

Daí o fato de `InputType` e `OutputType` serem **typedef**'s na área pública da classe. Isto será muito importante no passo de definição da classe de transformação.

Outro fato a que devemos chamar a atenção são as estruturas vazias `PontualTransformation` e `VectorTransformation`. Estas estruturas (poderiam ser classes também) servem como *classes marcadoras*[6], cuja única função é servir como instrumento de especialização, como ficará claro mais adiante.

```
// A transformação - note que ela depende de um parâmetro template.
template< class Structure >
class Transformation
{
  public:
    // Definindo os tipos de dados, com base no tipo definido
    // na classe traits.
    typedef typename
      Transformation_traits< Structure >::InputType InputType;
    typedef typename
      Transformation_traits< Structure >::OutputType OutputType;

    // A operação matemática em si.
    OutputType eval( const InputType& in ) { ... };
};
```

Notemos como a função membro `eval` tem seus tipos de entrada e saída definidos de acordo com os tipos em `Transformation_traits`. Acompanhemos o programa principal:

[6] Do Inglês, *tag classes*.

```
main() {
   // Uma transformação de um único ponto.
   Transformation< PontualTransformation > T;

   double x = 3.14159, y;

   y = T.eval( x );

   // E uma vetorial.
   Transformation< VectorTransformation > T2;

   Vector a( 100 ), b( 100 );

   b = T2.eval( a );
}
```

Como podemos observar, declaramos dois objetos do tipo `Transformation`, diferenciando-os através das classes `PontualTransformation` e `VectorTransformation`. No primeiro caso avaliamos a transformação, aqui representada por `T` em um valor real `x` e devolvemos o resultado, também real, em `y`. Já a transformação `T2` recebe um vetor de 100 componentes e retorna um vetor transformado de 100 componentes.

Parte II

Forja

CAPÍTULO 4

A forja de objetos

No capítulo 2 mostramos quais as principais deficiências de C++ quando de sua utilização como linguagem para Computação Científica. Apresentamos na seqüência algumas técnicas usando templates que serão usadas para superar essas limitações. Assim, neste capítulo, abordaremos técnicas especiais usadas na construção de objetos mantendo a eficiência e expandindo as capacidades de generalização da linguagem.

4.1 Introdução

Os conceitos apresentados na seção 2.1 formam a base da OO, constituindo, por si mesmos, uma ferramenta excepcionalmente poderosa. A tecnologia das seções 3.2, 3.3, 3.4 e 3.5, respectivamente templates, metaprogramação, expressões templates e especialização, adicionam algum tipo de generalidade extra à linguagem (em particular os templates e a especialização), mas sua principal razão de ser é o desempenho. O uso de templates e da metaprogramação formam aquilo que se convencionou chamar *Programação Genérica* (vide [VG98]).

A pergunta que fica é: "o que pode ser feito em relação à generalidade padrão oferecida pela OO, em um contexto similar ao que foi feito em relação ao desempenho?" Ou ainda, que outras tecnologias em OO podem ser usadas de forma a flexibilizar a construção de objetos e ainda assim manter o desempenho e a facilidade de manutenção/extensão de código? É o que veremos nas próximas seções, quando introduzimos (de forma bastante superficial) o conceito de Programação Orientada a Aspectos, classes mixin (essenciais na construções de objetos) e meta construções, que são a base para a criação automatizada de objetos.

4.2 Programação Orientada a Aspectos (OA)

OO é, além de uma filosofia de programação, uma técnica de projeto de sistemas. Isso significa que o projeto de uma determinada aplicação computacional (científica ou não) seguirá um caminho mais ou menos padrão. Serão identificadas as unidades funcionais do problema, a cada unidade serão associadas uma ou mais classes, e a aplicação final consistirá da orquestração de objetos dessas classes executando a tarefa especificada.

Esta capacidade de tratar partes importantes de uma aplicação, uma de cada vez, é denominada *separação de interesses* [CE00] e é um princípio fundamental na construção de um software.

A separação de interesses é realizada *em parte* pela OO mas, de acordo com [KLM+97], algumas decisões no projeto de uma aplicação são difíceis de serem representadas de forma clara no código. Essas decisões que se *entrecruzam*[1] são denominadas *aspectos*. Mais precisamente, e ainda de acordo com [KLM+97], dizemos que uma propriedade ou decisão de projeto é um:

Componente se pode ser claramente encapsulado em uma unidade funcional. Componentes tendem a ser as unidades resultantes da decomposição de um sistema. Por exemplo, temos classes de matrizes, vetores, métodos numéricos, etc;

Aspecto no caso contrário. Aspectos não são usualmente unidades da decomposição de um sistema, mas sim propriedades que afetam o desempenho ou semântica dos componentes. Como exemplos temos padrões de acesso de memória, para otimizar o uso da memória cache, fusão de loops, como no caso de expressões templates — seção 3.4, e sincronização de objetos na paralelização de algoritmos.

A figura 4.1, reproduzida de [CE00], ilustra graficamente os conceitos de componente e aspecto.

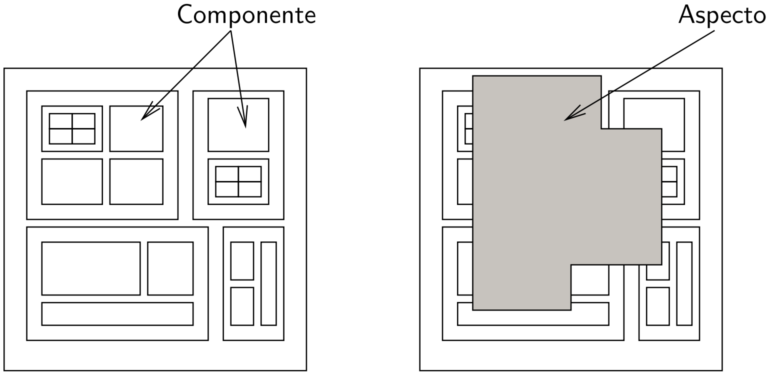


Figura 4.1: Componentes e aspectos.

O aspecto primordial que emerge na construção de aplicações científicas é o da otimização de código, tanto do ponto de vista da utilização da memória quanto do tempo de execução. Como esses aspectos entrecruzam praticamente todos os componentes deste tipo de aplicação, a tendência é a de desenvolvermos o software tomando como guia o desempenho. Como resultado, terminamos com um código extremamente eficiente, porém desnecessariamente complexo e particularmente difícil de manter, atualizar e estender.

Voltemos ao exemplo da seção 3.4. Como foi dito, as expressões templates vêm resolver o problema dos temporários e da multiplicação de loops. Uma outra solução — extremamente inadequada — seria a de definir diversas funções como na listagem 4.1. Notemos que ela resolve o problema: cada função tem apenas um loop e nenhum temporário. Porém a generalidade se perde, tanto pelo acúmulo de funções que essencialmente executam a mesma tarefa — somar vetores — como pela necessidade de adicionar novas funções.

[1] Termo técnico. Do Inglês, *cross-cut*.

```
// Faz z = x + y.
sum2( const Vector& x, const Vector& y, Vector& z );

// Faz w = x + y + z.
sum3( const Vector& x, const Vector& y, const Vector& z, Vector& w );

// E assim por diante ...
```

Listagem 4.1: Solução forçada para a eliminação de loops e temporários.

Portanto, as expressões templates realizam efetivamente o *aspecto otimização*, mantendo a generalidade do código. Em [CE00] a programação OA é abordada em profundidade, mas para os propósitos deste texto estamos interessados apenas em uma técnica lá apresentada para a realização da OA: a técnica de classes *mixin*.

4.3 Classes mixin

4.3.1 Introdução

As classes mixin[2] adicionam uma possibilidade interessante à construção de objetos (ver [Sma99], ou ainda [SB00]). Cada mixin representa uma funcionalidade ou aspecto que é *adicionado* ou *mudado* no objeto primordial.

A classe mixin é uma forma de composição de classes através de herança, onde a *classe ancestral é um parâmetro template da classe derivada*. Em outras palavras, *heranças também podem ser parametrizadas*. A listagem 4.2 exemplifica sintaticamente esta construção.

```
template< class SuperClass > // A super classe é um template...
class SubClass : public SuperClass { // e aparece aqui como classe ancestral.
  // Implementação da classe.
};
```

Listagem 4.2: Construção de uma classe mixin.

Esta construção não se limita a um único "andar" na hierarquia: podemos construir mixins com diversas classes intermediárias. Acompanhemos o exemplo da listagem 4.3.

Notemos a existência de uma última classe não mixin (`Last`). Esta classe serve para encerrar a hierarquia pois não é derivada de nenhum parâmetro template, mas poderia ter parâmetros templates convencionais. Esta classe será denominada classe *terminal* do mixin. O diagrama da figura 4.2 ilustra esta hierarquia.

O poder contido neste tipo de construção é enorme. Se o conjunto de classes for bem planejado — tanto as classes mixin quanto as classes terminais — podemos *agregar* ou *alterar* comportamentos padrão sem a necessidade da reconstrução da classe terminal.

[2] Termo técnico. Sem uma boa tradução ainda.

```
template< class Super1 > // Super classe 1.
class Sub1 : public Super1 { ... }

template< class Super2 > // Super class 2.
class Sub2 : public Super2 { ... }

template< class Super3 > // Super class 3.
class Sub3 : public Super3 { ... }

class Last { ... } // Última classe não mixin.

// A composição dessas classes é feita assim:

main() {
   Sub1< Sub2< Sub3< Last > > > Object;
}
```

Listagem 4.3: Contrução de classes mixin em vários níveis.

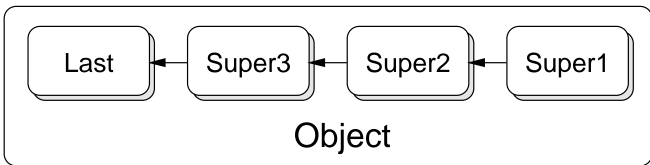


Figura 4.2: Vários níveis mixin.

4.3.2 Classes mixin como modificadoras

As listagens 4.4 e 4.5 mostram uma possível implementação de uma classe que calcula $\int_{-1}^{1} f(x)\,dx$ numericamente através de integração gaussiana com três pontos [Cun00]. Alguns detalhes não relevantes para a discussão foram omitidos do programa.

```
    // Classe para a função a ser integrada.
    class f {
3     public:
        // Requerimento:  possuir uma função eval com entrada e saída tipo double.
        double eval( double x ) const
6       {
          return x * x;
        }
9   };

    // Classe de pontos e pesos para a integração gaussiana.
12  class PW {
      public:
        // Requerimento:  exportar como N o número de pontos.
15      enum { N = 3 };

        // Requerimento:  prover métodos de acesso p() e w() para pontos e pesos.
```

```
18        double p( int ii ) const { return pts[ ii ]; }
          double w( int ii ) const { return wgs[ ii ]; }

21      private:
          static const double pts[ N ]; // pontos
          static const double wgs[ N ]; // pesos
24    };

      const double PW::pts[ PW::N ] = { -sqrt( 0.6 ), 0.0, sqrt( 0.6 ) };
27    const double PW::wgs[ PW::N ] = { 5.0/9.0, 8.0/9.0, 5.0/9.0 };


      template< class Function_t, class PW_t >
      class Gaussian {
30      public:
          Gaussian( const PW_t& pwModel ) : pw( pwModel ) {}

33        double eval() const {
            double Sum = 0.0;

36          Function_t F; // Instância da função a ser integrada.

            // Integrando.  Notemos o uso do requisito sobre N.
39          for( int ii = 0; ii < PW_t::N; ii++ ) {
              // Aqui aparecem os requisitos sobre eval() e sobre p() e w().
              Sum += F.eval( pw.p( ii ) ) * pw.w( ii );
42          }

            return Sum;
45        }

        private:
48        PW_t pw;
      };
```

Listagem 4.4: Classes para integração gaussiana.

Aproveitamos a chance para introduzir o conceito de *requisitos de uma classe*. Para o bom funcionamento de um programa baseado em templates é necessária a observância de determinados requisitos para cada classe passível de se tornar um parâmetro de outra classe. Ainda que sintaticamente um parâmetro template possa ser qualquer coisa, semanticamente ele tem de fazer sentido para a aplicação onde está inserido. Portanto a função a ser integrada necessariamente tem de ter um método `eval()`, recebendo um **`double`** e retornando um **`double`**. Caso algum requisito não seja seguido, a compilação falhará.

Suponhamos primeiramente que os elementos da listagem 4.4 não possam ser alterados. Esta hipótese não é irreal, posto que em alguns casos não se possui o código fonte e sim uma biblioteca pré-compilada. A pergunta é: de que forma podemos aproveitar o código já existente, sem reescrever tudo desde o princípio e ainda ampliar o espectro de uso dessa nossa integração gaussiana (por exemplo, calculando a integral em um intervalo $[a, b]$ e não só em $[-1, 1]$)?

```
main() {
   PW pw;  // Declarando pontos e pesos.

   Gaussian< f, PW > G( pw ); // Objeto que faz a integração.

   cout << "Integral␣[-1,1]␣=␣" << G.eval() << endl;
}
```

Listagem 4.5: Programa principal de integração gaussiana.

A resposta: através de uma classe mixin! Usaremos uma classe mixin para se interpor entre a classe `PW`, que tem os pontos de integração e os pesos padrão em $[-1, 1]$ e a classe `Gaussian`, que de fato executa a operação de integração. A "cirurgia" no programa das listagens 4.4 e 4.5 está esquematizada no diagrama da figura 4.3. Lembramos que a notação com flecha tracejada indica parâmetro template e flecha cheia herança. Do lado esquerdo temos a hierarquia original e do direito a nova, com a introdução do mixin.

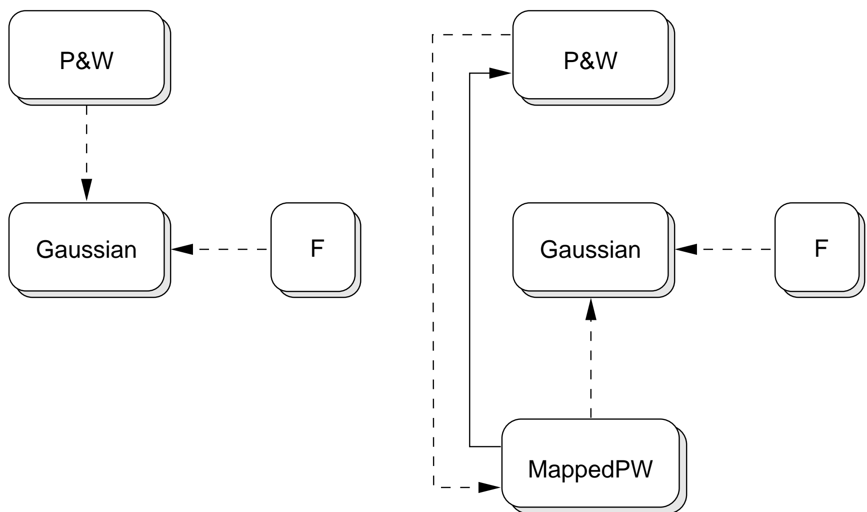


Figura 4.3: Nova configuração de classes com introdução da classe mixin.

No novo projeto a classe `MappedPW` é posicionada entre a de pontos e pesos original (`PW`) e a classe `Gaussian`. A listagem 4.6 esclarecerá o que foi dito e a listagem 4.7 o novo programa principal.

O que ocorreu?

A classe `Gaussian` efetua seus cálculos baseada em duas coisas:

1. A existência de um objeto com um método `eval()`;
2. Um objeto com métodos `p()` (pontos) e `w()` (pesos).

O item 1 permanece inalterado. O item 2, no caso original ainda sem uma classe mixin, é realizado pela classe `PW`. Para podermos efetuar a integração gaussiana em um intervalo diferente, é

```
// A classe mixin que altera o comportamento da classe PW.
template< class SuperClass >
class MappedPW : public SuperClass {
  public:
    MappedPW( double a1, double b1 ) : a( a1 ), b( b1 ) {}

    double p( int ii ) const { // Mapeando os pontos.
      return ( (b-a) * SuperClass::p( ii ) + (b+a) ) / 2.0;
    }

    double w( int ii ) const { // Corrigindo os pesos.
      return (b-a) * SuperClass::w( ii ) / 2.0;
    }

  private:
    double a, b;
};
```

Listagem 4.6: Classe mixin `MappedPW`.

```
main() {
  // Classe mixin, baseada em PW.
  MappedPW< PW > pw_mapped( -2, 3.5 );

  // O objeto, agora com o mixin.
  Gaussian< f, MappedPW< PW > > Gmapped( pw_mapped );

  // E o cálculo de ∫_{-2}^{3.5} x^2 dx.
  cout << "Integral [-2,3.5] = " << Gmapped.eval() << endl;

  return 0;
}
```

Listagem 4.7: Novo programa principal.

necessária a transformação desse intervalo, acompanhada da respectiva correção através do jacobiano dessa transformação.

Fazemos a classe `MappedPW` receber com parâmetro template a própria classe `PW` e ser derivada dela (classe mixin). Dessa forma, métodos e dados de `PW` ficam visíveis a `MappedPW`. Portanto, pontos e pesos podem ser transformados conforme o necessário, *como se fossem originalmente da classe `MappedPW`*. O que resta fazer é manter os requerimentos satisfeitos: `MappedPW` possui métodos `p()` e `w()`, exatamente como esperado por `Gaussian`, só que operando de forma diferenciada.

Para finalizar este exemplo, observamos que mesmo o requerimento sobre a função integranda (possuir um método `eval()`) pode ser relaxado. Suponhamos uma outra função cuja avaliação seja feita por um método `calculate()` e não `eval()`. Esse fato isolado já é suficiente para invalidar nosso projeto. Mas uma classe mixin resolve o problema. Vejamos a listagem 4.8. A utilização dessa nova classe mixin pode ser vista em 4.9.

```
// Outra função a ser integrada.
class g {
  public:
    double calculate( double x ) { return x * x * x; }
};


// Mixin para transformar calculate em eval.
template< class Function_t >
class Eval {
  public:
    double eval( double x ) {
      // Aqui é feita a ''transformação'' de calculate() em eval().
      return Function_t().calculate( x );
    }
};
```

Listagem 4.8: Usando uma classe mixin para satisfazer um requerimento.

```
main() {
  // Classe mixin, baseada em PW
  MappedPW< PW > pw_mapped( -2, 3.5 );

  // O objeto, agora com dois mixins
  Gaussian< Eval< g >, MappedPW< PW > > Gg( pw_mapped );

  // E o cálculo de ∫_{-2}^{3.5} x^3 dx
  cout << "Integral␣[-2,3.5]␣=␣" << Gg.eval() << endl;

  return 0;
}
```

Listagem 4.9: Usando a classe mixin para a função a ser derivada.

O fato mais relevante é que *nenhuma mudança foi feita* em `PW` ou mesmo em `Gaussian`! As classes originais permaneceram intocadas.

4.3.3 Classes mixin para a construção de objetos

Na seção anterior usamos as classes mixin para alterar o comportamento padrão de um objeto. Os métodos `p()` e `w()` foram reescritos para atender a uma necessidade, mas nada novo foi introduzido.

O exemplo que apresentaremos agora é o de *construção de novos objetos* através de classes mixin. Inicialmente definiremos um objeto que avalia uma determinada função, para posteriormente introduzirmos derivadas neste objeto, tanto calculadas analiticamente quanto calculadas numericamente.

Como são construídos objetos na listagem 4.10? A classe `Function` nada apresenta de especial. Ela possui um único método `eval()` que a avalia em um ponto dado.

Já as classes `NDerivative` e `Derivative` são classes mixin. Notemos que seu parâmetro template

```
// Esta é a função de interesse f(x) = sin x cos x.
struct Function {
  inline double eval( double x ) { return sin( x ) * cos( x ); }
};

// Esta classe implementa a derivada numérica (diferença centrada).
template< class F >
struct NDerivative : public F {
    inline NDerivative( double _h ) : h( _h ) {}

    inline double dx( double x ) {
      return ( eval( x + h ) - eval( x - h ) ) / ( 2 * h );
    }

    double h;
};

// E esta implementa a derivada analítica.
template< class F >
struct Derivative : public F {
    inline double dx( double x ) {
      return pow( cos( x ), 2 ) - pow( sin( x ), 2 );
    }
};

int main() {
  // Derivadas (numérica e analítica) de f(x) = sin x cos x.
  NDerivative< Function > df1( 0.1 );
  Derivative< Function > df2;

  // Ponto onde a função e as derivadas serão calculadas.
  double x = 0.5;

  // Avaliando a função...
  cout << df1.eval( x ) << endl;
  cout << df2.eval( x ) << endl;

  // E suas derivadas.
  cout << df1.dx( x ) << endl;
  cout << df2.dx( x ) << endl;

  return 0;
}
```

Listagem 4.10: Usando a class mixin para a função a ser integrada.

`F` é também classe base para elas. Se `F` representar uma função válida (ou seja, se satisfizer o requisito de possuir um método `eval()`), então `NDerivative` estará bem definida. Notemos que o método `dx()` de `NDerivative` faz menção única e exclusivamente a `eval()`.

Apesar de `Derivative` não possuir esta restrição, ainda sim o método `eval()` de `Function` pode ser acessado por causa da herança.

Para efeito de ilustração, é como se tivéssemos um objeto (no caso `Function`) e o recobríssemos com uma casca (`NDerivative` ou `Derivative`) que adiciona a funcionalidade necessária. Vale reforçar que `NDerivative` funciona para *qualquer classe que implemente o método `eval()` da mesma forma.*

4.4 Meta construções

4.4.1 Introdução

É interessante observarmos que gradualmente estamos caminhando na direção de uma forma paulatinamente mais sofisticada e automatizada de construção de objetos. Continuando desta forma, introduziremos o próximo ingrediente essencial na construção de objetos: as chamadas *meta construções*, ou simplesmente "metas".

As metas são as contra-partes templates de construções usuais como **`if`**, **`switch`**/**`case`** e mesmo as não tão usuais como listas ligadas. Com as metas, podemos tomar decisões *em tempo de compilação* dentro do programa, permitindo ampliar em muito as capacidades de um metaprograma ou de uma construção baseada em mixins.

4.4.2 Meta IF

A meta construção mais simples e também uma das mais úteis, é o chamado meta `IF` (listagem 4.11).

```
// O padrão é sempre retornar ‘‘true’’ (Then).
template< int condition, class Then, class Else >
struct IF {
   typedef Then RET;
};

// Especializamos o caso ‘‘false’’ (retornando Else).
template< class Then, class Else >
struct IF< 0, Then, Else > {
   typedef Else RET;
};
```

Listagem 4.11: Meta `IF`.

A estrutura `IF` é de fato bem simples. Ela possui três parâmetros templates; o primeiro é um inteiro, que assumirá valores zero (representando o booleano "false") ou diferente de zero (representando "true"). O segundo e terceiro são, respectivamente, os resultados caso a condição seja verdadeira ou falsa.

Note-se que implementamos o caso geral e especializamos (como descrito na seção 3.5.1) somente o caso em que o teste retorna falso. Por convenção, todo resultado de uma operação template é feito através do tipo `RET`.

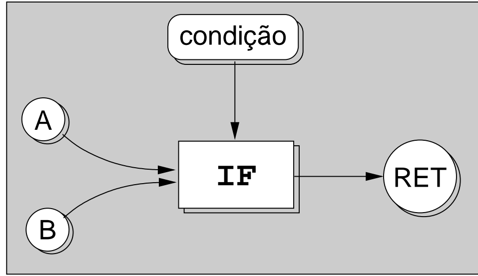


Figura 4.4: Esquematização do meta `IF`.

É muito importante destacar os seguintes pontos:

- A meta-construção `IF` é inteiramente avaliada *durante a compilação*;
- O resultado de sua avaliação (`RET`) *é um tipo válido* e serve para declarar um objeto.

Na figura 4.4 ilustramos a meta-construção `IF`. `A` e `B` são as condições `Then` e `Else` e `condição` é o booleano que orienta a escolha. O resultado final da avaliação do meta `IF` é `RET`. Para o compilador, toda a estrutura dentro do retângulo cinza é vista como uma única e bem definida classe.

Exemplo simplificado de meta IF

A listagem 4.12 traz o esqueleto de uma aplicação da meta-construção `IF`.

Embora o exemplo da listagem 4.12 seja bastante simples, serve para mostrar o uso da meta-construção `IF`. Primeiramente são declaradas duas classes matriciais:

- `Matrix`, representando uma matriz cheia e não-simétrica;
- `SymmMatrix`, representando uma matriz cheia, porém simétrica.

Em cada uma destas classes é declarada uma *enumeração anônima* [Eck95] chamada `IsSymmetric`, que sintaticamente age como uma constante (do tipo booleano), mas que funcionalmente serve como um micro repositório de configuração (vide capítulo 5, sobre objetos com DNA) com uma única informação.

A declaração chave para uso do `IF` é:

```
typedef IF< m.IsSymmetric, DecomposeSymmMatrix, DecomposeMatrix >::RET DecompM;
```

Vejamos por etapas o que ocorre durante a compilação.

1. Em primeiro lugar, o compilador avalia `m.IsSymmetric`;
2. Caso `m.IsSymmetric` seja verdadeiro (**`true`**), então `RET` será definido como o tipo `DecomposeSymmMatrix`;
3. Caso contrário, se `m.IsSymmetric` é falso (**`false`**), então `RET` será definido como `DecomposeMatrix`;
4. O resultado da decisão acima é definido como um tipo chamado `DecompM`, através de um **`typedef`**.

Em suma, na declaração acima `DecompM` equivale a `DecomposeSymmMatrix` ou a `DecomposeMatrix`: a decisão é tomada em tempo de compilação de acordo com o valor de `m.IsSymmetric`.

```
class Matrix { enum { IsSymmetric = false }; }; // Matriz cheia, não simétrica.
class SymmMatrix { enum { IsSymmetric = true }; }; // Matriz cheia, simétrica.

class DecomposeMatrix { // Decompõe uma matriz não-simétrica.
  void decompose( Matrix& m );
};

class DecomposeSymmMatrix { // Decompõe uma matriz simétrica.
  void decompose( SymmMatrix& m );
};

int main() {
  SymmMatrix m; // Simétrica.

  // Se m for simétrica, instancia DecomposeSymmMatrix; DecomposeMatrix caso contrário
  typedef IF< m.IsSymmetric, DecomposeSymmMatrix, DecomposeMatrix >::RET DecompM;

  DecompM D1;

  D1.decompose( m );

  SymmMatrix n;

  typedef IF< n.IsSymmetric, DecomposeSymmMatrix, DecomposeMatrix >::RET DecompN;

  DecompN D2;

  D2.decompose( n );
}
```

Listagem 4.12: Meta `IF`.

Um exemplo mais elaborado

Nesta seção apresentamos outro exemplo usando meta `IF`, desta vez voltada à construção de objetos com classes mixin. Conforme vimos na seção 4.3, podemos construir novos objetos através da composição de classes mixin. Apesar de se constituírem em instrumento extremamente flexível e poderoso, a principal desvantagem que se afigura quando do uso deste recurso é a de *permitir ao usuário a criação de combinações que não façam sentido*. É claro que em exemplos didáticos e pequenos como os apresentados até agora esta possibilidade torna-se remota, mas em grandes aplicações, com dezenas de possibilidades de classes mixin esta possibilidade começa a ganhar peso.

Desta forma seria bastante conveniente poder *decidir* — de alguma forma — quais combinações de classes mixin são válidas. Este assunto será explorado em profundidade no capítulo 5, mas já podemos introduzir algumas idéias. Voltemos ao exemplo dado na seção 4.3.3, onde mostramos como agregar funcionalidade (no caso derivadas) a uma classe pré-existente (que só avalia funções).

A idéia é simples: uma função pode ou não ter sua derivada analiticamente definida. O parâmetro booleano `hasDerivative` traz esta informação: caso o parâmetro seja verdadeiro, a função tem sua derivada analiticamente definida, caso seja falso, não. Se sua derivada estiver definida, nada é feito. Caso contrário, agregamos uma classe mixin que calcula a derivada numericamente.

O exemplo traz os dois casos. `Function1` tem a derivada definida analiticamente, e portanto o parâmetro `hasDerivative` é verdadeiro, ao passo que `Function2` não satisfaz esse requerimento; neste caso o **`enum`** `hasDerivative` é falso.

No primeiro caso, a declaração

```
typedef IF< Function1::hasDerivative, Function1, NDerivative< Function1 > >::RET DFunction1;
```

resulta na própria classe `Function1`, pois `hasDerivative` é verdadeiro. Portanto, o tipo `DFunction1` acaba sendo a classe `Function1`.

Já no segundo caso, a declaração

```
typedef IF< Function2::hasDerivative, Function2, NDerivative< Function2 > >::RET DFunction2;
```

resulta em `NDerivative< Function2 >`, dado que `hasDerivative` é falso. Desta forma, uma classe mixin foi usada para acrescentar comportamento a `Function2` (derivada numérica). Enfim, o tipo `DFunction2` possui o método `eval()` original de `Function2`, mais o método `dx()` oriundo da classe `NDerivative`.

Deve-se notar — com vistas a aplicações futuras — que características bastante peculiares ocorrem nos exemplos dados:

1. As classes "carregam" informações sobre si mesmas (e.g. `hasDerivative` nas classes de função);
2. Esta informação pode ser usada através de meta construções;
3. As classes originais podem ser modificadas ou acrescidas de funcionalidades (usando classes mixin) de acordo com suas informações internas.

```
struct Function1 { // Primeira função.
     enum { hasDerivative = true };

     inline double eval( double x ) { return sin( x ) * cos( x ); }
     inline double dx( double x ) { return pow( cos( x ), 2 ) - pow( sin( x ), 2 ); }
};

struct Function2 { // Segunda função.
     enum { hasDerivative = false };

     inline double eval( double x ) { return sin( x ) * cos( x ); }
};

template< class F > struct NDerivative : public F
{
     inline NDerivative() : h( 0.01 ) {}

     inline double dx( double x ) { return ( eval( x + h ) - eval( x - h ) ) / ( 2 * h ); }

     double h;
};

int main() {
   typedef
   IF< Function1::hasDerivative, Function1, NDerivative< Function1 > >::RET DFunction1;
   typedef
   IF< Function2::hasDerivative, Function2, NDerivative< Function2 > >::RET DFunction2;

   DFunction1 df1; // Objeto com derivada analítica.
   DFunction2 df2; // Objeto com derivada numérica.

   double x = 0.5;

   cout << df1.eval( x ) << endl;
   cout << df2.eval( x ) << endl;

   cout << df1.dx( x ) << endl;
   cout << df2.dx( x ) << endl;
}
```

Listagem 4.13: Meta `IF` usado com mixins.

4.4.3 Meta comparações

As meta comparações são construções template simples, que vêm imitar o comportamento dos operadores de comparação de igualdade (`==`), de desigualdade (`!=`), de menor (`<`) e de maior (`>`). A única diferença é que a comparação é exclusivamente entre números inteiros e constantes (ou seja, em tempo de compilação).

Na listagem B.2 do apêndice B, podemos apreciar a implementação das meta comparações mais comuns.

Checagem simples de erro através de meta comparações

Dado o caráter construtivo e auto-configurativo dos templates exibidos até agora, bem como sua semelhança com operações comuns a várias linguagens, é natural que mecanismos de checagem de erro possam ser implementados usando-se estas mesmas construções. O exemplo que apresentamos na listagem 4.14 ilustra este aspecto, porém checagens de erro em tempo de compilação realmente sofisticadas são descritas em [SL].

Na listagem 4.14 declaramos três estruturas (`Line`, `Triangle` e `Tetra`), cada uma correspondendo a um tipo (hipotético) de elemento finito. Note-se que cada uma delas está associada a uma dimensão espacial. Uma quarta estrutura chamada `WrongElement` é definida e será associada a um tipo "errado" de elemento finito. Por tipo errado definimos aquele que tem dimensão menor que 1 ou maior que 3. Todas as quatro estruturas possuem um método `hello()`, que é usado para identificação.

Usamos as meta comparações na classe `ElementSelector`. É ela que, dada uma dimensão espacial `D`, seleciona o tipo de elemento finito apropriado. O processo é simples: primeiro fazemos um teste usando as comparações de maior e menor (respectivamente `BT` e `LT`), checando se a dimensão fornecida não se encontra fora dos limites desejados. Se for este o caso, então o resultado da operação é a classe `WrongElement`. Caso contrário, vários teste aninhados são usados com a dimensão `D` com a finalidade de determinarmos qual o elemento apropriado. Notemos que, de fato, fazemos apenas dois testes e não três; um elemento finito do tipo `Tetra` é selecionado por default caso a dimensão espacial seja válida, porém diferente de 1 e de 2.

No programa principal declaramos cinco objetos usando `ElementSelector`, com dimensões variando de 0 a 4. Como esperado, os métodos `hello()` retornam mensagens de elemento inválido para dimensões 0 e 4, e as respectivas mensagens para as dimensões entre 1 e 3, inclusive.

4.4.4 Meta `SWITCH/CASE`

Uma construção bastante similar ao meta `IF` apresentado na seção anterior é a meta construção `SWITCH/CASE`. Ela imita o comportamento de um **`switch`**/**`case`** padrão de C++, porém permitindo a avaliação de tipos em tempo de compilação de uma maneira mais legível do que a que seria obtida usando-se `IF`'s aninhados. A listagem completa da implementação do meta `SWITCH/CASE` pode ser apreciada na listagem B.1, no apêndice B.

O uso da meta construção `SWITCH/CASE` pode ser visto na seqüência.

```
typedef SWITCH< Key, // Chave de seleção.
  CASE< Option1, Type1, // Opção 1.
```

```
struct Line {
    void hello() { cout << "Sou␣um␣elemento␣linear␣(1D)" << endl; }
};

struct Triangle {
    void hello() { cout << "Sou␣um␣elemento␣triangular␣(2D)" << endl; }
};

struct Tetra {
    void hello() { cout << "Sou␣um␣elemento␣tetraedrico␣(3D)" << endl; }
};

struct WrongElement {
    void hello() { cout << "NAO␣ME␣USE!␣ELEMENTO␣ERRADO!" << endl; }
};

template< int D >
struct ElementSelector {
  // Caso D < 1 ou D > 3, instancia WrongElement.
  typedef typename
  IF< LT< D, 1 >::RET | BT< D, 3 >::RET,
     WrongElement,
     typename IF< EQUAL< D, 1 >::RET, // Se D = 1, instancia uma linha (Line).
                 Line,
                 typename IF< EQUAL< D, 2 >::RET, // Ou um Triangle, se D = 2.
                             Triangle,
                             Tetra // Ou, finalmente, um Tetra se D = 3.
  >::RET >::RET >::RET RET;
};

int main()
{
  ElementSelector< 0 >::RET a;
  ElementSelector< 1 >::RET b;
  ElementSelector< 2 >::RET c;
  ElementSelector< 3 >::RET d;
  ElementSelector< 4 >::RET e;

  a.hello();  b.hello();  c.hello();  d.hello();  e.hello();
}
```

Listagem 4.14: Escolha de classes e checagem de erro através de meta comparações.

```
    CASE< Option2, Type2, // Opção 2.
    CASE< Option3, Type3, // Opção 3.
    DEFAULT< DefaultType > > > > // Opção default.
>::RET Result;
```

O meta `SWITCH/CASE` é portanto, composto por:

- Uma chave de seleção `Key`, que é um inteiro constante;

- Diversas opções, também inteiras e constantes, que serão comparadas com a chave;
- Associadas às opções, tipos — que podem ser classes definidas pelo usuário ou tipos padrão da linguagem — que serão selecionados em caso de coincidência entre a chave e as opções;
- Um tipo `Default`, que será selecionado caso nenhuma das opções case com a chave.

Devemos notar mais uma vez que *o resultado da operação é um dos tipos selecionados pelo meta*. Este tipo é retornado por padrão em `RET`.

Exemplo de `SWITCH/CASE`

Na listagem 4.15 apresentamos um exemplo simplificado de uso da meta construção `SWITCH/CASE`. Inicialmente, definimos o conjunto de opções válidas para o `SWITCH`. Estas opções, como no caso do `IF`, são representadas por inteiros e armazenadas na forma de uma enumeração anônima. Neste exemplo definimos três opções: `OptA`, `OptB` e `OptC`, que equivalem respectivamente aos inteiros 0, 1 e 2 (o valor dos inteiros não importa, devendo apenas ser diferentes e constantes do ponto de vista do compilador; a enumeração anônima é uma forma conveniente de estruturar este conjunto de inteiros).

São definidas também as estruturas ou classes que serão usadas como opções do `CASE`. Note-se que *qualquer tipo válido* da linguagem poderia ter sido usado, fosse ele padrão ou definido pelo usuário. Estas classes, criadas apenas com o intuito de melhor ilustrar o exemplo, são `A`, `B`, `C` e `Default`. Cada classe possui um método `hello()` que imprime na tela seu próprio nome. Naturalmente as classes `A`, `B` e `C` serão associadas aos inteiros (ou opções) `OptA`, `OptB` e `OptC`. `Default` é associada a qualquer caso diferente destes.

Desta forma, o tipo `Opt1_t` definido através do primeiro `SWITCH/CASE`, é exatamente equivalente à classe `A`, pois o primeiro `CASE` é satisfeito (opção `OptA`). Analogamente, `Opt2_t` e `Opt3_t` são equivalentes às classes `B` e `C`, respectivamente, ao passo que `Opt4_t` o é em relação à classe `Default`.

Instanciados os objetos (`op1` a `op4`), teremos a série de mensagens das estruturas `A` a `C` e também `Default` impressas na tela.

4.4.5 Outras meta construções

As meta construções apresentadas nas seções 4.4.2, 4.4.3 e 4.4.4 nem de longe esgotam as possibilidades das metas. Porém, é a partir delas que as outras serão construídas. Podemos pensar nestes metas como os primordiais. Maiores referências podem ser encontradas principalmente em [CE00], mas também em [CE01], [EBC00], [GDL00], [Vel95b] e [CE99].

```
enum { OptA, OptB, OptC }; // Marcadores de opções.

// Diversas possibilidades de seleção.
struct A { void hello() { cout << "Eu␣sou␣A!" << endl; } };
struct B { void hello() { cout << "Eu␣sou␣B!" << endl; } };
struct C { void hello() { cout << "Eu␣sou␣C!" << endl; } };
struct Default { void hello() { cout << "Eu␣sou␣o␣padrao!" << endl; } };

int main() {
   // Fazendo diversas seleções com SWITCH/CASE.
   typedef SWITCH< OptA, CASE< OptA, A,
                         CASE< OptB, B,
                         CASE< OptC, C,
                         DEFAULT< Default > > > > >::RET Opt1_t;

   typedef SWITCH< OptB, CASE< OptA, A,
                         CASE< OptB, B,
                         CASE< OptC, C,
                         DEFAULT< Default > > > > >::RET Opt2_t;

   typedef SWITCH< OptC, CASE< OptA, A,
                         CASE< OptB, B,
                         CASE< OptC, C,
                         DEFAULT< Default > > > > >::RET Opt3_t;

   typedef SWITCH< 15, CASE< OptA, A,
                       CASE< OptB, B,
                       CASE< OptC, C,
                       DEFAULT< Default > > > > >::RET Opt4_t;

   // Criando os objetos e chamando a função hello().
   Opt1_t op1;  Opt2_t op2;  Opt3_t op3;  Opt4_t op4;

   op1.hello();  op2.hello();  op3.hello();  op4.hello();
}
```

Listagem 4.15: Exemplo de meta `SWITCH/CASE`.

CAPÍTULO 5

Repositórios de configuração: objetos com DNA

Tendo assentado a base para a construção de objetos no capítulo 4, introduziremos agora o conceito de repositórios de configuração [CE99], [Cza00] e [CE00]. Através desta técnica podemos criar classes que encerram em si todas as informações de sua própria construção, como em uma molécula de DNA. Esta característica única será aproveitada na construção de novos objetos a partir de descrições de mais alto nível do que as já apresentadas.

5.1 Introdução

5.1.1 Repositórios de configuração

Uma parte relevante da pesquisa de Krzysztof Czarnecki e de Ulrich Eisenecker relaciona-se à possibilidade de criação de objetos usando a tecnologia apresentada no capítulo 4. O que diferencia o trabalho destes pesquisadores é que as características dos objetos a serem criados, e que são fornecidas pelo usuário, são de fato *aspectos do objeto final desejado*. Estas idéias foram sendo buriladas em [CE99], [Cza00] e culminaram em um livro inteiramente voltado à Programação Generativa [CE00].

Nós as exporemos brevemente pois serão o cerne do desenvolvimento do pacote Osiris de elementos finitos. Devemos notar entretanto que Osiris utiliza uma filosofia que, consideramos, é aprimorada em relação àquela apresentada em [CE00]: lá, o conjunto de funcionalidades que se deseja alcançar é estudado exaustivamente em todos os seus aspectos[1]. Esses aspectos são traduzidos em um *repositório monolítico* de possíveis configurações e composições destes objetos, bem como um repositório, também monolítico, de comportamentos ou operações.

Nossa abordagem é essencialmente diferente, pois:

- Usamos repositórios de configuração menores e mais localizados, evitando a necessidade de um

[1] No caso específico da referência citada, é apresentada uma biblioteca experimental de Álgebra Matricial chamada GMCL (Generative Matrix Computation Library). Considerando-se todas as combinações possíveis de tipos de matrizes, suas formas de armazenamento, esparsidade, GMCL pode gerar mais de 1700 tipos de matrizes.

estudo exaustivo de características antes de implementação efetiva. Este levantamento de características pode ser feito em grupos menores, reduzindo, portanto, a extensão desse estudo;

- Estes repositórios podem ser usados na construção de outros repositórios e assim por diante. O efeito final é o de um nível extra de abstração, pois podemos combinar configurações de objetos diferentes;
- Os repositórios de configuração, da forma como são implementados, podem ser facilmente estendidos dando conta de eventuais aspectos não previstos ou planejados, que surgem em uma implementação monolítica;
- As operações locais sobre os elementos finitos são realizadas através de especializações de um mecanismo particular de expressões templates, permitindo implementações praticamente independentes dos objetos configurados. O capítulo 6 é inteiramente dedicado a este assunto.

5.1.2 Um exemplo inicial

Com o propósito de ilustrar na prática o que é um repositório de configuração, mostraremos como definir um determinado tipo de elemento finito usando Osiris. O elemento será bidimensional, triangular, com base de Lagrange linear. A listagem 5.1 mostra como declarar um novo tipo chamado `TriangleP1_t` que corresponde ao elemento finito descrito.

```
typedef FE_GENERATOR< // Este é o gerador de elementos.
     FE_Family< lagrange< 1 > >, // Especificando a ordem de aproximação...
     FE_Geometry< triangle > // ...e a geometria.
>::RET TriangleP1_t; // Aqui temos o resultado final.
```

Listagem 5.1: Declarando um triângulo do tipo P1.

O interessante deste tipo de abordagem é que a *especificação da classe a ser construída é feita em alto nível*. Por exemplo, para definirmos uma geometria triangular no plano, usamos a sintaxe `FE_Geometry< triangle >`. Caso desejássemos um tetraedro com as mesmas características de aproximação, agora em R^3, bastaria a simples troca de geometria via uma declaração do tipo `FE_Geometry< tetra >`.

Devemos chamar a atenção também ao fato de que `TriangleP1_t` é uma classe *completamente funcional*, configurada pelas escolhas de ordem de aproximação `FE_Family` e de geometria `FE_Geometry`. Apesar da aparente simplicidade de escolhas, *uma série de outras informações é automaticamente criada e disponibilizada* dentro de `TriangleP1_t`, em tempo de compilação e sem intervenção do usuário. A tabela 5.1 resume o conteúdo do repositório de configuração final para `TriangleP1_t`.

5.2 Componentes para geração de configurações

Como vimos na seção anterior, podemos, a partir de uma descrição de alto nível, gerar uma classe completa que implementa as características desejadas. Este processo de configuração automática compreende os seguintes passos:

Característica	Nome	Em `TriangleP1_t`
Tipo de aproximação	`FE_Family_t`	`lagrange< 1 >`
Geometria	`FE_Geometry_t`	`triangle`
Tipo do indexador dos nós	`RefsContainerAtom_t`	**`int`**
Tipo do container de nós	`RefsContainer_t`	`Vector< 3, int >`
Ordem de aproximação	`order`	1
Dimensão geométrica	`dimensions`	2
Dimensão espacial	`spatialDimension`	2
Elemento interno ou fronteira	`role`	`full_element`
Graus de liberdade	`ndof`	3
Graus de liberdade na fronteira	`ndofb`	2

Tabela 5.1: Repositório de configuração para `TriangleP1_t`.

Análise de domínio (teórico) É onde estudamos os conceitos que serão modelados e onde ressaltamos suas características desejáveis. Por exemplo, no caso de elementos finitos, como descrito anteriormente, temos a FAMÍLIA (Lagrange, Hermite, etc), a GEOMETRIA (triângulo, quadrado, tetraedro, etc) e a DIMENSÃO ESPACIAL. Os aspectos levantados durante este estudo devem ser suficientes para a completa construção de qualquer dos conceitos que se deseje modelar.

DSL (teórico/prático) A DSL (Domain Specific Language) é a concretização dos aspectos levantados no item anterior. Nela são dados nomes às características levantadas previamente. No caso de nosso exemplo com elementos finitos, uma das definições da DSL é a de tipos válidos de geometria, como pode ser visto na listagem 5.2. Podemos notar que cobrimos todos os aspectos geométricos que mais nos interessam (ponto, linha, triângulo, etc). Conforme as necessidades do usuário, ou dentro da evolução natural do software, novos aspectos podem ser introduzidos sem o comprometimento dos anteriores.

```
enum GeomId // Identificadores de geometria para elementos finitos.
{
  point_id, // Elemento pontual.
  line_id, // Linha.
  triangle_id, // Triângulo.
  rectangle_id, // Retângulo.
  tetra_id, // Tetraedro.
  brick_id // Hexaedro.
};
```

Listagem 5.2: Parte de uma DSL para configuração de um elemento finito.

Parser (prático) O parser lê uma descrição fornecida pelo usuário e a processa de forma a gerar uma configuração básica. Este processo envolve checagem de erros [SL], decisão de opções padrão, caso o usuário deixe algumas opções em branco, e criação do repositório de configurações.

Geração (prático) Este é o último ponto da cadeia, onde a configuração produzida é analisada e as

partes corretas — usualmente classes mixin previamente preparadas — são montadas em conjunto para produzir um objeto funcional (processo de geração). É interessante observar que, se feito de maneira correta, este processo de produção embute a informação de configuração e o próprio gerador dentro da classe assim criada.

Antes de exemplificarmos este processo usando OSIRIS, introduziremos na seção seguinte um pouco da notação diagramática utilizada no processo de modelagem de características e na representação da DSL.

5.2.1 Diagramas de características

A descrição de DSL's pode ser feita de várias maneiras, mas com certeza uma das mais ilustrativas é através dos diagramas de características. Um *diagrama de características* é uma árvore onde:

- O nó raiz representa o conceito a ser modelado (representado por C);
- Cada nó da árvore é uma possível característica (ou aspecto) daquele conceito (representadas por f_i);
- O tipo de ramo diz a forma em que se percorre a árvore, resultando portanto em diferentes possíveis configurações, conforme o caso de interesse. Cada tipo de ramo possui uma representação gráfica diferente, como veremos à frente.

Um diagrama de características típico pode ser visto na figura 5.1.

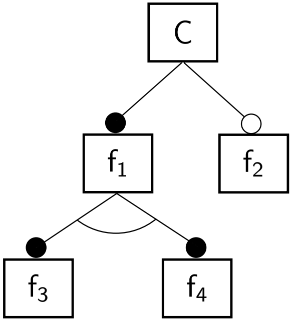


Figura 5.1: Exemplo de diagrama de características.

A tabela 5.2 resume os diagramas de características usados mais comumente.

Com estes diagramas básicos estamos aptos a descrever um exemplo de DSL para elementos finitos, tal qual implementada em OSIRIS.

5.2.2 DSL para Elementos Finitos

A DSL (Domain Specific Language), ou Linguagem para um Domínio Específico, serve para descrever sinteticamente o conjunto de informações que deve efetivar perfeitamente todas as possibilidades de configuração de que o usuário necessita. Por *possibilidades de configuração* queremos dizer *todos os*

Diagrama	Descrição	Configurações
Mandatória	São características imprescindíveis para a construção do objeto. Por exemplo, no caso de elementos finitos, uma geometria é indispensável. A representação de uma característica mandatória é feita através de um círculo cheio.	$\{C, f_1, f_2, f_3, f_4\}$
Opcional	São características que podem ou não estar presentes. No caso de elementos finitos, podemos dispor de uma base (elemento computacional) ou não (elemento geométrico). Sua representação é feita através de um círculo vazado.	$\{C\}$, $\{C, f_1\}$, $\{C, f_2\}$, $\{C, f_1, f_3\}$, $\{C, f_1, f_2\}$, $\{C, f_1, f_2, f_3\}$
Alternativa	As características alternativas combinam-se em grupos. São assinaladas por um arco ligando os ramos de cada grupo, indicando que pelo menos um deste grupo deve ser escolhido.	$\{C, f_1, f_3\}$, $\{C, f_1, f_4\}$, $\{C, f_1, f_5\}$, $\{C, f_2, f_3\}$, $\{C, f_2, f_4\}$, $\{C, f_2, f_5\}$
Tipo "ou"	É uma junção entre opcionais e alternativas. Neste caso todas as combinações contendo pelo menos um elemento de cado grupo são válidas. É representada por um arco cheio ligando os ramos.	$\{C, f_1, f_3\}$, $\{C, f_1, f_4\}$, $\{C, f_1, f_5\}$, $\{C, f_2, f_3\}$, $\{C, f_2, f_4\}$, $\{C, f_2, f_5\}$ $\{C, f_1, f_2, f_3\}$, $\{C, f_1, f_2, f_4\}$, $\{C, f_1, f_2, f_5\}$, etc

Tabela 5.2: Diagramas de características mais comuns.

diferentes objetos que podem ser instanciados e que fazem algum sentido para o usuário. Fazendo a ponte com os diagramas de características, o conceito C é parte do domínio (e.g. um elemento finito), enquanto que as características (ou aspectos) são representados pelos f_i.

No caso particular da configuração de um elemento finito, interessam-nos quatro aspectos, assim nomeados: FAMÍLIA, GEOMETRIA, DIMENSÃO ESPACIAL e PAPEL.

Estes quatro aspectos — FAMÍLIA, GEOMETRIA, DIMENSÃO ESPACIAL e PAPEL — se bem escolhidos, devem ser suficientes para a definição de todos os tipos de elementos desejados. Analisemos o papel de cada um na configuração.

Família O aspecto FAMÍLIA, em conjunto com sua ordem de aproximação, influencia na escolha e na definição do tipo de função de base (a ser gerada em outro configurador) e no número de graus de liberdade. Por exemplo, se escolhermos como FAMÍLIA `FE_Family< lagrange< 1 > >`, o configurador assume que o elemento suporta uma base de Lagrange de primeira ordem.

Geometria A GEOMETRIA define a forma do elemento e influencia, conforme já assinalado, em outro configurador, na transformação geométrica entre elemento padrão e o real. As formas padrão são linhas (1D), triângulos e quadriláteros (2D), tetraedros, hexaedros e prismas (3D). É interessante neste ponto observarmos a flexibilidade desta abordagem template. Suponhamos que no futuro

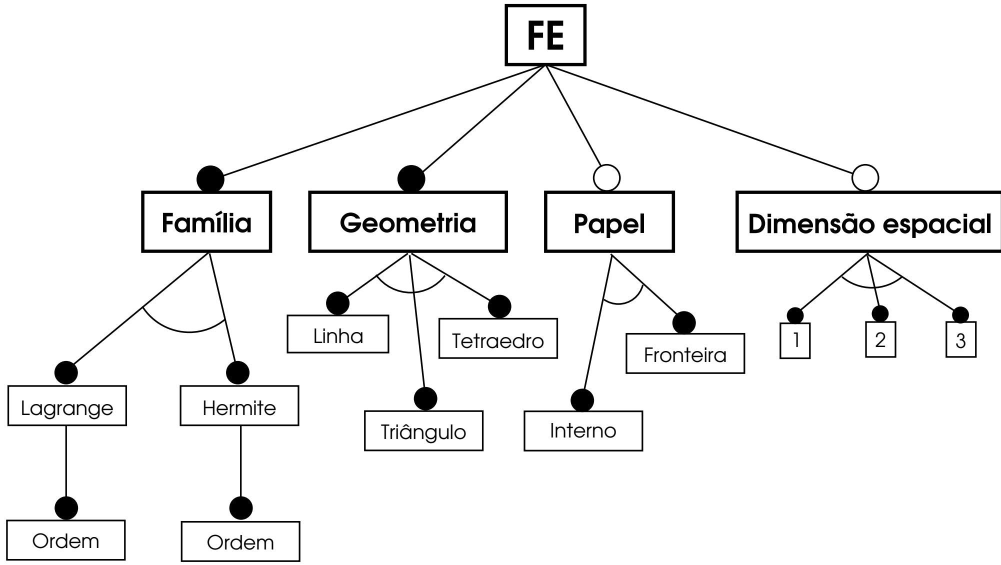


Figura 5.2: DSL para elementos finitos.

desejássemos suportar elementos curvos. O aspecto GEOMETRIA poderia ser decomposto em dois subaspectos: GEOMETRIAS PARAMÉTRICAS (elementos com lados ou faces retos) e GEOMETRIAS SUPER-PARAMÉTRICAS (elementos com lados ou faces curvos). Poucas mudanças no configurador absorveriam este novo requerimento, pois as características já existentes permaneceriam iguais, e essas novas características poderiam ser introduzidas de acordo com a necessidade de momento. Como exemplo de GEOMETRIA, temos `FE_Geometry< triangle >`. Em conjunto com a FAMÍLIA definida acima, temos um elemento triangular, de Lagrange, com aproximação polinomial de primeira ordem. Com estes dois aspectos escolhidos, o configurador já pode, por exemplo, decidir o número de graus de liberdade (no nosso exemplo, três, um para cada vértice do triângulo). Uma escolha diferente de geometria resultaria em um número diferente de graus de liberdade (quatro, caso escolhêssemos `FE_Geometry< tetra >`). Idem para uma escolha diferente de família (por exemplo, seis, para `FE_Family< lagrange< 2 > >`).

Papel Um elemento pode ter dois papéis bem definidos: pode ser um elemento do interior do domínio ou um elemento de fronteira. Este aspecto é opcional, como podemos ver do diagrama, o que significa que o configurador tem de possuir alguns padrões pré-definidos. No caso do aspecto PAPEL, o configurador escolhe por padrão `FE_Role< full_element >`, que corresponde a um elemento interno ao domínio.

Dimensão espacial Apesar de aparentemente redundante ou irrelevante (a `geometria` automaticamente nos dá uma dimensão espacial), podemos necessitar definir elementos bidimensionais no espaço tridimensional (como elementos de casca). O papel do aspecto DIMENSÃO ESPACIAL é justamente fornecer esta informação extra necessária. Caso não seja fornecida, a DIMENSÃO ESPACIAL é assumida igual à dimensão geométrica.

5.3 Detalhes de implementação do configurador

Examinaremos na seqüência alguns dos detalhes de implementação do configurador de elementos finitos usado em Osiris.

5.3.1 A descrição da DSL

A descrição computacional da DSL para um determinado conjunto de aspectos é feita utilizando-se uma técnica template denominada *parâmetros nomeados* [CE01].

Inicialmente foi criada uma enumeração anônima com um valor para cada aspecto descrito acima (listagem 5.3).

```
enum FE_Parameters_ {
  fe_Family_id, // Aspecto FAMÍLIA.
  fe_Geometry_id, // Aspecto GEOMETRIA.
  fe_SpatialDimension_id, // Aspecto DIMENSÃO ESPACIAL.
  fe_Role_id  // Aspecto PAPEL.
};
```

Listagem 5.3: Enumeração anônima com um enumerado para cada aspecto.

Tendo especificado os aspectos gerais da DSL, iremos agora esmiuçar cada um deles. Começando pelo aspecto Família, criamos outra enumeração anônima com os identificadores para cada possível membro do aspecto Família (listagem 5.4)

```
enum FeId {
  lagrange_id, // Lagrange.
  hermite_id // Hermite.
};
```

Listagem 5.4: Identificadores para tipos de Família.

É interessante observarmos que já neste ponto é introduzida uma flexibilidade relevante, pois as enumerações apresentadas nas listagens 5.3 e 5.4 podem ser acrescidas de novos elementos sem alterações nos já existentes.

O próximo passo consiste em especificarmos de modo ainda mais preciso cada uma das características nomeadas em 5.4. Para alcançarmos este propósito, definimos classes (ou, nos casos mais simples, estruturas) que contenham, obrigatoriamente, um membro de uma união anônima chamado `id`. Como podemos ver na listagem 5.5, este `id` recebe exatamente o valor definido na lista de identificadores; no exemplo, escolhemos `id` igual a `lagrange_id`. Estas classes ou estruturas são denominadas *descritoras*. Caso seja necessário ou conveniente, outras informações podem ser adicionadas às descritoras. No nosso exemplo, incluímos também a ordem de aproximação (`order`) e uma cadeia de caracteres para identificar a classe por nome (`FamilyName`).

Por fim, basta o último elo da corrente: a classe responsável por carregar e nomear a informação. Na listagem 5.6 temos a descrição da classe `FE_Family`, que é derivada de uma classe especial de-

```
template< uinteger_t order_ = 1 >
struct lagrange {
     // Ordem de aproximação e identificador.
     enum { order = order_, id = lagrange_id };

     // Nome da família.
     static const char* FamilyName;
};

template< uinteger_t order_ >
const char* lagrange< order_ >::FamilyName = "lagrangean";
```

Listagem 5.5: Definição da classe descritora `lagrange`, do aspecto FAMÍLIA.

nominada `TypeTemplateParam`. A class `TypeTemplateParam` age como um marcador, associando o parâmetro template `feFamily_t` ao respectivo identificador `fe_Family_id`, definido em 5.3. Ou seja, o aspecto escolhido efetivamente (`feFamily_t`) é aqui associado ao seu aspecto abstrato (através de `fe_Family_id`). Mais adiante veremos em detalhes a classe `TypeTemplateParam`. Aqui, basta-nos lembrar que ela associa um tipo a um identificador.

```
template< class feFamily_t >
class FE_Family :
     public TypeTemplateParam< feFamily_t, fe_Family_id > {};
```

Listagem 5.6: Classe marcadora `FE_Family`.

Podemos perceber que uma declaração aparentemente simples como `FE_Family< lagrange< 1 > >` cria, durante a compilação, uma série de pequenos objetos relacionados. A figura 5.3 ilustra como ficam os objetos quando instanciamos `FE_Family< lagrange< 1 > >`. Lá, podemos claramente ver quem é objeto template de quem (os enumerados estão em cinza claro).

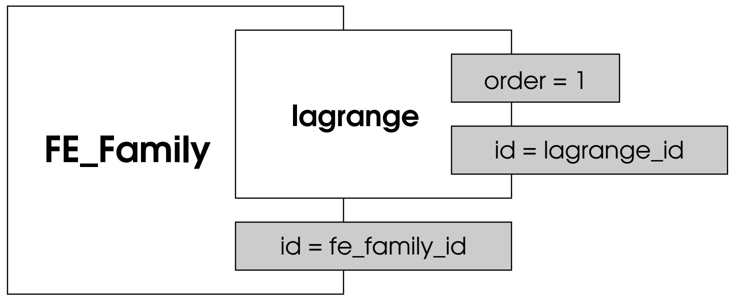


Figura 5.3: Instância de `FE_Family< lagrange< 1 > >`.

Os outros aspectos para criação de um determinado elemento finito são definidos de maneira análoga.

5.3.2 Processamento de configurações (parsing)

O processamento de configurações (doravante denominado *parsing*) é, como não poderia deixar de ser, feito através de mecanismos template. Como mencionado no capítulo 4, podemos replicar estaticamente diversas estruturas computacionais, como **`if`**, **`switch`** e listas ligadas. A estrutura subjacente ao processo de parsing é exatamente uma lista ligada template de configurações, de onde serão extraídas as informações pertinentes.

Esta lista possui as seguintes características:

- É uma lista que armazena *tipos*, e não objetos;
- Seus elementos são inteiramente definidos *em tempo de compilação*;
- É *completamente heterogênea*, podendo armazenar um tipo diferente em cada posição.

Não entraremos nos detalhes sobre a implementação da lista em si, que pode ser conferida em [CE01] ou nas listagens anexas, mas devemos ter em mente sua função na criação de objetos. Para firmarmos o conceito, voltemos à listagem 5.1. Nela fornecemos dois aspectos para o configurador: `FE_Family< lagrange< 1 > >` e `FE_Geometry< triangle >`. Estes dois aspectos serão inseridos em uma lista ligada estática e passados ao parser. É interessante observarmos que a DSL para um elemento finito, conforme definida na seção 5.2.2, comporta quatro aspectos diferentes, enquanto estamos fornecendo apenas dois. Os aspectos que não são utilizados (no caso, PAPEL e DIMENSÃO ESPACIAL) são inseridos como objetos nulos na lista.

A listagem da classe que faz o parsing para elementos finitos pode ser apreciada em 5.7.

O processo de parsing pode ser descrito como segue:

- É recebida uma lista template `ParamList_t`, onde cada elemento é um tipo relacionado a um aspecto (por exemplo, `FE_Geometry< tetra >`). *Cf* linha 1 da listagem 5.7.
- A lista é vasculhada usando-se a classe `FindWithId`. Esta classe recebe a lista com os aspectos e um identificador determinado. A lista é inteiramente varrida e seus elementos têm seus respectivos identificadores comparados com o fornecido pelo usuário. Caso haja coincidência, a classe dona do identificador procurado é retornada. Em suma, é feita uma busca na lista (linhas 5–8). Voltando ao nosso exemplo, no caso específico da linha 5, o tipo `FE_Family_p` seria `lagrange< 1 >`, como havíamos escolhido previamente. Na linha seguinte, o tipo `FE_Geometry_p` seria `triangle`.

  Caso um determinado tipo seja procurado na lista e não seja encontrado, é retornado o tipo `ListNil`.
- Um dos cuidados que devemos tomar na criação de configuradores automatizados é com as opções padrão, assumidas caso o usuário não opte por algum aspecto obrigatório, ou ainda, opte errado. As linhas 10–11 mostram a ação da classe `GetTypeOrDefault`, que trata o este caso. Imaginemos que por uma distração o usuário tenha se esquecido de fornecer o aspecto FAMÍLIA. Neste caso, como dissemos no item anterior, `FE_Family_p` assumirá o tipo `ListNil`. O que `GetTypeOrDefault` da linha 10 faz é, caso o tipo fornecido seja `ListNil`, seja assumido um tipo padrão (`lagrange< 1 >` no nosso exemplo). Naturalmente poderíamos ter optado por gerar um erro em tempo de compilação, mas esta saída é mais prática e elegante.

- A enumeração anônima que vem na seqüência extrai informações secundárias (não definidas explicitamente pelo usuário), como a ordem de aproximação (`_FE_Family_t::order`) ou o número de dimensões geométricas (`_FE_Geometry_t::dimensions`). Notemos a presença de uma classe similar a `GetTypeOrDefault`, denominada `GetIntOrDefault`. Sua função é exatamente a mesma, porém trabalhando com valores enumerados.

  Nas linhas seguintes de 5.7 são também decididos o número de graus de liberdade (`LagrangeanDOF`[2]) e o tipo de container que abrigará os nós do elemento finito (`Vector< ... >`). Como podemos notar, o tipo `DOF` criado a partir de `LagrangeanDOF`, já tem definido o número de graus de liberdade do elemento requisitado, baseado na GEOMETRIA e na ordem de aproximação.

- Finalmente, entre as linhas 30–47 são exportadas (notemos o **`public:`**) todas as configurações do elemento finito, baseadas nas escolhas feitas pelo usuário, que correspondem às apresentadas na tabela 5.1.

5.3.3 O gerador de elementos

O processo de parsing (descrito na seção 5.3.2) toma as informações fornecidas pelo usuário em forma de lista ligada template, e produz um conjunto maior de informações "mastigadas" que representam o objeto imaginado.

O passo seguinte no processo consiste em tomar este conjunto — agora bem definido — e analisá-lo para decidirmos quais classes que, quando combinadas, produzirão o objeto desejado pelo usuário. Esta etapa final é de responsabilidade das *classes geradoras*.

A listagem 5.8 exibe a listagem da classe geradora de elementos finitos (`FE_GENERATOR`), que analisaremos a seguir.

O primeiro fato a ser notado na classe configuradora são seus parâmetros template (`P1` a `P5`), todos assumindo como padrão `ListNil`. Este recurso é necessário para que possamos especificar um número variável de características de configuração. Por exemplo, caso especifiquemos somente a FAMÍLIA e a GEOMETRIA, três opções de configuração (`P3` a `P5`) ficarão "vazias" (iguais a `ListNil`).

Estes cinco parâmetros são organizados em uma classe que simula uma lista ligada template (linha 8), que é passada ao parser (linha 11). É interessante notar que o tipo assim definido (`Config_t`) possui todas as configurações definidas pelo processo de parsing, como descritas na seção anterior.

Até agora fizemos vários "malabarismos" com templates, mas a linha 6 realmente *ultrapassa* o que já foi feito até o momento. Se observarmos claramente, veremos que a linha define um tipo `generator_t` *que é o próprio gerador de elementos finitos*. Colocando de outra forma, *o gerador é auto-contido*. Sob este ponto de vista, isto significa que se o usuário (ou outras classes) conhecem o gerador, elas conhecem tudo, *inclusive a própria configuração que deu origem ao gerador*. Daí dizermos que o objeto comporta-se como um *organismo com DNA*.

Esta característica única é aproveitada já nas linhas seguintes (14–18), onde o PAPEL do elemento determina qual classe base será usada. Caso `Config_t::role` seja `full_element` — o que corres-

[2] De fato, deveria haver um teste aqui, para sabermos se estamos com um elemento com base de Lagrange ou de Hermite. Mas, como na implementação atual só estão sendo levados em conta elementos finitos com base de Lagrange, este teste foi eliminado.

ponde a um elemento do interior do domínio, o resultado do meta `IF` (vide capítulo 4) será a classe `FeContainers< generator_t >`; caso contrário será `FeBoundary< generator_t >`, que corresponde a um elemento de fronteira. Qualquer que seja o resultado da decisão, este será armazenado no tipo `_CleanFe_t`. É interessante observarmos que o gerador (via o tipo `generator_t`) é passado para as classes `FeContainers` ou `FeBoundary`, para que elas possam extrair informações sobre o elemento requisitado. Estas duas classes são a primeira camada mixin que efetivamente implementa o conceito de elemento finito e seu funcionamento depende das configurações armazenadas em `generator_t`.

As classes `FeContainers` e `FeBoundary` implementam — baseadas em `generator_t` — as estruturas de dados necessárias para o armazenamento dos nós de um elemento finito e a ordem em que são listados, bem como diversos métodos para a recuperação e uso desta informação.

Finalmente, na linha 21, o tipo `_CleanFe_t` é empacotado na classe mixin final, chamada `Fe`. A classe `Fe` é apenas uma casca externa ao elemento, que implementa métodos básicos de leitura e escrita de um elemento finito.

O diagrama da figura 5.4 ilustra as relações entre as diversas classes até agora apresentadas para a configuração de um elemento finito. Notemos que as classes dentro do retângulo cinza são mutuamente exclusivas, ou uma ou outra é escolhida.

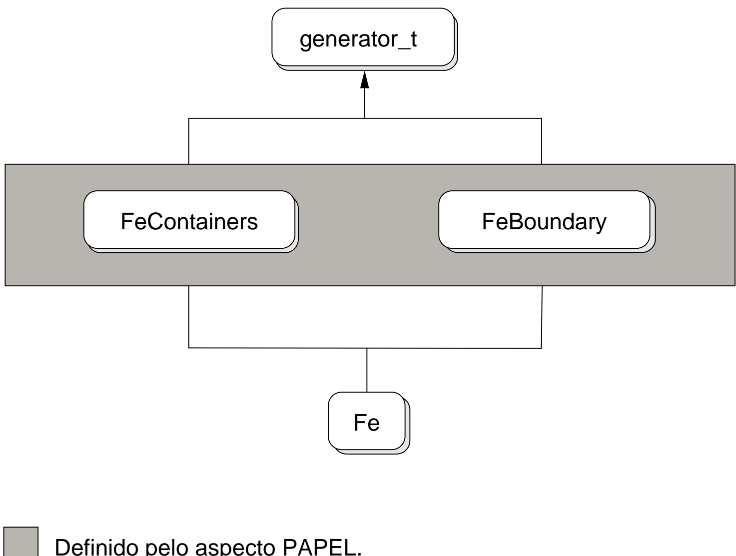


Figura 5.4: Relações entre as diversas classes que formam um elemento.

5.4 Outros configuradores

Como mencionamos no início deste capítulo, escolhemos uma abordagem que consideramos aprimorada em relação à apresentada, por exemplo, em [CE00]. Ao contrário do que é feito na referência citada, escolhemos por criar diversos configuradores menores, ao invés de um único configurador mono-

lítico[3]. Além da maior modularidade que este conceito impõe, uma nova possibilidade se apresenta: a de combinar configurações ou, ainda dentro da nossa analogia genética, combinar DNAs.

Nas próximas seções apresentaremos outros aspectos relacionados a uma simulação usando elementos finitos que são automaticamente configurados. As relações entre os configuradores serão ressaltadas quando necessário.

5.4.1 Configurador da base

O configurador para a base de elementos finitos cria o conjunto de funções que a compõe usando:

- Algumas informações provenientes do elemento finito a ela atrelado;
- Informações dadas pelo usuário;
- Informações extraídas dos operadores diferenciais do problema em questão.

A figura 5.5 mostra a DSL para o conceito "base de elementos finitos".

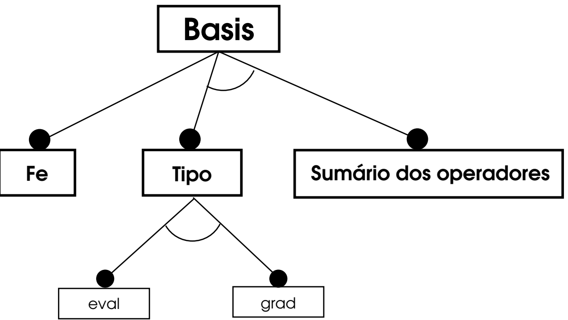


Figura 5.5: DSL para o conceito "base de elementos finitos".

Temos, portanto, três aspectos para modelarmos uma base: Fe, Tipo e Sumário de operadores:

Fe O primeiro (e indispensável) aspecto refere-se ao elemento finito "dono" daquela base. Este é o primeiro exemplo de um configurador que usa informações de outros configuradores.

Tipo Por tipo estamos nos referindo de fato a quantas derivadas esta base necessita. Alguns operadores necessitam apenas da função de base (opção `eval`) e outras necessitam dos gradientes das funções (opção `grad`). Este aspecto é mutuamente exclusivo com o aspecto sumário.

Sumário Caso o usuário deseje que o configurador decida qual o tipo de base que para ele satisfaz às condições de aproximação de um determinado conjunto de operadores diferenciais, o aspecto a ser usado é o sumário. A grosso modo, é uma classe que recolhe informações de diversos operadores diferenciais e decide de quantas derivadas a base necessita. Este aspecto é mutuamente exclusivo com o aspecto tipo. Novamente temos aqui um exemplo de combinação de configurações.

[3] De fato, no início das implementações, tentamos definir um único configurador, mas a complexidade da tarefa se mostrou tão grande que acabou nos levando a idéia dos "micro-configuradores", que se revelou bem mais viável.

A implementação da DSL para a base é muito similar à do elemento finito, podendo ser apreciada nas listagens anexas. Porém, vale a pena comentarmos as características que são aproveitadas do aspecto FE na configuração da base. Lembramos que este elemento finito foi inteiramente configurado por outro conjunto de classes; o que o configurador da base faz é aproveitar estas informações, sem que o usuário precise fornecê-las novamente.

Como vimos na seção 5.3.3, a característica mais marcante deste tipo de abordagem é o fato de os objetos carregarem informações sobre sua própria construção (na analogia, o DNA). O elemento configurado automaticamente não é exceção, o que significa que acessando-o, o configurador da base tem acesso a todas as informações de geometria, de grau de aproximação ou do que mais for necessário.

A listagem 5.9 nos mostra parte do gerador de bases para elementos finitos. Nas linhas 4 e 5 temos a declaração do gerador (dentro dele mesmo) e da lista de parâmetros templates de configuração (com três elementos).

Na linha 8 temos o repositório de configurações da base de elementos finitos, após o processo de parsing. Nas linhas 11 e 12 recuperamos o repositório de configurações para o elemento finito subjacente.

Finalmente, entre as linhas 17 e 26 são feitos os testes que efetivamente constróem a classe final. É checada a GEOMETRIA (obtida do repositório de configurações do elemento), e depois a ordem de aproximação[4]. Por exemplo, tendo escolhido `lagrange< 2 >` como FAMÍLIA e `triangle` como GEOMETRIA, o resultado da meta construção `SWITCH/CASE` será `LagrangeTriangle_P2< generator_t >`, devidamente armazenada no tipo `_Basis1_t`.

Num segundo momento, entre as linhas 28 e 41, testamos a necessidade de gradientes na base. Caso sejam necessários, novamente testamos a GEOMETRIA e a ordem de aproximação e empacotamos `_Basis1_t` em uma classe mixin que contém os gradientes. Ainda seguindo no exemplo, teríamos como resultado a classe `LagrangeTriangleGrad_P2< _Basis1_t >`, devidamente armazenada no tipo `_Basis2_t`.

Finalmente `_Basis2_t` é empacotada em uma classe chamada `Basis`, que cuida de fornecer uma interface comum a todas as bases de elementos finitos.

5.4.2 Configurador da transformação padrão ↔ real

Outro componente essencial na modelagem através de elementos finitos são as *transformações entre elemento padrão e real*[5], em conjunto com os respectivos jacobianos.

A figura 5.6 nos mostra a DSL para o conceito de transformação entre elementos padrão e real.

Do diagrama notamos que temos três aspectos para modelarmos uma transformação do tipo L2G: FE, MALHA e TIPO:

Fe Como na base, o aspecto indispensável refere-se ao elemento finito "dono" da transformação.

Malha Armazena o tipo de malha em que estamos fazendo as necessárias transformações padrão ↔ real. Esta informação é necessária pois a transformação depende fortemente do elemento onde está sendo aplicada.

[4] Deveríamos também checar qual família de aproximação foi escolhida, mas no exemplo estamos nos restringindo a Lagrange.

[5] E aqui abreviadas por L2G, do Inglês "*local to global*".

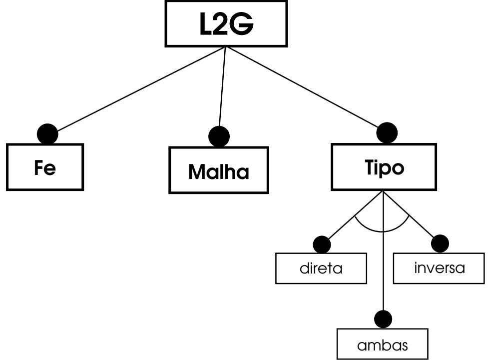


Figura 5.6: DSL para transformação elemento padrão ↔ elemento real.

Tipo Por TIPO estamos nos referindo ao fato da transformação ser direta (padrão → real), inversa (real → padrão) ou ambas.

Não entraremos em maiores detalhes sobre a implementação do configurador da transformação, pois é bastante similar aos outros já apresentados (elemento e base).

5.4.3 Configuradores dos operadores diferenciais

A configuração de operadores diferenciais em OSIRIS é bastante sofisticada, pois além do configurador em si, ela introduz um mecanismo de expressões templates, que faz com que o assunto mereça um capítulo a parte (capítulo 6).

```
1   template< class ParamList_t >
2   class FeParser {
3     private:
4       // Encontrando as configurações na lista estática.
5       typedef typename FindWithId< ParamList_t, fe_Family_id >::RET FE_Family_p;
6       typedef typename FindWithId< ParamList_t, fe_Geometry_id >::RET FE_Geometry_p;
7       typedef typename FindWithId< ParamList_t, fe_SpatialDimension_id >::RET FE_SpDim_p;
8       typedef typename FindWithId< ParamList_t, fe_Role_id >::RET FE_Role_p;
9
10      typedef typename GetTypeOrDefault< FE_Family_p, lagrange<> >::RET _FE_Family_t;
11      typedef typename GetTypeOrDefault< FE_Geometry_p, triangle >::RET _FE_Geometry_t;
12
13      enum {
14        order_             = _FE_Family_t::order,
15        dimensions_        = _FE_Geometry_t::dimensions,
16        spatialDimensions_ = GetIntOrDefault< FE_SpDim_p, dimensions_ >::RET,
17        role_              = GetIntOrDefault< FE_Role_p, full_element >::RET
18      };
19
20      // Retorna o número de graus de liberdade a partir da geometria e da ordem.
21      typedef LagrangeanDOF< _FE_Geometry_t, order_ > DOF;
22
23      typedef uinteger_t _RefsContainerAtom_t;
24
25      // Container de nós (incidência do elemento).
26      typedef Vector< DOF::D, _RefsContainerAtom_t > _RefsContainer_t;
27
28    public:
29      // Exportando toda a configuração do elemento.
30      struct Config {
31          typedef _FE_Family_t FE_Family_t;
32          typedef _FE_Geometry_t FE_Geometry_t;
33          typedef _RefsContainerAtom_t RefsContainerAtom_t;
34          typedef _RefsContainer_t RefsContainer_t;
35
36          enum {
37            order = order_,
38            dimensions = dimensions_,
39            spatialDimensions = spatialDimensions_,
40            role = role_,
41            ndof = DOF::D,
42            ndofb = DOF::B
43          };
44      };
45
46      typedef Config RET;
47      typedef Config Config_t;
48  };
```

Listagem 5.7: Parser para configuração de um elemento finito.

```
1  template< class P1 = ListNil, class P2 = ListNil, class P3 = ListNil,
2           class P4 = ListNil, class P5 = ListNil >
3  class FE_GENERATOR
4  {
5     private:
6        typedef FE_GENERATOR< P1, P2, P3, P4, P5 > generator_t;
7
8        typedef typename MakeList5< P1, P2, P3, P4, P5 >::RET ParamList_t;
9
10    public:
11       typedef typename FeParser< ParamList_t >::RET Config_t;
12
13    private:
14       typedef typename
15       IF< EQUAL< Config_t::role, full_element >::RET,
16           FeContainers< generator_t >,
17           FeBoundary < generator_t >
18       >::RET _CleanFe_t;
19
20    public:
21       typedef Fe< _CleanFe_t > RET;
22
23       typedef generator_t Generator_t;
24 };
```

Listagem 5.8: Gerador para um elemento finito.

```
template< class P1 = ListNil, class P2 = ListNil, class P3 = ListNil >
class BASIS_GENERATOR {
  private:
    typedef BASIS_GENERATOR< P1, P2, P3 > generator_t;
    typedef typename MakeList3< P1, P2, P3 >::RET ParamList_t;

  public:
    typedef typename BasisParser< ParamList_t >::RET Config_t;

  private:
    typedef typename Config_t::BASIS_FeGenerator_t FeGenerator_t;
    typedef typename FeGenerator_t::Config_t FeConfig_t;

    enum { basisType = Config_t::basisType };
    enum { order = FeConfig_t::order };

    typedef typename
    IF< EQUAL< FeConfig_t::FE_Geometry_t::id, triangle_id >::RET,
        typename SWITCH< order,
                CASE< 0, LagrangeTriangle_P0< generator_t >,
                CASE< 1, LagrangeTriangle_P1< generator_t >,
                CASE< 2, LagrangeTriangle_P2< generator_t >,
                DEFAULT< LagrangeTriangle_P1< generator_t > >
        > > > >::RET,
        LagrangeTetra_P1< generator_t >
    >::RET _Basis1_t;

    typedef typename
    IF< EQUAL< basisType, grad_basis >::RET,
        typename
        IF< EQUAL< FeConfig_t::FE_Geometry_t::id, triangle_id >::RET,
            typename SWITCH< order,
                    CASE< 0, LagrangeTriangleGrad_P0< _Basis1_t >,
                    CASE< 1, LagrangeTriangleGrad_P1< _Basis1_t >,
                    CASE< 2, LagrangeTriangleGrad_P2< _Basis1_t >,
                    DEFAULT< LagrangeTriangleGrad_P1< _Basis1_t > >
            > > > >::RET,
            LagrangeTetraGrad_P1< _Basis1_t >
        >::RET,
        _Basis1_t
    >::RET _Basis2_t;

  public:
    typedef generator_t Generator_t;

    typedef Basis< _Basis2_t > RET;
};
```

Listagem 5.9: Gerador para uma base de elementos finitos.

Parte III

Amálgama

CAPÍTULO 6

Operadores e expressões templates

Dando continuidade ao exposto no capítulo 5, apresentaremos o processo de configuração de operadores diferenciais como implementados em Osiris. *Reservamos um capítulo a parte pois, além da configuração — bastante similar às já apresentadas para elementos, bases e transformações — uma nova característica emerge na criação dos operadores. Como eles são essencialmente estruturas de cálculo, faz-se necessária a introdução de um mecanismo conveniente de expressões templates. Este capítulo será devotado à construção destas estruturas de cálculo, enquanto que o próximo será dedicado aos vários tipos de operadores e suas possibilidades de construção.*

6.1 Introdução

No capítulo 5 apresentamos os conceitos que embasam a modelagem e criação de objetos através de especificações de alto nível. Utilizamos este método para a criação de elementos finitos, suas bases e respectivas transformações (L2G).

A sistemática para a criação dos operadores diferenciais[1] é similar; *o diferencial original está nas classes mixin que serão compostas para a criação do operador final.* Os operadores diferenciais são essencialmente estruturas de cálculo e, portanto, merecem um tratamento especial. Todos os operadores criados aqui são baseados em um elemento finito, significando que esses operadores são definidos localmente.

Este tratamento se dará na forma de *expressões templates* (seção 3.4). O ponto interessante de nossa abordagem é que em *nenhum momento será criada uma estrutura própria* que implemente um mecanismo de expressões templates. Este mecanismo será aproveitado de uma biblioteca chamada Pooma — Parallel Object-Oriented Methods and Applications [CCH+98a, CCH+98b].

6.2 Pooma

6.2.1 A biblioteca

Pooma [CCH+98b, CCH+98a] é uma biblioteca escrita em C++ usando templates com vistas à Computação Científica de alto desempenho. Suas versões iniciais foram desenvolvidas por uma equipe

[1] Referimo-nos aqui às versões já discretizadas de cada operador.

de cientistas do Los Alamos National Laboratory, LANL[2]. Atualmente a biblioteca POOMA é mantida por membros dessa equipe em uma empresa chamada Code Sourcery[3] e também por membros da comunidade[4].

Essencialmente POOMA traz ao usuário estruturas matriciais cheias com até 7 dimensões, com todas as operações algébricas e funções matemáticas implementadas através de expressões templates.

Outra característica interessante da biblioteca é a de isolar o usuário dos aspectos internos de implementação e dos esquemas de armazenamento. Isto significa que um programa escrito usando-se POOMA em um computador serial é facilmente adaptado para sistemas com memória distribuída ou computadores paralelos (clusters).

POOMA também provê outras estruturas, como malhas, associações de malha e valores definidos sobre a malha (*fields*), sistemas de coordenadas e alguns operadores diferenciais. O que deve ser mantido em mente, entretanto, é que a biblioteca *foi desenvolvida inteiramente baseada no método de diferenças finitas*. Em outras palavras, todas as operações são baseadas em estênceis de diferenças finitas e malhas estruturadas.

De todas as funcionalidades da POOMA, a que mais nos interessa é uma das menos óbvias, e com certeza uma das mais usadas, embora de forma indireta e sem o conhecimento do usuário: seu poderoso mecanismo de expressões templates. A forma como este mecanismo foi utilizado em OSIRIS criou uma ligação interessante entre as duas bibliotecas. Explicando melhor podemos dizer, grosso modo, que uma biblioteca pode ser usada de duas formas:

1. Ao longo do código em construção, quando ela pode fornecer todas as funcionalidades que o usuário necessita; e,

2. de modo estendido, quando o usuário inclui na biblioteca funcionalidades não presentes, porém necessárias. Uma nova versão ou, em alguns casos, uma nova biblioteca emerge deste processo.

A inclusão da POOMA em OSIRIS, então, se deu de uma forma híbrida entre as duas: ela não apresentava todas as funcionalidades necessárias, mas foi usada amplamente ao longo do código, sendo portanto estendida, *mas sem alterações em seu código fonte*. Esta afirmação ficará clara ao longo do texto.

6.2.2 Expressões templates em Pooma

A mesma equipe responsável pela criação do POOMA criou uma biblioteca auxiliar chamada PETE[5] (Portable Expression Template Engine) [CCH+98a]. A idéia era a de isolar o mecanismo de expressões templates em uma biblioteca separada, que pudesse ser reaproveitada em outras implementações. De fato, nossa idéia inicial era justamente esta: usar a PETE para criar as operações de soma, subtração

[2] `http://www.lanl.gov`

[3] `http://www.codesourcery.com/pooma/pooma`. Este nome faz uma brincadeira com fonte = source e bruxaria = sourcery.

[4] Embora o POOMA possa ser usado em projetos acadêmicos e de pesquisa, até o presente momento a questão de seu licenciamento ainda não foi concluída.

[5] `http://www.codesourcery.com/pooma/pete`

e multiplicação por constante — entre outras — diretamente *entre* operadores diferenciais. Se assim implementado, o usuário poderia escrever de forma quase natural as aproximações de seu modelo.

Esta idéia foi abandonada devido às dificuldades, como a definição formal do escopo de cada operador e das subseqüentes combinações possíveis de operadores, e à complexidade de mecanismos deste tipo, mesmo quando implementados através de uma biblioteca auxiliar como a PETE.

Porém o estudo detalhado tanto do PETE quanto do POOMA nos fez chegar a uma solução híbrida. PETE define uma série de classes para a construção de expressões templates. POOMA especializa estas classes para suas estruturas de dados, definindo diversos operadores unários, operadores que atuam sobre um único argumento, como a negação, o seno e o cosseno, e binários, dois operadores, como a soma, o produto e o quociente. Sob certo aspecto, *as especializações em* POOMA *ensinam as classes de* PETE *sobre como elas devem executar as operações matemáticas.* Não entraremos em detalhes sobre o funcionamento completo destes mecanismos, mas exemplificaremos o seu uso para posterior conexão com os operadores em OSIRIS. A listagem 6.1 mostra uma operação bastante simples usando um array do POOMA.

```
int main() {
   // Declarando dois arrays de precisão dupla.
   Array< 1, double, Brick > x( 100 ), y( 100 );

   x = 0.01 * iota( 100 ).comp( 0 );

   y = sin( x ) + x;
}
```

Listagem 6.1: Exemplo de utilização de arrays do POOMA.

Neste exemplo, são criados dois arrays de precisão dupla com cem componentes cada, através da declaração `Array< 1, double, Brick >`. O "1" da declaração é o número de dimensões do array (neste caso uma, correspondendo a um vetor), **double** é o tipo armazenado no array e `Brick` é a forma de armazenamento (um único bloco de memória, contíguo), denominada no jargão POOMA de *engine*[6].

Engines têm um papel central no funcionamento do POOMA. São eles que de fato *definem o funcionamento de um array*. O engine `Brick` diz que o array será um único bloco contíguo de memória. Outros exemplos de engine são o `Remote`, que transforma um engine local em um engine remoto, hospedado na memória de outra máquina ou ainda o engine `MultiPatch`, que quebra um array em vários e os distribui entre vários processadores. A arquitetura de engines é bastante poderosa, pois realiza pelo usuário todas as complexidades de comunicação entre processadores ou gerenciamento de memória sem a necessidade de interferência deste.

Um engine muito especial e de importância vital ao funcionamento do POOMA é o chamado *engine de expressões*[7]. Pelo nome já podemos imaginar que ele tem alguma relação com o mecanismo de expressões templates, o que é inteiramente correto, como veremos na continuação do exemplo.

Voltando à listagem 6.1, o array `x` é preenchido com a seqüência $0, 0.01, 0.02, \ldots 0.99$ via uma função especial chamada `iota`. Finalmente, um *operador unário* é aplicado para o cálculo do seno das

[6] Termo técnico. Máquina, maquinário. Será mantido o original.
[7] *Expression engine*, do original.

componentes do vetor `x` e um *operador binário* é aplicado para a soma. O resultado é armazenado em `y` via um *operador de atribuição*.

A figura 6.1 ilustra a árvore binária equivalente para a expressão $y = \sin(x) + x$. Nos nós cinzas estão as operações e em nos nós brancos os operandos.

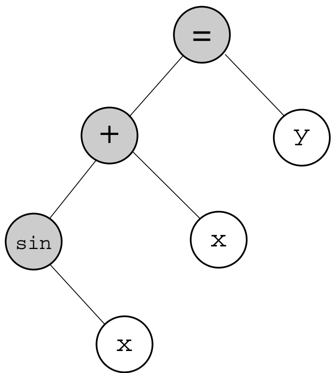


Figura 6.1: Árvore binária representando a expressão $y = \sin(x) + x$.

Como o mecanismo de expressões templates, associado aos engines, representa esta expressão dentro do POOMA? A resposta a esta pergunta é essencial, pois é a partir dela que contruiremos novas operações. O operador de atribuição é comum. O segredo está na expressão "$sin(x)+x$". Explicaremos a montagem da expressão de dentro para fora, em partes. Depois exibiremos o resultado completo e, finalmente, um diagrama, de forma a propiciar um completo entendimento deste aspecto tão essencial.

6.2.3 Referenciando os operandos

O primeiro passo na construção da expressão é o *referenciamento dos operandos (ou argumentos)*. Este passo é essencial pois evita cópias temporárias dos operandos que, como vimos no capítulo 2, é um dos grandes responsáveis pela lentidão da linguagem.

Este referenciamento é feito através da classe `Reference`, que define uma referência para o tipo a ela passado. Para criar uma referência ao vetor `x`, que é do tipo `Array< 1, double, Brick >`, basta declararmos `Reference< Array< 1, double, Brick > >`. Com isto, dentro desta classe é criada uma referência para o array, evitando assim a cópia desnecessária.

6.2.4 Operadores unários

Tendo criado a referência ao vetor, o próximo passo é expandir a operação de seno. A operação seno é definida como uma *operação unária*, o que significa que recebe um único argumento de entrada. O mecanismo de expressões templates do POOMA define expressões unárias através da classe `UnaryNode`, que recebe dois parâmetros template. O primeiro é uma classe associada à operação e a segunda ao operando. Neste caso, a classe associada à operação é `FnSin` (cuja definição será importante mais à frente) e associado ao operando é a referência ao array. Desta forma temos:

```
UnaryNode< // Nó unário.
  FnSin, // Operação seno.
```

```
    Reference< Array< 1, double, Brick > > // Operando (referência).
  >
```

6.2.5 Operadores binários

Finalmente podemos expandir o operador de soma, que é um operador binário. Operadores binários são definidos através da classe `BinaryNode`, que recebe três parâmetros template, um associado à operação e dois aos argumentos. A classe associada à operação de adição é `OpAdd`. O primeiro argumento é o operador unário de seno e o segundo é a referência ao vetor `x`. Assim, temos o seguinte trecho de código:

```
BinaryNode< // Nó binário.
  OpAdd, // Operação (adição).
  UnaryNode< FnSin, Reference< Array< 1, double, Brick > > >, // Primeiro argumento (seno).
  Reference< Array< 1, double, Brick > > > // Segundo argumento.
```

6.2.6 O engine de expressões

O passo final na construção da expressão é a criação de um array com um engine de expressões, com todos os operadores já definidos, como acima. O engine de expressões é definido na classe `ExpressionTag` e a declaração final fica como a da listagem 6.2.

```
Array< 1, double,
  ExpressionTag< // O engine de expressões.
    BinaryNode< OpAdd, // As operações unárias e binárias.
      UnaryNode< FnSin, Reference< Array< 1, double, Brick > > >,
      Reference< Array< 1, double, Brick > >
    > // BinaryNode
  > // ExpressionTag
> // Array
```

Listagem 6.2: Expressão completa para $\sin(x) + x$.

Todos os operadores serão avaliados em um mesmo loop na ocasião da atribuição ao vetor `y`, eliminando o problema da multiplicação de loops, como apresentado no capítulo 2. Este efeito é obtido quando do acesso ao operador **`operator()`** do array da listagem 6.2, que é expandido nos respectivos operadores **`operator()`** para cada nó da árvore de expressões.

6.2.7 Árvore de expressões

A figura 6.2 finaliza a explanação sobre os operadores do Pooma. Nela, podemos ver uma reprodução da árvore binária em 6.1 (a menos do operador de atribuição), construída inteiramente de forma automática e durante a compilação.

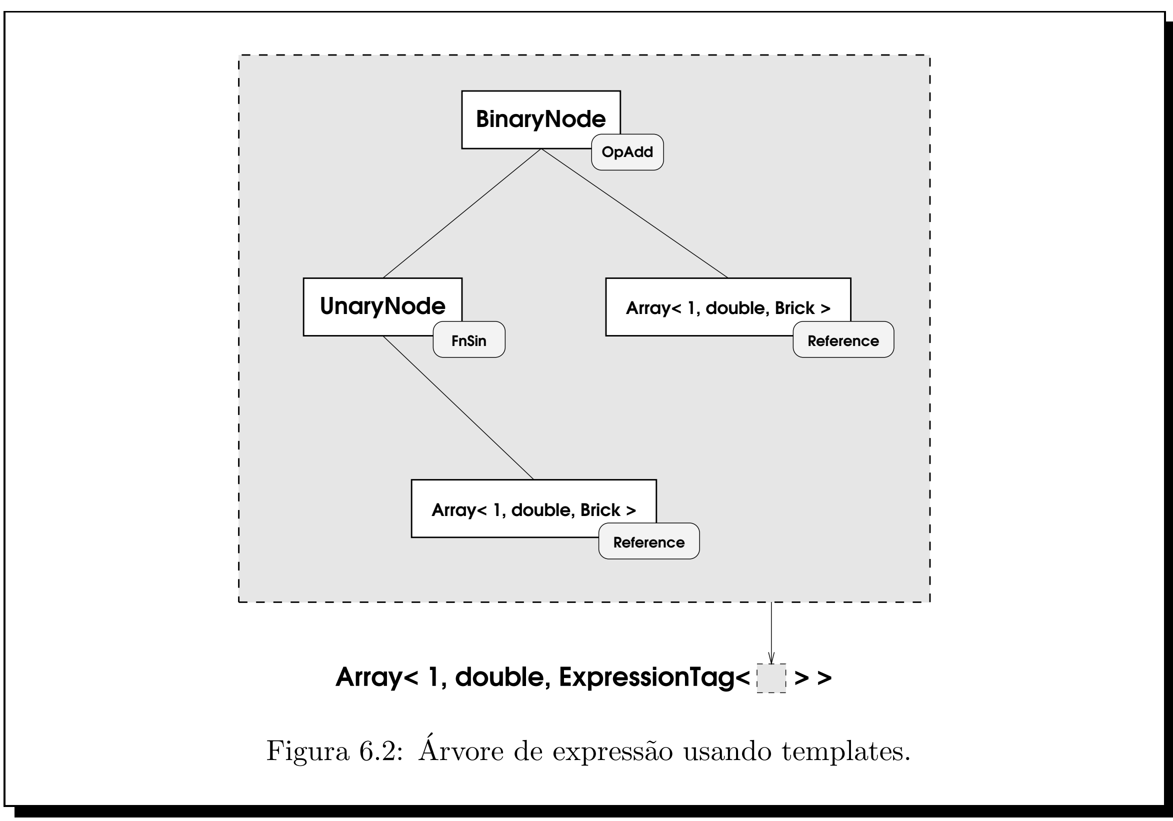


Figura 6.2: Árvore de expressão usando templates.

6.2.8 Tags de operadores

As classes que se associam a operações (`FnSin` e `OpAdd`) são denominadas *tags de operadores*. Estas classes são muito simples (normalmente são apenas estruturas), e definem como uma operação deve ser aplicada em um único elemento do array. Posteriormente, o engine de expressões trata de aplicar esta operação a cada elemento componente do array.

A listagem 6.3 — retirada do arquivo `OperatorTags.h` do código fonte do POOMA — mostra a definição da classe (de fato, uma estrutura) `FnSin`. A classe é extremamente simples, contendo basicamente um construtor (não exibido) e um operador parêntesis.

```
struct FnSin {
  // Alguns construtores aqui...

  template< class T > inline typename UnaryReturn< T, FnSin >::Type_t
  operator()( const T& a ) const {
    return sin(a);
  }
};
```

Listagem 6.3: Tag de operador `FnSin`.

Como podemos observar, o operador parêntesis tem um parâmetro template `T`, e recebe exatamente um argumento do tipo `T`, na variável `a`. A única operação executada pelo operador parêntesis é o

seno desta variável `a`. Todo o segredo do funcionamento desta classe está em seu valor de retorno, `UnaryReturn< T, FnSin >::Type_t`. A classe `UnaryReturn` é uma classe traits (ver seção 3.6). A construção `UnaryReturn< T, FnSin >::Type_t` *decide o tipo de retorno da função de acordo com o argumento de entrada*, em um tipo de polimorfismo às avessas. Uma definição típica para `UnaryReturn` seria:

```
struct UnaryReturn< double, FnSin > {
  typedef double Type_t;
};
```

que diz que, para a classe `FnSin`, com entrada do tipo **`double`**, a saída (`Type_t`) deve ser também do tipo **`double`**. O poder desta abordagem reside na possibilidade de *especializar a classe `UnaryReturn` quantas vezes forem necessárias*, para diferentes tags de operadores e diferentes tipos de entrada. Podemos "ensinar" o mecanismo de expressões templates de Pooma de forma a fazê-lo entender as operações que são de nosso interesse.

Para ilustrar o poder desta abordagem, criaremos uma tag de operador que, dado um valor real x, retorna, por exemplo, o vetor $[1-x(3-2x), 4x(1-x), -x(1-2x)]^T$. Não por coincidência, esta é a base de Lagrange de segunda ordem em um elemento unidimensional de referência $[0,1]$. O código da tag, juntamente com a especialização conveniente de `UnaryReturn` compõe a listagem 6.4.

```
struct Lag2 {
  inline Lag2() {} // Construtor vazio.

  template< class T > inline typename UnaryReturn< T, Lag2 >::Type_t
  operator()( const T& x ) const {
    typedef UnaryReturn< T, Lag2 >::Type_t Type_t;

    return Type_t( 1.0-x*(3.0-2.0*x), 4.0*x*(1.0-x), -x*(1.0-2.0*x) );
  }
};

struct UnaryReturn< double, Lag2 > {
  typedef Vector< 3, double, Full > Type_t;
};
```

Listagem 6.4: Tag de operador `Lag2`.

A especialização de `UnaryReturn` diz ao mecanismo de expressões templates que a tag de operador `Lag2`, em conjunto com um **`double`** deve retornar um tipo `Vector< 3, double, Full >`. O tipo `Vector` é um vetor estático, o que significa que seu tamanho é fixo ao longo de todo o código, no caso, com 3 componentes.

O operador parêntesis de `Lag2` simplesmente cria o vetor já atribuindo às suas três componentes os valores desejados. Assim, se quisermos efetuar a operação podemos fazer

```
double a = 0.7;

Vector< 3, double, Full > b = Lag2()( a );
```

que resulta em um vetor `b` com componentes $(-0.12, 0.84, 0.28)^T$.

Mas ainda falta um passo para completarmos a definição de nosso operador. Até o momento ensinamos às expressões templates como atuar em uma única variável numérica. Vamos supor que quiséssemos atuar em um array de números reais, exatamente como fizemos com a função seno. Ainda precisamos "ensinar" às expressões templates como tratar desse caso.

```
template< int D, class E > inline typename
MakeReturn<
  UnaryNode< Lag2, typename CreateLeaf< Array< D, double, E > >::Leaf_t >
>::Expression_t
lag2( const Array< D, double, E >& a ) {
  typedef UnaryNode< Lag2, typename CreateLeaf< Array< D, double, E > >::Leaf_t > Tree_t;

  return MakeReturn< Tree_t >::make(
    Tree_t( CreateLeaf< Array< D, double, E > >::make( a ) ) );
}
```

Listagem 6.5: Construção da função `lag2()`.

A listagem 6.5 mostra uma função chamada `lag2`, que será associada à tag de operador `Lag2`. Através desta função, o mecanismo de expressões templates saberá como aplicar o operador parêntesis por nós definido em estruturas como os arrays da POOMA.

Não examinaremos todos os detalhes por trás das classes `MakeReturn` ou `CreateLeaf`. Basta sabermos que a classe `CreateLeaf` empacotará o tipo de entrada em uma referência (como já havíamos dito anteriormente), que será inserido em uma classe do tipo `UnaryNode`, com sua respectiva tag. Devemos observar que o tipo de entrada tem sua dimensão e seu engine como parâmetros template. Esta flexibilidade permite a aplicação da função em arrays de qualquer dimensão (dentro das suportadas pelo POOMA) e com qualquer engine. A implicação disto é que podemos aplicar a função `lag2()` em arrays com engine `Brick` (arrays contíguos na memória), `MultiPatch` (arrays em memória distribuída) ou, o mais importante, *em engines de expressão*, o que nos permitiria escrever, por exemplo `lag2( sin(x) + x )`.

Estamos agora em posição de assentar as bases para os operadores diferenciais necessários dentro da sistemática apresentada até o momento.

6.3 Operadores diferenciais e Pooma: bases

Com os subsídios fornecidos na seção 6.2, estamos em condições de construir os operadores diferenciais discretizados que atuarão em conjunto com as outras classes de OSIRIS no intuito de produzir uma simulação completa de elementos finitos.

6.3.1 Geometria, integração numérica e Pooma

O método de escolha para integração numérica é a *integração gaussiana* [Cun00], cuja definição clássica pode ser aqui recordada como segue.

Sejam $f : [-1, 1] \to \mathbb{R}$, $x_i \in (-1, 1)$, $i = 1, \ldots, m$ pontos e w_i, $i = 1, \ldots, m$ pesos, todos convenien-

temente escolhidos. A *quadratura ou aproximação gaussiana* para $\int_{-1}^{1} f(x)\,dx$ é dada por (6.1).

$$\int_{-1}^{1} f(x)\,dx \approx \sum_{i=1}^{m} f(x_i)\,w_i. \tag{6.1}$$

As fórmulas (6.2) e (6.3) são, respectivamente, aproximações Gaussianas para integrais duplas de funções $g : [-1,1]^2 \to \mathbb{R}$ e triplas de funções $h : [-1,1]^3 \to \mathbb{R}$. No caso bidimensional os pontos e pesos de integração são (x_i, y_j) e w_i, w_j, para $i = 1,\ldots,m$ e $j = 1,\ldots,n$. No caso tridimensional, (x_i, y_j, z_k) e w_i, w_j, w_k para $i = 1,\ldots,m$, $j = 1,\ldots,n$ e $k = 1,\ldots,p$.

$$\int_{-1}^{1}\int_{-1}^{1} f(x,y)\,dy\,dx \approx \sum_{i=1}^{m}\sum_{j=1}^{n} f(x_i,y_j)\,w_iw_j. \tag{6.2}$$

$$\int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1} f(x,y,z)\,dz\,dy\,dx \approx \sum_{i=1}^{m}\sum_{j=1}^{n}\sum_{k=1}^{p} f(x_i,y_j,z_k)\,w_iw_jw_k. \tag{6.3}$$

Naturalmente, não estamos limitados a integração em $[-1,1]$. Transformações convenientes nos permitem a aproximação da integral em qualquer domínio razoável sob hipóteses adequadas.

Os pontos de integração unidimensionais e seus respectivos pesos são obtidos a partir dos zeros dos polinômios de Legendre, usando um método de Newton com deflação polinomial (detalhes podem ser obtidos em [KS99]).

O aspecto crucial na escolha dos pontos e pesos de integração não é exatamente o matemático, mas sim o de estrutura de dados, pois queremos que as operações sobre estes pontos e pesos sejam integradas ao mecanismo de expressões templates da Pooma de maneira simples e eficiente.

Para facilitar a compreensão das estruturas de dados relativas aos pontos de integração, e também para utilização futura no capítulo de operadores, introduziremos algumas notações gráficas para os componentes mais comuns, de acordo com a tabela 6.1.

Como visto na tabela de representações gráficas, as diversas estruturas são combináveis, graças à maquinaria template por trás delas.

Pontos e pesos unidimensionais

Este caso não oferece grandes problemas. Os pontos serão armazenados em um array unidimensional de reais e os pesos em outro, ambos gerenciados pela classe `IntegrationPW< 1 >`. O acesso aos pontos é feito através do método `p()` e aos pesos via o método `w()`.

Como mostrado na tabela 6.1, tanto pontos quanto pesos terão a representação gráfica $[\bullet\cdots]$. Note-se que há uma relação entre a geometria dos elementos unidimensionais, usualmente isomorfa a uma reta, e a forma de um array unidimensional; daí vem nossa escolha.

Pontos e pesos bidimensionais

A primeira forma óbvia que nos vem à mente é a de dois vetores para os pontos — um para os pontos em x e outra para os pontos em y — e dois vetores para os pesos, da mesma forma um para cada coordenada. Esta é uma escolha que faz muito sentido em um contexto diferente da Pooma.

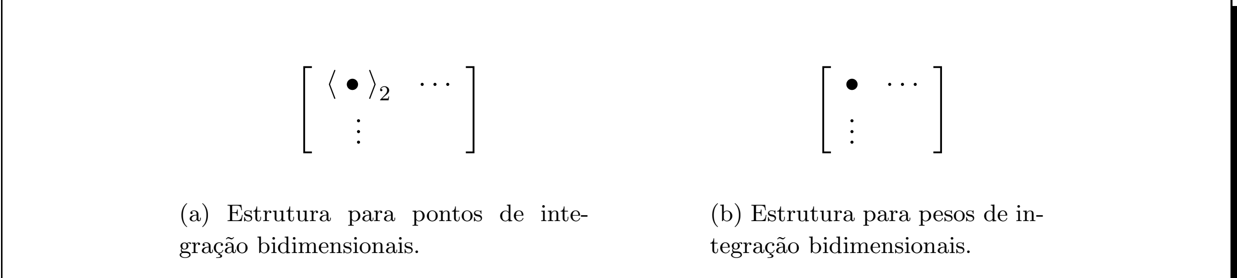
Estrutura	Descrição	Representção
`double`	Número real	$\bullet$
`Vector< N >`	Vetor de `N` componentes	$\langle \bullet \rangle_N$
`Tensor< N >`	Tensor de `N` componentes	$[\![\bullet]\!]_N$
`Array< 1 >`	Array unidimensional de reais	$[\bullet \cdots]$
`Array< 2 >`	Array bidimensional de reais	$\begin{bmatrix} \bullet & \cdots \\ \vdots & \end{bmatrix}$
`Array< 3 >`	Array tridimensional de reais	$\begin{matrix} & \ulcorner & \urcorner \\ \begin{bmatrix} \bullet & \cdots \\ \vdots & \end{bmatrix} & & \lrcorner \end{matrix}$
`Array< 2, Vector< 2 > >`	Array bidimensional de 2-vetores	$\begin{bmatrix} \langle \bullet \rangle_2 & \cdots \\ \vdots & \end{bmatrix}$
`Array< 3, Vector< 3 > >`	Array tridimensional de 3-vetores	$\begin{matrix} & \ulcorner & \urcorner \\ \begin{bmatrix} \langle \bullet \rangle_3 & \cdots \\ \vdots & \end{bmatrix} & & \lrcorner \end{matrix}$
`Array< 2, Tensor< 2 > >`	Array bidimensional de 2-tensores	$\begin{bmatrix} [\![\bullet]\!]_2 & \cdots \\ \vdots & \end{bmatrix}$
`Array< 3, Tensor< 3 > >`	Array tridimensional de 3-tensores	$\begin{matrix} & \ulcorner & \urcorner \\ \begin{bmatrix} [\![\bullet]\!]_3 & \cdots \\ \vdots & \end{bmatrix} & & \lrcorner \end{matrix}$
`Vector< 3, Array< 2 > >`	3-Vetor com componentes como arrays bidimensionais	$\left\langle \begin{bmatrix} \bullet & \cdots \\ \vdots & \end{bmatrix} \right\rangle_3$
`Vector< 4, Array< 3, Vector< 3 > > >`	4-Vetor com componentes como arrays tridimensionais de 3-vetores	$\left\langle \begin{matrix} & \ulcorner & \urcorner \\ \begin{bmatrix} \langle \bullet \rangle_3 & \cdots \\ \vdots & \end{bmatrix} & & \lrcorner \end{matrix} \right\rangle_4$

Tabela 6.1: Representação gráfica de estruturas de dados.

Como queremos aproveitar seus mecanismos de avaliação de expressões, armazenaremos os pontos de integração como um array bidimensional, onde cada componente é um vetor estático com duas posições (`Array< 2, Vector< 2 > >`). Esta escolha permitirá uma grande flexibilidade, que ficará clara mais à frente. Os pesos serão armazenados em um array bidimensional de reais, já contendo o produto $w_i w_j$.

Um array bidimensional composto de vetores com duas componentes, como é o caso dos pontos de integração bidimensionais, será representado pela figura 6.3(a). A representação gráfica dos pesos bidimensionais será de acordo com a figura 6.3(b).

Novamente chamamos a atenção para a relação entre elementos finitos bidimensionais planos e a forma bidimensional dos arrays envolvidos.



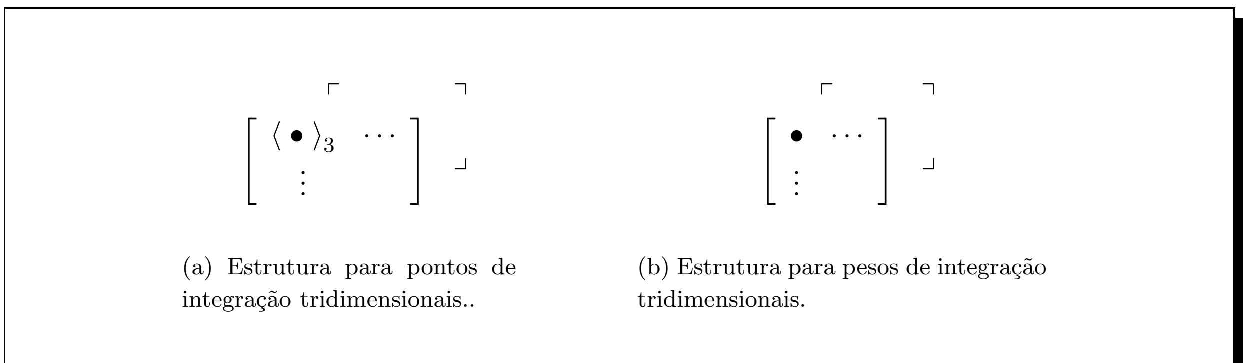
(a) Estrutura para pontos de integração bidimensionais.

(b) Estrutura para pesos de integração bidimensionais.

Figura 6.3: Representações gráficas para pontos e pesos de integração bidimensionais.

Pontos e pesos tridimensionais

Analogamente ao caso bidimensional, os pontos de integração tridimensionais serão vetores estáticos com três componentes, inseridos em um array tridimensional (similar a um cubo), resultando em objetos do tipo `Array< 3, Vector< 3 > >` (graficamente como na figura 6.4(a)). Os pesos serão tratados como um array tridimensional de números reais (figura 6.4(b)).



(a) Estrutura para pontos de integração tridimensionais..

(b) Estrutura para pesos de integração tridimensionais.

Figura 6.4: Representações gráfica para pontos e pesos de integração tridimensionais.

6.3.2 Avaliação da base

Usaremos os conceitos de expressões templates em POOMA como apresentados na seção 6.2 em conjunto com os pontos e pesos de integração definidos em 6.3.1.

Tomaremos como exemplo a base de Lagrange de primeira ordem, definida sobre um triângulo padrão (como no apêndice A). Esta base é composta de três funções, cada uma associada a um vértice do triângulo padrão, como apresentado em (6.4).

$$\begin{aligned} \varphi_1(\xi,\eta) &= 1-\xi-\eta \\ \varphi_2(\xi,\eta) &= \xi \\ \varphi_3(\xi,\eta) &= \eta \end{aligned} \tag{6.4}$$

Como já mostramos na seção 6.3.1, os pontos de integração bidimensionais (sobre os quais a base será

avaliada), serão armazenados como arrays bidimensionais de vetores com duas componentes (figura 6.3). Para fixar idéias, imaginemos que este array de pontos tem 3 linhas e 3 colunas, totalizando 9 pontos de integração (3 em cada eixo coordenado). Desta forma, a avaliação de cada função φ_i, $i = 1, 2, 3$ da base resulta em um outro array 3×3, mas desta vez de números reais. Esquematicamente:

$$\varphi_i\left(\begin{bmatrix}\langle\bullet\rangle_2 & \cdots\\ \vdots & \end{bmatrix}\right) = \begin{bmatrix}\bullet & \cdots\\ \vdots & \end{bmatrix}, \tag{6.5}$$

para $i = 1, 2, 3$.

Ainda aproveitando a estrutura de vetores da POOMA, o resultado da avaliação de cada uma das funções da base será colocado em uma posição de um vetor de três componentes. O resultado final será um vetor de três componentes, onde cada uma corresponde a um array bidimensional. Usando a notação gráfica e tomando $\boldsymbol{\varphi} = \langle\varphi_1, \varphi_2, \varphi_3\rangle$ temos:

$$\boldsymbol{\varphi}\left(\begin{bmatrix}\langle\bullet\rangle_2 & \cdots\\ \vdots & \end{bmatrix}\right) = \left\langle\begin{bmatrix}\bullet & \cdots\\ \vdots & \end{bmatrix}\right\rangle_3 \tag{6.6}$$

Para melhor ilustramos a discussão, exibimos a representação gráfica de um caso unidimensional, com base de Lagrange de primeira ordem ($\varphi_1(\xi) = \xi$ e $\varphi_2(\xi) = 1 - \xi$ sobre um elemento de linha (ou seja, sobre um segmento de reta)

$$\boldsymbol{\varphi}([\bullet\cdots]) = \langle\ [\bullet\cdots]\ \rangle_2 \tag{6.7}$$

e de um caso tridimensional, com base de Lagrange de primeira ordem ($\varphi_1(\xi, \eta, \zeta) = 1 - \xi - \eta - \zeta$, $\varphi_2(\xi, \eta, \zeta) = \xi$, $\varphi_3(\xi, \eta, \zeta) = \eta$ e $\varphi_4(\xi, \eta, \zeta) = \zeta$) sobre um tetraedro

$$\boldsymbol{\varphi}\left(\ulcorner\begin{bmatrix}\langle\bullet\rangle_3 & \cdots\\ \vdots & \end{bmatrix}\lrcorner\right) = \left\langle\ulcorner\begin{bmatrix}\bullet & \cdots\\ \vdots & \end{bmatrix}\lrcorner\right\rangle_4 \tag{6.8}$$

para $i = 1, 2, 3, 4$.

A listagem 6.6 mostra a classe `LagrangeTriangle_P1_op`, que implementa o cálculo efetivo das três funções que compõem a base (6.4).

A classe `LagrangeTriangle_P1_op` faz exatamente o mesmo papel da classe `Lag2`, que exemplificamos anteriormente nas listagens 6.4 e 6.5.

Como podemos observar na listagem, entre as linhas 5 e 12 programamos no mecanismo de expressões templates como avaliar as bases no caso de um único vetor com duas componentes. Observando o trecho que especializa `UnaryReturn` (linhas 30 até 33), vemos que, para um vetor de entrada de duas componentes, atuando em conjunto com a classe `LagrangeTriangle_P1_op`, o resultado (`Type_t`) é um vetor de três componentes (uma para cada função da base). A chamada da linha 11 constrói este vetor de três componentes (diretamente através de seu construtor) já o inicializando com `(1-v(0)-v(1),v(0),v(1))`, ou ainda, correspondentemente, com $(1 - \xi - \eta, \xi, \eta)$.

Para finalizar, entre as linhas 16 e 26 instruímos o mecanismo de expressões templates como avaliar este operador em um array de vetores de duas componentes. Com este passo, conseguimos definir a base inteiramente dentro dos mecanismos do POOMA.

Devemos notar que a classe `LagrangeTriangle_P1_op` é completamente atômica. O que significa que não depende nem deriva de nenhuma outra. É uma classe essencialmente *de cálculo*. Para fazermos

```
1  // Implementação da base triangular de Lagrange de ordem 1.
2  class LagrangeTriangle_P1_op {
3    public:
4      // Avaliando a base em um único ponto bidimensional.
5      template< class T, class E > inline typename
6      UnaryReturn< Vector< 2, T, E >, LagrangeTriangle_P1_op >::Type_t
7      operator()( const Vector< 2, T, E >& v ) const {
8        typedef typename UnaryReturn< Vector< 2, T, E >,
9          LagrangeTriangle_P1_op >::Type_t Type_t;
10
11       return Type_t( 1.0 - v( 0 ) - v( 1 ), v( 0 ), v( 1 ) );
12     }
13
14     // Ensinando o mecanismo de expressões templates a avaliar a base
15     // em um array de pontos.
16     template< int D, class T, class E > inline typename
17     MakeReturn< UnaryNode<
18       LagrangeTriangle_P1_op, typename CreateLeaf< Array< D, T, E > >::Leaf_t >
19     >::Expression_t
20     operator()( const Array< D, T, E >& l ) const {
21       typedef UnaryNode< LagrangeTriangle_P1_op,
22         typename CreateLeaf< Array< D, T, E > >::Leaf_t > Tree_t;
23
24       return MakeReturn< Tree_t >::make(
25         Tree_t( CreateLeaf< Array< D, T, E > >::make( l ) ) );
26     }
27 };
28
29 // Especialização de UnaryReturn.
30 template< class T, class E >
31 struct UnaryReturn< Vector< 2, T, E >, LagrangeTriangle_P1_op > {
32     typedef Vector< 3, T, E > Type_t;
33 };
```

Listagem 6.6: Construção da base de Lagrange triangular de ordem 1.

"a cola" desta operação no configurador da base (capítulo 5, seção 5.4.1) usaremos outra classe projetada especificamente com esta finalidade. Esta classe chama-se `LagrangeTriangle_P1` e pode ser vista na listagem 6.7.

A única função da classe `LagrangeTriangle_P1` é a de fazer a ponte entre o operador de cálculo efetivo (`LagrangeTriangle_P1_op`) e o configurador de bases. Para tanto, devemos notar que a classe possui dois métodos `eval()` conectados ao mecanismo de expressões templates da mesma forma que `LagrangeTriangle_P1_op`; porém a única coisa que estes métodos fazem é *repassar a operação* para o objeto `EvalOp`. Não por coincidência, `EvalOp` é do tipo `LagrangeTriangle_P1_op`.

Este padrão se repetirá ao longo de OSIRIS:

- Uma classe de cálculo estanque, que apenas opera dados, conectada às expressões templates de POOMA;

- Outra classe que faz a conexão da primeira e dos configuradores.

```
1  // Classe mixin para integração no configurador de bases.
2  template< class generator_t >
3  class LagrangeTriangle_P1 : public generator_t {
4     public:
5        enum { dimensions = 2 }; // Dimensão espacial da base.
6        enum { ndof = 3 };  // Graus de liberdade.
7
8        // Avaliando a base em um único ponto bidimensional.
9        template< class T, class E > inline typename
10       UnaryReturn< Vector< 2, T, E >, LagrangeTriangle_P1_op >::Type_t
11       eval( const Vector< 2, T, E >& v ) const {
12          return EvalOp( v );
13       }
14
15       // Ensinando o mecanismo de expressões templates a avaliar a base
16       // em um array de pontos.
17       template< int D, class T, class E > inline typename
18       MakeReturn< UnaryNode< LagrangeTriangle_P1_op, typename
19                              CreateLeaf< Array< D, T, E > >::Leaf_t >
20       >::Expression_t
21       eval( const Array< D, T, E >& l ) const {
22          return EvalOp( l );
23       }
24
25    private:
26       // Operador de cálculo.
27       LagrangeTriangle_P1_op EvalOp;
28 };
```

Listagem 6.7: Mixin usado na configuração da base triangular de Lagrange de primeira ordem.

As bases para outros tipos de elementos são definidas de maneira inteiramente análoga, observando-se apenas as dimensões espaciais corretas.

Em muitos casos necessitamos avaliar não só as funções de base, mas também seus gradientes. A única diferença entre a avaliação da função e de seu gradiente reside no valor de retorno. No nosso exemplo bidimensional, a avaliação da função era representada por

$$\boldsymbol{\varphi}\left(\begin{bmatrix} \langle\,\bullet\,\rangle_2 & \cdots \\ \vdots & \end{bmatrix}\right) = \left\langle \begin{bmatrix} \bullet & \cdots \\ \vdots & \end{bmatrix} \right\rangle_3$$

enquanto que no caso do gradiente será

$$\boldsymbol{\varphi}\left(\begin{bmatrix} \langle\,\bullet\,\rangle_2 & \cdots \\ \vdots & \end{bmatrix}\right) = \left\langle \begin{bmatrix} \langle\,\bullet\,\rangle_2 & \cdots \\ \vdots & \end{bmatrix} \right\rangle_3 = \left\langle \begin{bmatrix} \langle \partial_x\boldsymbol{\varphi}, \partial_y\boldsymbol{\varphi}\rangle & \cdots \\ \vdots & \end{bmatrix} \right\rangle_3$$

6.3.3 Transformações padrão ↔ real

Os elementos de referência utilizados neste trabalho, bem como suas respectivas transformações padrão ↔ real, jacobianos e determinantes de jacobianos são descritos no apêndice A.

Para exemplificar, tomemos a transformação triângulo[8] padrão ↔ real, como em (A.1). Repetiremos a expressão da transformação por conveniência;

$$\begin{bmatrix} x(\xi,\eta) \\ y(\xi,\eta) \end{bmatrix} = \begin{bmatrix} x_2 - x_1 & x_3 - x_1 \\ y_2 - y_1 & y_3 - y_1 \end{bmatrix} \begin{bmatrix} \xi \\ \eta \end{bmatrix} + \begin{bmatrix} x_1 \\ y_1 \end{bmatrix},$$

com $\boldsymbol{b} = [x_1, y_1]^T$.

Da expressão acima observamos diversos elementos que aparecerão de forma consistente ao longo dos cálculos. O primeiro deles é o próprio jacobiano. O segundo é o vetor $\boldsymbol{b} = [x_1, y_1]^T$. Não tão óbvio mas igualmente importante é o determinante do jacobiano. Naturalmente, o mesmo se aplica à transformação inversa. Estes cálculos se repetem para cada elemento da malha, mas *não necessitam ser refeitos* em cada operador diferencial. Basta fazê-los uma única vez e aproveitar o resultado em todos os operadores diferenciais.

Com esta finalidade criamos algumas classes que servem exclusivamente para cálculo e armazenamento dessas quantidades. Ressaltamos que tais classes não fazem parte do mecanismo de expressões templates, pois seus cálculos são executados uma única vez durante a simulação.

A título de exemplo, apresentamos a estrutura `TriangleMeasure` (listagem 6.8). Ela é responsável por calcular e armazenar o jacobiano da transformação no triângulo de referência, seu determinante e também o jacobiano da transformação inversa.

Observando-se a citada listagem, é fácil perceber que a estrutura `TriangleMeasure` é bastante simples. Dada uma malha de elementos finitos e um número global de elemento, seus três vértices são recuperados, o jacobiano construído, seu determinante calculado, este valor e também o vetor $\boldsymbol{b}$ são armazenados. Todos os seus dados são públicos, facilitando o seu acesso.

A representação gráfica das transformações é simples: um ponto n-dimensional, quando transformado, continua sendo um ponto n-dimensional. A tabela 6.2 resume as possibilidades de operação. Para o jacobiano, temos a tabela 6.3.

Vamos finalmente apresentar a listagem 6.9, que traz a classe `L2G_Triangle_op`, que efetivamente implementa a transformação padrão → real no caso de elementos triangulares. Entre as linhas 11 e 15 e também entre as linhas 19 e 29 temos a inserção da operação no mecanismo de expressões templates, como já feito anteriormente. É interessante observarmos que a operação $\boldsymbol{J}(\boldsymbol{x})\boldsymbol{x} + \boldsymbol{b}$ é feita usando-se operadores próprios da POOMA (soma e `dot()`). Desta forma garantimos o uso absoluto do mecanismo de expressões templates.

As novidades nesta listagem são o método `setMeasure`, que associa à transformação um objeto do tipo `TriangleMeasure`. Poderíamos inicialmente argumentar que este objeto poderia ser parte integrante da classe, mas claramente esta não é uma boa solução. Podemos de fato ter, por exemplo, elementos de Lagrange de primeira e segunda ordem, coexistindo em um mesmo problema, um mesmo

[8] De fato, poderíamos usar outras geometrias como o tetraedro ou o quadrado, mas nos ateremos ao triângulo por simplicidade.

```
1  // Estrutura auxiliar com diversas ‘‘medidas’’ no triângulo.
2  struct TriangleMeasure {
3    template< class mesh_t >
4    inline void setup( mesh_t& M, uinteger_t id ) {
5      typedef mesh_t Mesh_t; // Malha de elementos finitos.
6
7      typedef typename Mesh_t::Element_t Element_t;
8      typedef typename Mesh_t::PositionsArray_t PositionsArray_t;
9      typedef typename Mesh_t::PointType_t PointType_t;
10
11     const Element_t& E = M.elements()( id );
12
13     const PositionsArray_t& V = M.vertexPositions();
14
15     // Coordenadas do triângulo atual.
16     const PointType_t& P1 = V( E.references()( 0 ) );
17     const PointType_t& P2 = V( E.references()( 1 ) );
18     const PointType_t& P3 = V( E.references()( 2 ) );
19
20     real_t a = P2( 0 ) - P1( 0 ); // x2 - x1
21     real_t b = P3( 0 ) - P1( 0 ); // x3 - x1
22     real_t c = P2( 1 ) - P1( 1 ); // y2 - y1
23     real_t d = P3( 1 ) - P1( 1 ); // y3 - y1
24
25     // Jacobiano.
26     J( 0, 0 ) = a; // x2 - x1
27     J( 1, 0 ) = c; // y2 - y1
28     J( 0, 1 ) = b; // x3 - x1
29     J( 1, 1 ) = d; // y3 - y1
30
31     xy1  = P1;   // b = [x_1, y_1]^T
32     jDet = det( J ); // Determinante do jacobiano.
33
34     // Jacobiano da transformação inversa.
35     Ji( 0, 0 ) = d / jDet; // y3 - y1
36     Ji( 1, 0 ) = -c / jDet; // y1 - y2
37     Ji( 0, 1 ) = -b / jDet; // x1 - x2
38     Ji( 1, 1 ) = a / jDet; // x3 - x1
39   }
40
41   Tensor< 2, real_t, Full > J; // Tensor que armazena o Jacobiano.
42   Tensor< 2, real_t, Full > Ji; // Tensor que armazena o Jacobiano inverso.
43   Vector< 2, real_t, Full > xy1; // Vetor que armazena b = [x_1, y_1]^T .
44   real_t jDet; // Determinante do Jacobiano.
45 };
```

Listagem 6.8: Classe auxiliar para geometria triangular.

modelo e até mesmo em uma mesma expressão. Apesar da diferença na ordem de aproximação, a transformação continua sendo a mesma. Se os cálculos são feitos por um único objeto ligado externamente à

Transformação $T : A \to B$		
Dimensão	A	B
1	$[\bullet \cdots]$	$[\bullet \cdots]$
2	$\begin{bmatrix} \langle \bullet \rangle_2 & \cdots \\ \vdots & \end{bmatrix}$	$\begin{bmatrix} \langle \bullet \rangle_2 & \cdots \\ \vdots & \end{bmatrix}$
3	$\ulcorner \begin{bmatrix} \langle \bullet \rangle_3 & \cdots \\ \vdots & \end{bmatrix} \urcorner \lrcorner$	$\ulcorner \begin{bmatrix} \langle \bullet \rangle_3 & \cdots \\ \vdots & \end{bmatrix} \urcorner \lrcorner$

Tabela 6.2: Resultados do cálculo do jacobiano de uma transformação padrão $\leftrightarrow$ real.

Jacobiano $\boldsymbol{J} : A \to B$		
Dimensão	A	B
1	$[\bullet \cdots]$	$[\bullet \cdots]$
2	$\begin{bmatrix} \langle \bullet \rangle_2 & \cdots \\ \vdots & \end{bmatrix}$	$\begin{bmatrix} [\![\bullet]\!]_2 & \cdots \\ \vdots & \end{bmatrix}$
3	$\ulcorner \begin{bmatrix} \langle \bullet \rangle_3 & \cdots \\ \vdots & \end{bmatrix} \urcorner \lrcorner$	$\ulcorner \begin{bmatrix} [\![\bullet]\!]_3 & \cdots \\ \vdots & \end{bmatrix} \urcorner \lrcorner$

Tabela 6.3: Resultados da aplicação do jacobiano uma transformação padrão $\leftrightarrow$ real.

transformação, ele pode ser aproveitado em todas elas, evitando o recálculo do jacobiano, por exemplo.

Um grande cuidado deve ser tomado na definição de operações que contenham dados, como é o caso de `L2G_Triangle_op`. Quando o mecanismo de expressões templates vai avaliar uma expressão, objetos relacionados às operações são constantemente criados, *usando-se um construtor padrão, vazio*. Em outras palavras, dados dos objetos representantes daquela classe serão perdidos. A única solução por nós encontrada foi a transformação dos dados das classes operacionais em dados do tipo *estático*. Um dado estático (declarado através da palavra chave **`static`** é compartilhado entre todos os objetos de uma mesma classe. Assim, quando a POOMA instancia um objeto `L2G_Triangle_op`, o dado `Tm` está seguramente associado ao objeto `TriangleMeasure` correto. Este detalhe original aparentemente pequeno, mas de graves conseqüências, foi a última barreira a ser vencida para a total integração do mecanismo de expressões templates de POOMA em OSIRIS.

Assim como no caso das bases de elementos finitos (seção 6.3.2), esta classe é apenas de cálculo, estando completamente dissociada dos configuradores. A conexão é feita através de uma classe muito similar à apresentada na listagem 6.7, que portanto não será esmiuçada aqui.

Os operadores que inserem o jacobiano e seu determinante fazem parte da listagem 6.10; a trans-

```
 1  // Transformação padrão → real.
 2  class L2G_Triangle_op {
 3    public:
 4      // Guarda um objeto do tipo TriangleMeasure.
 5      inline void setMeasure( const TriangleMeasure& tm ) {
 6        Tm = &const_cast< TriangleMeasure& >( tm );
 7      }
 8
 9      // Transformando um único ponto bidimensional.  Notemos o uso do operador dot()
10      // do POOMA.
11      template< class T, class E > inline typename
12      UnaryReturn< Vector< 2, T, E >, L2G_Triangle_op >::Type_t
13      eval( const Vector< 2, T, E >& v ) const {
14        return dot( Tm->J, v ) + Tm->xy1;
15      }
16
17      // Ensinando o mecanismo de expressões templates a avaliar a base
18      // em um array de pontos.
19      template< int D, class T, class E > inline typename
20      MakeReturn< UnaryNode< L2G_Triangle_op, typename
21                             CreateLeaf< Array< D, T, E > >::Leaf_t >
22      >::Expression_t
23      eval( const Array< D, T, E >& l ) const {
24        typedef UnaryNode< L2G_Triangle_op,
25          typename CreateLeaf< Array< D, T, E > >::Leaf_t > Tree_t;
26
27        return MakeReturn< Tree_t >::make(
28          Tree_t( CreateLeaf< Array< D, T, E > >::make( l ) ) );
29      }
30
31    protected:
32      // A classe que calcula jacobianos e outras quantidades úteis.
33      static TriangleMeasure* Tm;
34  };
35
36  // Inicializando o membro estático.
37  TriangleMeasure* L2G_Triangle_op::Tm = 0;
38
39  // Especializando UnaryReturn.
40  template< class T, class E >
41  struct UnaryReturn< Vector< 2, T, E >, L2G_Triangle_op > {
42      typedef Vector< 2, T, E > Type_t;
43  };
```

Listagem 6.9: Transformação padrão → real para triângulos.

formação inversa, seu jacobiano e respectivo determinante são completamente análogos aos já apresentados.

```
1  // Jacobiano da transformação padrão → real.
2  class J_L2G_Triangle_op {
3    public:
4      // Guarda um objeto do tipo TriangleMeasure.
5      inline void setMeasure( const TriangleMeasure& tm ) {
6        Tm = &const_cast< TriangleMeasure& >( tm );
7      }
8
9      // Retorna o jacobiano para um único ponto.
10     template< class T, class E >
11     inline Tensor< 2, real_t, Full >& j( const Vector< 2, T, E >& l ) const {
12       return Tm->J;
13     }
14
15     // Retorna o jacobiano para um array de pontos.
16     template< int D, class T, class E >
17     inline Tensor< 2, real_t, Full >& j( const Array< D, T, E >& l ) const {
18       return Tm->J;
19     }
20
21     // Retorna o determinante do jacobiano para um único ponto.
22     template< class T, class E >
23     inline real_t jD( const Vector< 2, T, E >& l ) const {
24       return Tm->jDet;
25     }
26
27     // Retorna o determinante do jacobiano para um array de pontos.
28     template< int D, class T, class E >
29     inline real_t jD( const Array< D, T, E >& l ) const {
30       return Tm->jDet;
31     }
32
33   protected:
34     // A classe que calcula jacobianos e outras quantidades úteis.
35     static TriangleMeasure* Tm;
36 };
37
38 // Inicializando o membro estático.
39 TriangleMeasure* J_L2G_Triangle_op::Tm = 0;
```

Listagem 6.10: Jacobiano da transformação padrão → real para triângulos.

6.4 Configurando os operadores

6.4.1 DSL para operadores

Seguindo as diretivas apresentadas no capítulo 5, vamos agora definir a DSL (Domain Specific Language) para o caso da configuração de operadores diferenciais em OSIRIS. Partiremos do diagrama da figura 6.5, que exibe os aspectos de interesse na configuração.

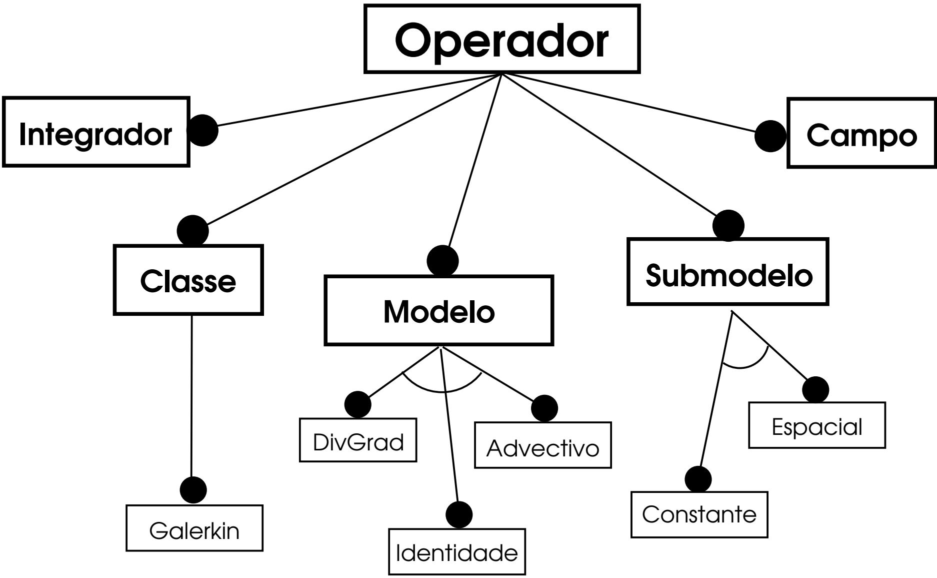


Figura 6.5: DSL para operadores diferenciais.

Estes aspectos de interesse são: CLASSE, MODELO, SUBMODELO, CAMPO e INTEGRADOR.

Classe O aspecto CLASSE diz respeito ao tipo de aproximação desejada quando da construção do operador. Na atual versão de OSIRIS existe uma única opção que faz com que os operadores assumam o *método de Galerkin* (funções teste e de base coincidem). Futuras implementações podem incluir outras aproximações.

Modelo Aqui decidimos que tipo de operador estamos usando. Dentre os disponíveis temos *divgrad*, correspondente aos operadores de difusão (espalhamento espacial, vide seção 7.2), *identidade*, correspondente aos operadores de decaimento (seção 7.3) e *advectivo*, relacionado aos operadores de advecção (seção 7.4).

Submodelo Cada operador descrito acima possui pelo menos um coeficiente (*cf.* as seções correspondentes) que pode ser constante ou variar espacialmente. O aspecto SUBMODELO permite escolher entre estas duas opções. Note-se que outros tipos de parâmetro podem futuramente ser definidos, como, por exemplo, parâmetros dependentes do tempo e não do espaço.

Campo e integrador CAMPO e INTEGRADOR não são exatamente aspectos, mas sim "espaços deixados em branco" na criação do operador. Estes espaços serão preenchidos com o campo (malha + valores) onde o operador atuará e com o integrador numérico usado nas avaliações.

É interessante observar que a construção do configurador completo para os operadores diferenciais segue o mesmo padrão descrito em 5. Definimos uma DSL (como acima), criamos um parser de configurações e finalmente o configurador completo. Um exemplo de uso deste configurador pode ser apreciado na listagem 6.11, onde configuramos um operador do tipo laplaciano $-\alpha\Delta u$ com α constante. Mais exemplos de configurações de operadores estão no capítulo 8.

```
1  // A dimensão espacial do problema (3D, no caso).
2  const int D = 3;
3
4  // Um campo e um integrador fictícios, a título de exemplo.
5  typedef MyField Field_t;
6  typedef MyIntegrator Integrator_t;
7
8  // Operador laplaciano com coeficiente constante.
9  typedef LOCOP_GENERATOR< // Esta é a classe configuradora.
10   LOCOP_Class< galerkin >, // Classe Galerkin.
11   LOCOP_Model< divgrad< D > >, // Modelo divgrad.
12   LOCOP_SubModel< cte_parameter >, // Parâmetro α constante.
13   LOCOP_Field< Field_t >, // O tipo de campo (previamente definido).
14   LOCOP_Integrator< Integrator_t > // O integrador numérico (previamente definido).
15   >::RET CLaplacian_t;
16 };
```

Listagem 6.11: Configuração automática de um operador tipo laplaciano com coeficiente constante.

Com a definição constante da listagem 6.11, `CLaplacian_t` é um tipo válido e pode ser usado para calcular aproximações do laplaciano em qualquer geometria.

6.5 Condições de fronteira

6.5.1 DSL para condições de fronteira

Condições de fronteira também são configuradas automaticamente a partir de aspectos selecionados pelo usuário. Estes aspectos constam da figura 6.5.1, e são TIPO, CAMPO e INTEGRADOR.

Tipo Em TIPO selecionamos qual a condição que queremos aplicar. Na atual implementação de OSIRIS temos duas opções: *Dirichlet* e *Neumann*. A condição de Dirichlet se desdobra em três subcasos, Dirichlet nula (zero na fronteira), Dirichlet constante (valor prescrito na fronteira) e Dirichlet espacialmente variável (função de $\boldsymbol{x}$ na fronteira). Neste último caso temos ainda um aspecto chamado AVALIADOR, que receberá a função responsável por avaliar a condição de fronteira. Por questões de simplicidade, a condição de Neumann tem apenas a variante nula.

Campo e integrador Têm exatamente o mesmo papel de CAMPO e INTEGRADOR para o caso da configuração do operador (seção 6.4). Contudo, devemos observar que em alguns casos o aspecto INTEGRADOR não é necessário (aspecto opcional).

Na listagem 6.12 exemplificamos a construção de uma condição de fronteira do tipo Dirichlet espacialmente variável.

Os operadores que, a título de exemplo, figuram no texto acima serão, no capítulo seguinte, abordados de modo mais sistemático.

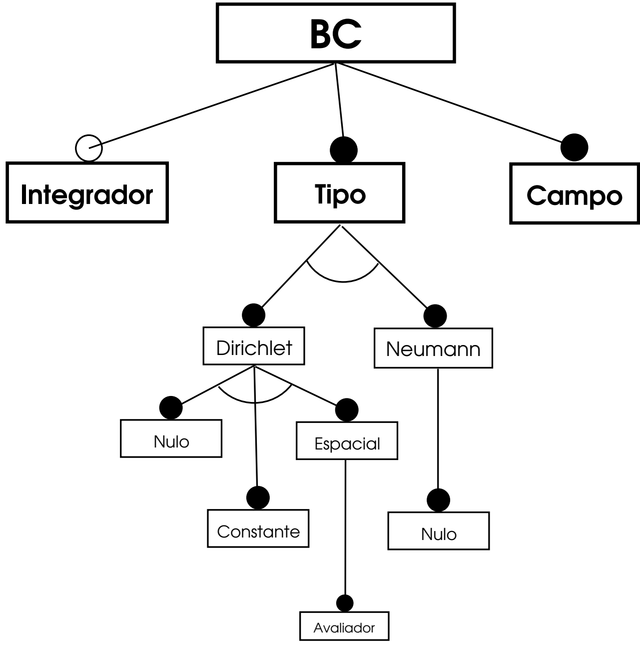


Figura 6.6: DSL para condições de fronteira.

```
1   // A dimensão espacial do problema (3D, no caso).
2   const int D = 3;
3
4   // Um campo fictício, a título de exemplo.
5   typedef MyField Field_t;
6
7   // A função a ser avaliada na fronteira.
8   typedef MyBoundaryFunctor Functor_t;
9
10  // Condição tipo Dirichlet espacialmente variável.
11   typedef BC_GENERATOR< // O configurador de condições de contorno.
12      BC_DirichletEvaluator< Functor_t >, // A função a ser avaliada no contorno.
13      BC_Type< spatial_dirichlet< D > >, // O tipo de condição.
14      BC_Field< Field_t > // O campo sobre o qual estamos resolvendo o problema.
15      >::RET DirichletBC_t; // O resultado final.
16  };
```

Listagem 6.12: Configuração automática de uma condição de contorno tipo Dirichlet espacialmente variável.

CAPÍTULO 7

Operadores essenciais

As classes apresentadas no capítulo 6 são a base para a criação de diversos operadores diferenciais de interesse em várias áreas das Ciências Aplicadas. Neste capítulo introduzimos alguns destes operadores e discutimos sua implementação utilizando o mecanismo de expressões templates de Pooma*, bem como mantendo sua característica de auto-configuração. Esta implementação, além de inovadora, é um grande avanço no cálculo aproximativo dos operadores diferenciais em diversos ambientes.*

7.1 Introdução

Neste capítulo faremos a construção efetiva de alguns operadores diferenciais de interesse em vários campos da Ciência, em particular na Biomatemática. Estes operadores deverão ser construídos baseados no mecanismo de expressões templates e completamente inseridos em um contexto de configuração automática, isolando desta forma o usuário de detalhes que não sejam diretamente relacionados ao problema.

Como mencionamos na seção 6.1, a definição dos operadores será feita de forma local a cada elemento, dada a estrutura do método de elementos finitos.

De forma geral, estamos interessados na aproximação numérica de problemas diferenciais do tipo:

$$\begin{array}{rcll} \mathcal{L}(u) & = & f, & \text{em } \Omega \\ \mathcal{F}(u) & = & g, & \text{em } \partial\Omega, \end{array} \tag{7.1}$$

onde $\Omega \subset \mathbb{R}^n$, com $n = 1, 2, 3$ e $\mathcal{L}$ é um operador diferencial. Dentro do nosso contexto denominaremos de *operador diferencial* a atuação de um único operador na variável (de fato, função) u.

A idéia básica do método de elementos finitos consiste de uma formulação variacional para o problema (7.1), seguida de uma aproximação discretizada da solução desse problema variacional. Esta solução aproximada é construída através da discretização de Ω em unidades geométricas bem definidas (triângulos, quadrados, prismas), denominadas *elementos*. Nosso interesse consiste na definição da atuação de operadores diferenciais em sua forma variacional discreta sobre cada um destes elementos. Assim, recuperando um pouco da notação introduzida em capítulos anteriores:

- Denotaremos por e um elemento real no espaço $\boldsymbol{x}$;

- Um elemento padrão no espaço $\boldsymbol{\xi}$ será denotado $\hat{e}$;
- As funções de base, quando avaliadas em $\hat{e}$, serão denotadas por $\hat{\varphi}_i(\boldsymbol{\xi})$. Quando avaliadas sobre Ω, ou sobre e, serão simplesmente $\varphi_i(\boldsymbol{x})$;
- A aproximação de elementos finitos de uma função u em Ω será denotada por

$$u_h(\boldsymbol{x}) = \sum_j c_j\,\varphi_j(\boldsymbol{x});$$

- Quando restritos a um elemento e, usaremos a notação

$$u_e(\boldsymbol{x}) = \sum_j c_j\,\varphi_j(\boldsymbol{x})$$

  para representar a aproximação de u sobre este elemento e;
- Quando não for contextualmente claro, operadores serão sufixados com $\boldsymbol{x}$ ou $\boldsymbol{\xi}$ indicando, respectivamente, sua atuação nas variáveis globais $\boldsymbol{x}$ ou locais $\boldsymbol{\xi}$.

7.2 Operadores do tipo $\nabla\cdot(-\alpha\nabla u)$

Dada uma função $u : \Omega \subset \mathbb{R}^n \to \mathbb{R}$, para $n = 1, 2, 3$, os operadores que denominaremos genericamente de *operadores difusivos* são da forma apresentada em (7.2).

$$\mathcal{D}(u) = \nabla\cdot(-\alpha\nabla u). \tag{7.2}$$

O operador ∇ é definido de acordo com a dimensão de Ω. O coeficiente de difusão α poderá assumir diversos significados, de acordo com o tipo de operador a ser definido (como veremos adiante).

Este tipo de operador é de suma importância na descrição de fenômenos onde a variável de interesse apresenta um comportamento de espalhamento espacial. Como exemplos de aplicação deste operador citamos a Dinâmica Populacional [dAP02, Din03] e a chamada Ecotoxicologia Matemática [dO03].

7.2.1 Difusão constante

Se tomarmos $\alpha \in \mathbb{R}^*_+$ e constante em (7.2), obtemos o *operador laplaciano*, definido como em (7.3):

$$\mathcal{D}(u) = \nabla\cdot(-\alpha\nabla u) = -\alpha\nabla\cdot\nabla u = -\alpha\Delta u. \tag{7.3}$$

Fazendo-se a formulação fraca para este operador (vide [dAP02], por exemplo), recaímos na forma bilinear da equação (7.4).

$$a(u, v) = \int_\Omega -\alpha\Delta u\, v\, d\boldsymbol{x}, \tag{7.4}$$

com v uma função teste apropriada. Após alguma manipulação algébrica e usando o Teorema de Green, obtemos a forma bilinear como em (7.5), com um possível termo de contorno.

$$a(u, v) = \alpha\int_\Omega \nabla u\cdot\nabla v\, d\boldsymbol{x} \quad + \quad \text{eventuais termos no contorno}. \tag{7.5}$$

Os termos de contorno que podem surgir em (7.5) não nos interessam no momento.

Lembrando que aproximaremos a função u por uma combinação linear dos φ_i's, e que as funções teste v serão funções da própria base (método de Galerkin), pode-se escrever a forma bilinear discreta, como em (7.6).

$$a_h(u_h, v_h) = \alpha \sum_j c_j \int_{\Omega_h} \nabla\varphi_j \cdot \nabla\varphi_i \, d\boldsymbol{x} = \alpha \sum_j c_j \, \langle \nabla\varphi_j \| \nabla\varphi_i \rangle_{\Omega_h}. \tag{7.6}$$

Note-se que já introduzimos a notação de produto interno.

Da construção do método de elementos finitos sabemos que o somatório em (7.6) pode ser calculado elemento a elemento; em outras palavras, o que nos interessa de fato é $\langle \nabla\varphi_j \| \nabla\varphi_i \rangle_e$, ou seja, o produto interno sobre um elemento.

Denotando-se por T a transformação que mapeia o elemento padrão no real (apêndice A), por $\boldsymbol{J}(\boldsymbol{\xi})$ seu jacobiano e por $\boldsymbol{J}^{-1}(\boldsymbol{x})$ o jacobiano da transformação inversa de T, temos que $\varphi_i(\boldsymbol{x}) = \varphi_i(T(\boldsymbol{\xi})) = \hat{\varphi}_i(\boldsymbol{\xi})$, que após alguma manipulação, leva à igualdade (7.7).

$$\nabla_{\boldsymbol{x}} \varphi_i(\boldsymbol{x}) = [\boldsymbol{J}^{-1}(\boldsymbol{\xi})]^T \, \nabla_{\boldsymbol{\xi}} \hat{\varphi}_i(\boldsymbol{\xi}). \tag{7.7}$$

Desta forma, temos:

$$\langle \nabla\varphi_j \| \nabla\varphi_i \rangle_e = \int_e \nabla\varphi_j \cdot \nabla\varphi_i \, d\boldsymbol{x} = \int_{\hat{e}} \Big\{ [\boldsymbol{J}^{-1}(\boldsymbol{\xi})]^T \, \nabla\hat{\varphi}_j \Big\} \cdot \Big\{ [\boldsymbol{J}^{-1}(\boldsymbol{\xi})]^T \, \nabla\hat{\varphi}_i \Big\} \, |\boldsymbol{J}(\boldsymbol{\xi})| \, d\boldsymbol{\xi}. \tag{7.8}$$

Para melhor entendimento de como estas operações serão introduzidas no mecanismo de avaliação de expressões templates, iremos reescrever (7.8) introduzindo algumas marcações na expressão, obtendo assim (7.9).

$$\langle \nabla\varphi_j \| \nabla\varphi_i \rangle_e = \underbrace{\int_{\hat{e}} \overbrace{\Big\{ \underbrace{[\boldsymbol{J}^{-1}(\boldsymbol{\xi})]^T}_{\text{(a)}} \, \underbrace{\nabla\hat{\varphi}_j}_{\text{(b)}} \Big\} \cdot \underbrace{\Big\{ [\boldsymbol{J}^{-1}(\boldsymbol{\xi})]^T \, \nabla\hat{\varphi}_i \Big\}}_{\text{(d)}}}^{\text{(e)}} \, \underbrace{|\boldsymbol{J}(\boldsymbol{\xi})|}_{\text{(c)}} \, d\boldsymbol{\xi}}_{\text{(f)}}. \tag{7.9}$$

A análise de cada uma das operações marcadas (a)-(f) nos dará uma visão clara da conexão entre o mecanismo de expressões templates e os operadores de OSIRIS. Faremos a análise a partir de elementos bidimensionais, mas ela é válida para qualquer das dimensões consideradas.

(a) Jacobiano inverso transposto

A avaliação do jacobiano inverso transposto é feita nos pontos de integração no triângulo padrão. Lembrando a tabela 6.3, temos:

$$[\boldsymbol{J}^{-1}(\boldsymbol{\xi})]^T = \left[\boldsymbol{J}^{-1} \left(\begin{bmatrix} \langle \bullet \rangle_2 & \cdots \\ \vdots & \end{bmatrix} \right) \right]^T = \begin{bmatrix} [\![\bullet]\!]_2^T & \cdots \\ \vdots & \end{bmatrix} =$$

$$= \begin{bmatrix} \texttt{transpose}\big([\![\bullet]\!]_2 \big) & \cdots \\ \vdots & \end{bmatrix} = \begin{bmatrix} [\![\bullet]\!]_2 & \cdots \\ \vdots & \end{bmatrix}.$$

A avaliação do jacobiano já foi introduzida no mecanismo de expressões templates (listagem 6.10, classe `J_L2G_Triangle_op`). A operação de transposição é suportada nativamente pela POOMA (através da função `transpose()`), o que significa que também faz parte do mecanismo de expressões templates.

(b) Avaliação do gradiente da base

Seguindo o exposto na seção 6.3.2, a avaliação do gradiente de uma das funções da base nos pontos de integração é:

$$\nabla \hat{\varphi}_j \left(\begin{bmatrix} \langle \bullet \rangle_2 & \cdots \\ \vdots & \end{bmatrix} \right) = \begin{bmatrix} \langle \bullet \rangle_2 & \cdots \\ \vdots & \end{bmatrix},$$

já devidamente integrada ao mecanismo de expressões templates (listagem 6.6).

(c) Determinante do jacobiano

Novamente de acordo com a tabela 6.3, temos:

$$|\boldsymbol{J}(\boldsymbol{\xi})| = \left| \boldsymbol{J} \left(\begin{bmatrix} \langle \bullet \rangle_2 & \cdots \\ \vdots & \end{bmatrix} \right) \right| = \begin{bmatrix} |\, [\![\bullet]\!]_2 \,| & \cdots \\ \vdots & \end{bmatrix} = \begin{bmatrix} \texttt{det}\Big([\![\bullet]\!]_2 \Big) & \cdots \\ \vdots & \end{bmatrix} = \begin{bmatrix} \bullet & \cdots \\ \vdots & \end{bmatrix}.$$

A função `det()`, que calcula o determinante de um tensor, é implementada dentro do mecanismo de expressões templates, com a adição de alguns metaprogramas especiais que fazem com que seja expandida em tempo de compilação.

(d) Correção do gradiente

O termo (d) representa a aplicação do jacobiano inverso transposto no gradiente da função de base $\hat{\varphi}_j$ e que, de fato, é uma multiplicação matriz por vetor. Tendo já calculado o jacobiano em (a) e o gradiente em (b), resta agora operá-los juntos, como no diagrama abaixo:

$$[\boldsymbol{J}^{-1}(\boldsymbol{\xi})]^T \, \nabla \hat{\varphi}_i = \texttt{dot} \left(\begin{bmatrix} [\![\bullet]\!]_2 & \cdots \\ \vdots & \end{bmatrix}, \begin{bmatrix} \langle \bullet \rangle_2 & \cdots \\ \vdots & \end{bmatrix} \right) =$$

$$= \begin{bmatrix} \texttt{dot}\Big([\![\bullet]\!]_2 , \langle \bullet \rangle_2 \Big) & \cdots \\ \vdots & \end{bmatrix} = \begin{bmatrix} \langle \bullet \rangle_2 & \cdots \\ \vdots & \end{bmatrix}.$$

A operação `dot()` faz o produto interno entre vetores ou o produto de matriz por vetor entre tensores e vetores. Sua implementação é especializada, sendo portanto extremamente eficiente. Vale notar que, até o momento, todas as operações estão sendo realizadas em conjunto, via expressões templates.

(e) Produto interno entre gradientes corrigidos

A operação de transformação do gradiente (d) resulta em um array bidimensional de vetores para cada função de base $\hat{\varphi}_i$ e $\hat{\varphi}_j$. O produto interno entre os dois será:

$$\left\{[\boldsymbol{J}^{-1}(\boldsymbol{\xi})]^T \nabla\hat{\varphi}_j\right\} \cdot \left\{[\boldsymbol{J}^{-1}(\boldsymbol{\xi})]^T \nabla\hat{\varphi}_i\right\} = \texttt{dot}\left(\begin{bmatrix} \langle\bullet\rangle_2 & \cdots \\ \vdots & \end{bmatrix}, \begin{bmatrix} \langle\bullet\rangle_2 & \cdots \\ \vdots & \end{bmatrix}\right)$$

$$= \begin{bmatrix} \texttt{dot}\Big(\langle\bullet\rangle_2, \langle\bullet\rangle_2\Big) & \cdots \\ \vdots & \end{bmatrix} = \begin{bmatrix} \bullet & \cdots \\ \vdots & \end{bmatrix}.$$

(f) Multiplicação pelo determinante do jacobiano e integração numérica

O último passo de cálculo consiste em multiplicar o resultado de (e) pelo de (c) e integrar numericamente. A integral numérica é obtida através da multiplicação pelos pesos e posterior somatório. Acompanhemos o diagrama:

$$\int_{\hat{e}} \left\{[\boldsymbol{J}^{-1}(\boldsymbol{\xi})]^T \nabla\hat{\varphi}_j\right\} \cdot \left\{[\boldsymbol{J}^{-1}(\boldsymbol{\xi})]^T \nabla\hat{\varphi}_i\right\} |\boldsymbol{J}(\boldsymbol{\xi})|\, d\boldsymbol{\xi} = \texttt{sum}\left(\underbrace{\begin{bmatrix} \bullet & \cdots \\ \vdots & \end{bmatrix}}_{(e)} * \underbrace{\begin{bmatrix} \bullet & \cdots \\ \vdots & \end{bmatrix}}_{(c)} * \underbrace{\begin{bmatrix} \bullet & \cdots \\ \vdots & \end{bmatrix}}_{w}\right) =$$

$$= \texttt{sum}\left(\begin{bmatrix} \bullet & \cdots \\ \vdots & \end{bmatrix}\right) = \bullet.$$

É importante notarmos que a operação de multiplicação na POOMA *é sempre tensorial, nunca matricial*; em outras palavras, se `A`, `B` e `C` são arrays do POOMA, `C = A * B` é tal que $c_{ij} = a_{ij} b_{ij}$ (vale o mesmo em todas as dimensões). Finalmente, chegamos à função `sum()`, que soma sequencialmente todos os elementos de um array, de onde resulta a integração numérica de (7.8).

O único detalhe que falta é a multiplicação pela constante α. Entretanto este passo é trivial, uma vez que o resultado final da operação (f) é um número real.

Mas o mais importante de tudo é termos a percepção de que *todas estas operações estão sendo executadas dentro do mecanismo de expressões templates*, o que resultará em *um único loop de avaliação* para todos os pontos de integração dentro de um elemento. Este fato ficará mais claro na listagem que mostra a implementação do operador.

A título de curiosidade apresentamos a expressão (7.10), que é a versão final da expressão para o laplaciano com todos os templates resultantes expandidos. Para obtermos (7.10), consideramos um elemento triangular, base de Lagrange de segunda ordem e dois pontos de integração em cada direção. Naturalmente, a escolha por mais pontos de integração resultará em expressões maiores. A integral 7.8

é calculada neste exemplo para $i = j = 1$.

$$\begin{aligned}\langle \nabla\varphi_1 \| \nabla\varphi_1 \rangle_e \quad &= [(y_3 - y_2)^2 + (x_2 - x_3)^2] * \{w_{11}\,[4 * (p_{x_1} + p_{y_1}) - 3]^2 + w_{12}\,[4 * (p_{x_1} + p_{y_2}) - 3]^2 + \\ &+ w_{21}\,[4 * (p_{x_2} + p_{y_1}) - 3]^2 + w_{22}\,[4 * (p_{x_2} + p_{y_2}) - 3]^2\} * \\ &* [(x_2 - x_1)(y_3 - y_1) - (x_3 - x_1)(y_2 - y_1)]\end{aligned} \tag{7.10}$$

Gerando combinações

Antes da análise da listagem, devemos chamar a atenção para mais uma operação envolvendo estruturas de dados e o mecanismo de expressões templates: a *operação de combinação* `combineDot()`, que não está representada em (7.9).

Como dissemos na seção 6.3.2, a avaliação da base resulta em um vetor da POOMA (classe `Vector`) com tantas componentes quantas são as funções de base. Cada componente é, naturalmente, uma das funções de base avaliada nos pontos de integração (vale o mesmo para os gradientes), o que se pode verificar consultando-se as representações (6.6), (6.7) e (6.8).

Desta forma, a transformação do gradiente (operação (d)), descrita nas seções anteriores, é aplicada a todas as funções de base. Assim, o resultado é um vetor de gradientes corrigidos para $i = 1, \ldots, n_b$, com n_b sendo o número de funções de base. O papel de `combineDot()` é produzir todas as combinações de produtos internos entre os gradientes corrigidos. Matematicamente falando, dado um vetor com gradientes como em (7.11)

$$\boldsymbol{v} = \left\langle \nabla\hat{\phi}_1, \nabla\hat{\phi}_2, \ldots, \nabla\hat{\phi}_{n_b} \right\rangle, \tag{7.11}$$

onde $\nabla\hat{\phi}_i = [\boldsymbol{J}^{-1}]^T \nabla\hat{\varphi}_i$, a função `combineDot()` resulta em um tensor do tipo[1] (7.12).

$$\texttt{combineDot}(\boldsymbol{v}) = \begin{pmatrix} \texttt{dot}(\nabla\hat{\phi}_1, \nabla\hat{\phi}_1) & \texttt{dot}(\nabla\hat{\phi}_1, \nabla\hat{\phi}_2) & \cdots & \texttt{dot}(\nabla\hat{\phi}_1, \nabla\hat{\phi}_{n_b}) \\ \texttt{dot}(\nabla\hat{\phi}_2, \nabla\hat{\phi}_1) & \texttt{dot}(\nabla\hat{\phi}_2, \nabla\hat{\phi}_2) & \cdots & \texttt{dot}(\nabla\hat{\phi}_2, \nabla\hat{\phi}_{n_b}) \\ \vdots & \vdots & \cdots & \vdots \\ \texttt{dot}(\nabla\hat{\phi}_{n_b}, \nabla\hat{\phi}_1) & \texttt{dot}(\nabla\hat{\phi}_{n_b}, \nabla\hat{\phi}_2) & \cdots & \texttt{dot}(\nabla\hat{\phi}_{n_b}, \nabla\hat{\phi}_{n_b}) \end{pmatrix}. \tag{7.12}$$

A multiplicação pelo determinante da matriz jacobiana e posterior integração numérica (operação (f)) é feita em cada um dos componentes do tensor (7.12). No final terminamos não com um único real, mas com um tensor de reais, com cada componente (i, j) sendo a aproximação numérica via integração gaussiana de $\langle \nabla\varphi_j \| \nabla\varphi_i \rangle_e$.

Implementação

A implementação do operador laplaciano com α constante está na listagem 7.1. Seguindo o padrão de listagens anteriores, estamos nos concentrando no aspecto puramente numérico. A classe de implementação do operador `CteLocalLaplacian_op` tem uma contraparte chamada `CteLocalLaplacian` que faz a conexão entre o mecanismo de expressões templates e os configuradores automáticos.

[1] No caso específico do operador laplaciano com α constante, este tensor é simétrico. Este caso é automaticamente tratado pois a POOMA dispõe de tensores simétricos.

Esta listagem difere pouco do padrão apresentado até agora para introdução de novos operadores no mecanismo de expressões templates. Entre as linhas 4 e 8 temos o construtor padrão, que recebe uma base de elementos finitos, uma transformação padrão $\leftrightarrow$ real. Devemos mais uma vez alertar para o fato de que os dados da classe *são todos estáticos*, devido à forma como a POOMA implementa as expressões templates.

Entre as linhas 10 e 19 "ensinamos" ao mecanismo de expressões templates como operar sobre uma entrada que consiste de um vetor (linha 12). A dimensão do vetor é também um parâmetro template, pois o operador laplaciano deve ser independente da dimensão. De fato, há também uma definição para o caso de uma entrada real, para o caso unidimensional, mas não está na listagem por economia de espaço.

Na linha 17 vemos toda a operacionalização do cálculo do laplaciano, exceto a integração numérica (feita em um módulo separado). O comando `transpose( M->inverseMap().j( a ) )` calcula o jacobiano inverso no ponto `a` (um dos pontos de integração) e o transpõe. Já `B->grad( a )` calcula o gradiente de todas as funções de base em `a`, para posterior combinação com o jacobiano inverso através de `dot()`. Finalmente o produto da função `dot()` é combinado através de `combineDot()`, multiplicado pelo determinante do jacobiano (`fabs( M->directMap().jD( a ) )`) em `a` e multiplicado pela constante `C` (a difusibilidade α).

Entre as linhas 21 e 31 "ensinamos" o mecanismo de expressões templates a tratar um array POOMA, nos moldes do que já foi apresentado em seções anteriores. Em 34 temos o início do bloco de dados (todos estáticos), representando a base, a transformação padrão $\leftrightarrow$ real e a constante α. Enfim, entre as linhas 41 e 44 especializamos `UnaryReturn` para que o resultado da aplicação do operador laplaciano resulte em um tensor com o número de componentes igual ao número de funções de base (`basis_t::ndof`) e simétrico `Symmetric`).

Com isto temos o operador laplaciano com difusão α constante completamente definido.

7.2.2 Difusão espacial

A primeira sofisticação a ser introduzida em (7.2) é a de tornar o coeficiente de difusão espacialmente variável

$$\alpha = \alpha(\boldsymbol{x}).$$

Do ponto de vista biológico ou ambiental, por exemplo, estaríamos modelando um meio em que a dispersão de uma população ou de um produto tóxico não acontece de forma espacialmente homogênea no mesmo [dAP02].

Assim, se tomarmos $\alpha : \Omega \to \mathbb{R}_+^*$ em (7.2), obtemos o *operador de difusão espacialmente variável*, definido como em (7.13).

$$\mathcal{D}(u) = \nabla \cdot (-\alpha(\boldsymbol{x}) \nabla u). \tag{7.13}$$

Repetindo o processo de formulação fraca e usando o Teorema de Green (*cf.* equações (7.4) e (7.5)), obtemos a forma bilinear para (7.13), como em (7.14).

$$a(u, v) = \int_\Omega \alpha(\boldsymbol{x}) \nabla u \cdot \nabla v \, d\boldsymbol{x} \quad + \quad \text{eventuais termos no contorno.} \tag{7.14}$$

Novamente postergaremos o tratamento dos termos de contorno para nos concentrarmos no operador.

```
template< class basis_t, class mapping_t >
class CteLocalLaplacian_op {
  public:
    inline CteLocalLaplacian_op( const basis_t& b, const mapping_t& m, const real_t& c ) {
      B = &const_cast< basis_t& >( b ); // Armazenando um objeto da base de EF.
      M = &const_cast< mapping_t& >( m ); // Idem para a transformação L2G.
      C = c;   // A constante α.
    }

    template< int D, class T, class E > inline typename
    UnaryReturn< Vector< D, T, E >, CteLocalLaplacian_op< basis_t, mapping_t > >::Type_t
    operator()( const Vector< D, T, E >& a ) const {
      typedef typename UnaryReturn< Vector< D, T, E >,
        CteLocalLaplacian_op< basis_t, mapping_t > >::Type_t Type_t;

      // Efetua a operação para um ponto de integração.
      return C * combineDot( dot( transpose( M->inverseMap().j( a ) ), B->grad( a ) ) ) *
          fabs( M->directMap().jD( a ) );
    }

    template< int D1, int D2, class T, class E, class EA > inline typename
    MakeReturn< UnaryNode< CteLocalLaplacian_op< basis_t, mapping_t >,
      typename CreateLeaf< Array< D2, Vector< D1, T, E >, EA > >::Leaf_t > >::Expression_t
    eval( const Array< D2, Vector< D1, T, E >, EA >& a ) const {
      typedef UnaryNode< CteLocalLaplacian_op< basis_t, mapping_t >,
        typename CreateLeaf< Array< D2, Vector< D1, T, E >, EA > >::Leaf_t > Tree_t;

      // Introduz o operador no mecanismo de expressões templates.
      return MakeReturn< Tree_t >::make(
       Tree_t( CreateLeaf< Array< D2, Vector< D1, T, E >, EA > >::make( a ) ) );
    }

  private:
    static basis_t* B;
    static mapping_t* M;
    static real_t   C;
};

// ``Ensinando'' o mecanismo de expressões templates qual o resultado da operação.

template< int D, class T, class E, class basis_t, class mapping_t >
struct UnaryReturn< Vector< D, T, E >, CteLocalLaplacian_op< basis_t, mapping_t > > {
    typedef Tensor< basis_t::ndof, T, Symmetric > Type_t;
};
```

Listagem 7.1: Operador laplaciano local com α constante.

A forma bilinear (7.14) tem seu equivalente discreto como em (7.15).

$$a_h(u_h, v_h) = \sum_j c_j \int_{\Omega_h} \alpha(\boldsymbol{x})\nabla\varphi_j \cdot \nabla\varphi_i \, d\boldsymbol{x} = \sum_j c_j \langle \alpha(\boldsymbol{x})\nabla\varphi_j \| \nabla\varphi_i \rangle_{\Omega_h}. \tag{7.15}$$

De (7.15) extraímos o termo que nos interessa (sobre um elemento): $\langle \alpha(\boldsymbol{x}) \nabla \varphi_j \| \nabla \varphi_i \rangle_e$. É natural que este termo seja bastante parecido com sua contraparte com difusão α constante. De fato, o tratamento numérico é similar, com os devidos cuidados com $\alpha(\boldsymbol{x})$.

Lembrando a relação (7.7) entre os gradientes no elemento real e no elemento padrão, podemos escrever:

$$\begin{aligned} \langle \alpha(\boldsymbol{x}) \nabla \varphi_j \| \nabla \varphi_i \rangle_e &= \int_e \alpha(\boldsymbol{x}) \nabla \varphi_j \cdot \nabla \varphi_i \, d\boldsymbol{x} \\ &= \int_{\hat{e}} \alpha\big(T(\boldsymbol{\xi})\big) \left\{ [\boldsymbol{J}^{-1}(\boldsymbol{\xi})]^T \, \nabla \hat{\varphi}_j \right\} \cdot \left\{ [\boldsymbol{J}^{-1}(\boldsymbol{\xi})]^T \, \nabla \hat{\varphi}_i \right\} |\boldsymbol{J}(\boldsymbol{\xi})| \, d\boldsymbol{\xi}, \end{aligned} \tag{7.16}$$

expressão esta muito similar a (7.8).

Calculando α

A única diferença no cálculo de (7.8) e (7.16) reside na introdução de $\alpha(\boldsymbol{x})$, que deve ser levada em conta com certo cuidado. Para que o mecanismo de expressões templates funcione a contento, a expressão que calcula α a partir de $\boldsymbol{x}$ *também deve estar no contexto das expressões templates*. Num primeiro momento estaremos focados em funções que relacionam algebricamente α e $\boldsymbol{x}$; na seção 7.5 veremos como associar o cálculo de α a outras variáveis do problema através de interpolação.

Osiris provê algumas macros que facilitam a definição de funções já inseridas no contexto das expressões templates. Para exemplificar, consideremos o domínio $\Omega = [0,1] \times [0,1]$, com a função $\alpha(\boldsymbol{x})$ dada por (7.17). Seu gráfico e curvas de densidade compõe a figura 7.1.

$$\alpha(\boldsymbol{x}) = 10^{-5} \left(1 + \frac{1}{1 + 10\,[(x-1)^2 + (y-1)^2]} \right). \tag{7.17}$$

A listagem 7.2 exemplifica o uso das macros para a definição de uma função que recebe um vetor de duas coordenadas e devolve um real.

As macros em questão são:

`VD2S_UNARY_FUNCTION_ATOM` – linha 4 Esta macro define o comportamento da função quando recebe um vetor. Note-se que passamos para a macro o nome da classe `RoundCorner` e a dimensão do vetor esperado (2).

`VD2S_UNARY_FUNCTION_REGISTER` – linha 12 Esta é a função que "ensina" o mecanismo de expressões templates a aplicar a operação em arrays e vetores.

`VD2S_UNARY_FUNCTION_IO` – linha 16 A última macro usada na definição de funções é a que especializa `UnaryReturn`. Para usá-la basta passarmos como argumentos para a macro o nome da classe, a dimensão do vetor e o tipo que desejamos como retorno de nossa função (**`double`**).

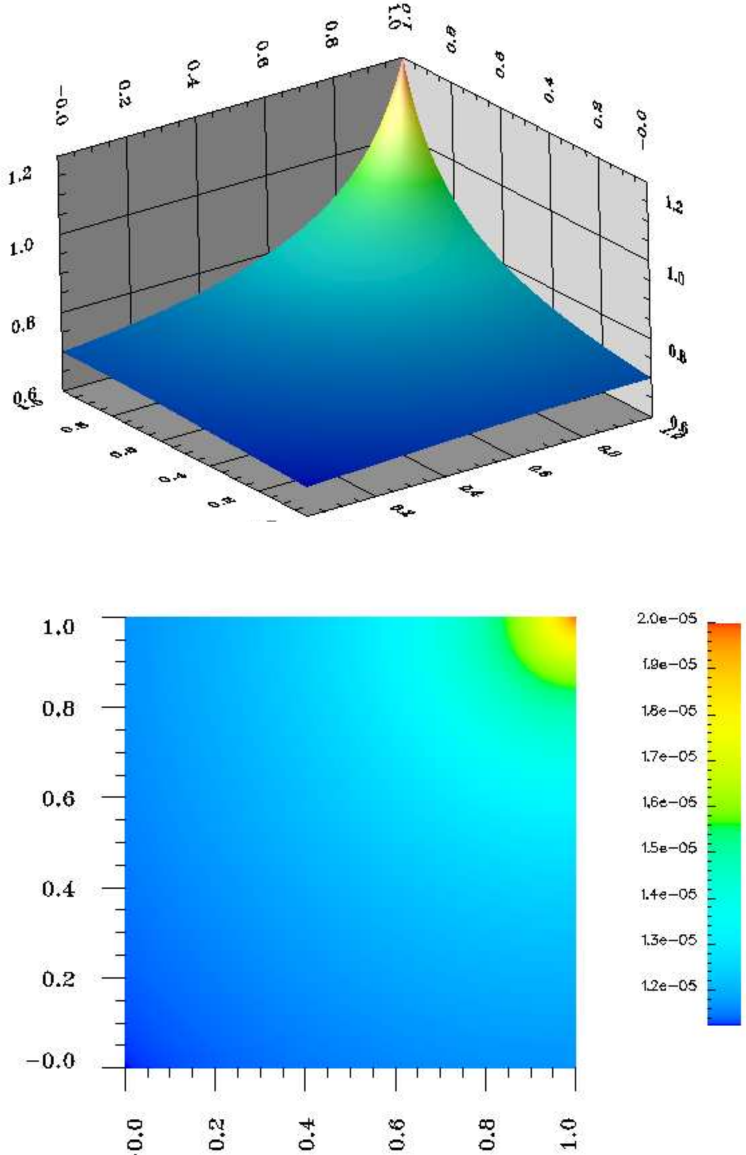


Figura 7.1: Gráfico de (7.17).

Representação gráfica da operação $\alpha\big(T(\boldsymbol{\xi})\big)$

Dada a semelhança entre (7.8) e (7.16), o diagrama a seguir mostra somente a representação gráfica de $\alpha(T(\boldsymbol{\xi}))$; os outros termos são idênticos.

$$\alpha\big(T(\boldsymbol{\xi})\big) = \alpha\left(T\left(\begin{bmatrix} \langle \bullet \rangle_2 & \cdots \\ \vdots & \end{bmatrix}\right)\right) = \alpha\left(\begin{bmatrix} \langle \bullet \rangle_2 & \cdots \\ \vdots & \end{bmatrix}\right) = \begin{bmatrix} \bullet & \cdots \\ \vdots & \end{bmatrix}.$$

Naturalmente este operador também precisa produzir as combinações $\langle \nabla\hat{\varphi}_j \| \nabla\hat{\varphi}_i \rangle_e$ para $i, j = 1, \ldots, n_b$, e, portanto, também passará pela operação de combinação `combineDot()`.

```
1   class RoundCorner {
2     public:
3       // Esta macro descreve o comportamento da função para um único vetor.
4       VD2S_UNARY_FUNCTION_ATOM( RoundCorner, 2 )
5       {
6         return 1e-5 *
7           ( 1.0 + 1.0 / ( 1.0 +
8               10.0 * norm( v - Vector< 2 >( 1.0, 1.0 ) ) ) );
9       }
10
11      // Esta macro ‘‘ensina’’ as expressões templates a trabalhar com arrays.
12      VD2S_UNARY_FUNCTION_REGISTER( RoundCorner, 2 );
13  };
14
15  // Aqui especializamos UnaryReturn.
16  VD2S_UNARY_FUNCTION_IO( RoundCorner, 2, double );
```

Listagem 7.2: Implementação de (7.17).

Implementação

A implementação do operador de difusão espacialmente variável (listagem 7.3) segue exatamente o mesmo padrão da implementação do operador com difusão constante (listagem 7.1).

As diferenças em relação a 7.1 resumem-se a:

- O dado `C` da classe (a partir da linha 34) agora não é mais um número real, e sim um template (`coef_t`) que fará as vezes da função $\alpha(\boldsymbol{x})$. Esta mudança também se reflete no constructor (linhas 4 a 8).
- O operador parêntesis (linhas 10 a 19) tem agora o termo referente a $\alpha(T(\boldsymbol{\xi}))$.

7.3 Operadores do tipo σu

Outra série de operadores muito importante na simulação de fenômenos ecológicos e ambientais é composta pelos operadores da forma (7.18), em que σ representa uma retirada de material do meio (seja ele um agente poluente ou uma parcela de uma população).

$$\mathcal{D}(u) = \sigma u. \tag{7.18}$$

Por trás da escolha deste operador está a hipótese de que este *decaimento é proporcional à quantidade de matéria existente*, ou em termos matemáticos,

$$\frac{du}{dt} = -k\,u.$$

Esta EDO tem como solução $u(t) = u_0\, e^{-kt}$, o que denota uma queda exponencialmente proporcional à u. Em termos de dinâmica populacional, este decaimento é uma mortalidade verhulstiana [Sos03];

```
template< class basis_t, class mapping_t, class coef_t >
class SpatialDivGrad_op {
  public:
    inline SpatialDivGrad_op( const basis_t& b, const mapping_t& m, const coef_t& c ) {
      B = &const_cast< basis_t& >( b );
      M = &const_cast< mapping_t& >( m );
      C = &const_cast< coef_t& >( c );
    }

    template< int D, class T, class E > inline typename
    UnaryReturn< Vector< D, T, E >, SpatialDivGrad_op< basis_t, mapping_t, coef_t > >::Type_t
    operator()( const Vector< D, T, E >& a ) const {
      typedef typename UnaryReturn< Vector< D, T, E >,
        SpatialDivGrad_op< basis_t, mapping_t, coef_t > >::Type_t Type_t;

      return (*C)( M->directMap().eval( a ) ) *
             combineDot( dot( transpose( M->inverseMap().j( a ) ),
             B->grad( a ) ) ) * fabs( M->directMap().jD( a ) );
    }

    template< int D1, int D2, class T, class E, class EA > inline typename
    MakeReturn< UnaryNode< SpatialDivGrad_op< basis_t, mapping_t, coef_t >,
      typename CreateLeaf< Array< D2, Vector< D1, T, E >, EA > >::Leaf_t >
    >::Expression_t
    eval( const Array< D2, Vector< D1, T, E >, EA >& a ) const {
      typedef UnaryNode< SpatialDivGrad_op< basis_t, mapping_t, coef_t >,
        typename CreateLeaf< Array< D2, Vector< D1, T, E >, EA > >::Leaf_t > Tree_t;

      return MakeReturn< Tree_t >::make(
        Tree_t( CreateLeaf< Array< D2, Vector< D1, T, E >, EA > >::make( a ) ) );
    }

  private:
    static basis_t* B;
    static mapping_t* M;
    static coef_t*  C;
};

// Não esqueça de inicializar os membros estáticos B, M e C...

struct UnaryReturn< Vector< D, T, E >, SpatialDivGrad_op< basis_t, mapping_t, coef_t > >
{
  typedef Tensor< basis_t::ndof, T, Symmetric > Type_t;
};
```

Listagem 7.3: Operador local de difusão com $\alpha = \alpha(\boldsymbol{x})$.

em termos de Ecotoxicologia Matemática corresponde ao decaimento típico do petróleo durante as primeiras horas de um derramamento [Can98].

Nas seções anteriores mantivemos uma preocupação constante com a clareza na construção dos operadores usados em OSIRIS, dentro do contexto de expressões templates da POOMA. Os operadores desta série são bastante parecidos aos difusivos em relação à implementação; assim, os detalhes que forem omitidos são em tudo similares aos já mencionados.

7.3.1 Decaimento constante

Tomando-se $\sigma \in \mathbb{R}_+^*$, e constante em (7.18), obtemos o operador de decaimento constante (7.19).

$$\mathcal{D}(u) = \sigma\, u. \tag{7.19}$$

Fazendo-se a formulação fraca para (7.18) (como por exemplo em [Can98]), chegamos à forma bilinear (7.20).

$$a(u,v) = \int_\Omega \sigma\, u\, v\, d\boldsymbol{x}, \tag{7.20}$$

novamente com v uma função teste apropriada. Este operador difere do operador difusivo por não conter derivadas.

A forma discreta de (7.20) é (7.21). Note-se que σ, sendo constante, já foi isolado.

$$a_h(u_h, v_h) = \sigma \sum_j c_j \int_{\Omega_h} \varphi_j \varphi_i\, d\boldsymbol{x} = \sigma \sum_j c_j\, \langle \varphi_j, \varphi_i \rangle_\Omega. \tag{7.21}$$

Na forma local, elemento a elemento, temos (7.22):

$$\langle \varphi_j, \varphi_i \rangle_e = \int_e \varphi_j \varphi_i\, d\boldsymbol{x} = \int_{\hat{e}} \hat{\varphi}_j \hat{\varphi}_i\, |\boldsymbol{J}(\boldsymbol{\xi})|\, d\boldsymbol{\xi}. \tag{7.22}$$

Representação gráfica

A representação gráfica da implementação de (7.22) é bastante simples e pode ser conferida em (7.23). Vale lembrar que estamos focados no caso triangular bidimensional, com base de Lagrange de

primeira ordem, embora OSIRIS trate também de outras dimensões, geometrias e aproximações.

$$\int_{\hat{e}} \hat{\varphi}_j \hat{\varphi}_i \, |\boldsymbol{J}(\boldsymbol{\xi})| \, d\boldsymbol{\xi} = \texttt{sum}\left(\underbrace{\texttt{combineDot}\left(\underbrace{\left\langle \begin{bmatrix} \bullet & \cdots \\ \vdots & \end{bmatrix} \right\rangle_3}_{\hat{\varphi}_1, \hat{\varphi}_2, \hat{\varphi}_3} \right)}_{\hat{\varphi}_j \hat{\varphi}_i,\ i,j=1,2,3} \times \underbrace{\left| \boldsymbol{J}\left(\begin{bmatrix} \langle \bullet \rangle_2 & \cdots \\ \vdots & \end{bmatrix} \right) \right|}_{|\boldsymbol{J}(\boldsymbol{\xi})|} \times \underbrace{\begin{bmatrix} \bullet & \cdots \\ \vdots & \end{bmatrix}}_{\texttt{w()}} \right) =$$

$$= \texttt{sum}\left(\left[\!\left[\begin{bmatrix} \bullet & \cdots \\ \vdots & \end{bmatrix} \right]\!\right]_3 \times \begin{bmatrix} \bullet & \cdots \\ \vdots & \end{bmatrix} \times \begin{bmatrix} \bullet & \cdots \\ \vdots & \end{bmatrix} \right) =$$

$$= \texttt{sum}\left(\left[\!\left[\begin{bmatrix} \bullet & \cdots \\ \vdots & \end{bmatrix} \times \begin{bmatrix} \bullet & \cdots \\ \vdots & \end{bmatrix} \times \begin{bmatrix} \bullet & \cdots \\ \vdots & \end{bmatrix} \right]\!\right]_3 \right)$$

$$= \texttt{sum}\left(\left[\!\left[\begin{bmatrix} \bullet & \cdots \\ \vdots & \end{bmatrix} \right]\!\right]_3 \right) = [\![\bullet]\!]_3 \tag{7.23}$$

Implementação

A implementação é completamente similar à apresentada em 7.1. Assim, a listagem 7.4 traz somente a implementação do operador **operator()**, que é a porção de código que mais muda de operador a operador.

```
1  template< int D, class T, class E > inline typename
2  UnaryReturn< Vector< D, T, E >, CteIdentity_op< basis_t, mapping_t > >::Type_t
3  operator()( const Vector< D, T, E >& a ) const {
4     typedef typename UnaryReturn< Vector< D, T, E >,
5        CteIdentity_op< basis_t, mapping_t > >::Type_t Type_t;
6
7     return C * combineDot( B->eval( a ) ) * fabs( M->directMap().jD( a ) );
8  }
```

Listagem 7.4: Operador local de decaimento, com σ constante.

Podemos notar que a implementação é bastante direta face aos outros operadores já apresentados.

A última operação é a multiplicação por σ, que poderia ser feita diretamente pelo usuário, já que a POOMA suporta o produto entre **double** e `Tensor`. Preferimos deixar a multiplição também por conta da classe.

7.3.2 Decaimento espacial

Em alguns casos, a hipótese de decaimento constante pode não ser a melhor para a situação que está sendo modelada. Um caso típico é o de modelos de dinâmica populacional espacial com dinâmica vital [Sos03]. Nestes modelos de várias espécies a mortalidade da presa *varia de acordo com a presença do*

predador. Neste caso podemos modelar o decaimento como uma função da quantidade de predadores em cada ponto $\boldsymbol{x}$ do domínio. Outra situação de interesse consiste na interação presa/poluente [Sos03]. A mortalidade da presa varia de acordo com a presença de poluente, de forma bastante parecida com a descrita acima para o caso de presas e predadores.

Isto posto, estamos interessados em implementar um operador em que $\sigma : \Omega \to \mathbb{R}_+^*$, obtendo (7.24), em tudo similar a (7.13).

$$\mathcal{D}(u) = \sigma(\boldsymbol{x})\, u. \tag{7.24}$$

A formulação variacional para (7.24) é análoga à de (7.20), de onde obtemos (7.25).

$$a(u, v) = \int_\Omega \sigma(\boldsymbol{x})\, u\, v\, d\boldsymbol{x}. \tag{7.25}$$

De forma discreta temos:

$$a_h(u_h, v_h) = \sum_j c_j \int_{\Omega_h} \sigma(\boldsymbol{x})\, \varphi_j\, \varphi_i\, d\boldsymbol{x} = \sum_j c_j\, \langle \sigma(\boldsymbol{x})\, \varphi_j, \varphi_i \rangle_{\Omega_h}. \tag{7.26}$$

Dado o caráter local das operações, mais uma vez vamos explorar a implementação do operador $\langle \sigma(\boldsymbol{x})\, \varphi_j, \varphi_i \rangle_e$. Como este operador não envolve derivadas, sua expressão em termos do elemento de referência fica como em (7.27).

$$\begin{aligned} \langle \sigma(\boldsymbol{x})\, \varphi_j, \varphi_i \rangle_e &= \int_e \sigma(\boldsymbol{x})\, \varphi_j\, \varphi_i\, d\boldsymbol{x} \\ &= \int_{\hat{e}} \sigma\big(T(\boldsymbol{\xi})\big)\, \hat{\varphi}_j\, \hat{\varphi}_i\, |\boldsymbol{J}(\boldsymbol{\xi})|\, d\boldsymbol{\xi}. \end{aligned} \tag{7.27}$$

Detalhes do operador

O operador $\mathcal{D}(u) = \sigma(\boldsymbol{x})\, u$ em sua forma fraca, discreta e local (7.27) não traz novidades em termos de implementação; o cálculo de $\sigma(\boldsymbol{x})$ será feito de forma totalmente análoga àquela usada para $\alpha(\boldsymbol{x})$.

7.4 Operadores do tipo $\nabla \cdot (\mathbf{V}u)$

Finalizando a série de operadores diferenciais[2] temos o *operador de advecção* (7.28)

$$\mathcal{D}(u) = \nabla \cdot (\mathbf{V}u), \tag{7.28}$$

onde $\mathbf{V} : \mathbb{R}^n \to \mathbb{R}^n$ é o campo vetorial responsável pelo movimento da variável u. Cabe aqui uma observação bastante importante. O operador (7.28) pode ser escrito como

$$\nabla \cdot (\mathbf{V}u) = u\, \nabla \cdot \mathbf{V} + \mathbf{V} \cdot \nabla u = \mathbf{V} \cdot \nabla u.$$

pois para efeitos de implementação consideramos que o campo $\mathbf{V}$ tem divergente nulo, ou seja, $\nabla \cdot \mathbf{V} = 0$. Assim, o operador que foi efetivamente implementado é dado por (7.29).

$$\mathcal{D}(u) = \mathbf{V} \cdot \nabla u. \tag{7.29}$$

[2] Naturalmente outros operadores podem ser implementados dentro do mesmo ambiente; devemos ressaltar que não é nossa intenção esgotar uma tabela de possíveis operadores diferenciais, muitos de grande interesse prático, mas sim exemplificar a criação daqueles com os quais trabalhamos há tempos.

7.4.1 Advecção com velocidade constante

Neste caso temos $\mathbf{V} \in \mathbb{R}^n$ constantce para todo $\mathbf{x} \in \Omega$, ou seja, temos a mesma velocidade em todo o domínio.

Procedendo com a formulação variacional de (7.29) obtemos a forma bilinear (7.30).

$$a(u,v) = \int_\Omega (\mathbf{V} \cdot \nabla u)\, v \, d\boldsymbol{x}. \tag{7.30}$$

A forma discreta de (7.30) é (7.31):

$$a_h(u_h, v_h) = \sum_j c_j \int_{\Omega_h} (\mathbf{V} \cdot \nabla \varphi_j)\, \varphi_i \, d\boldsymbol{x} = \sum_j c_j \langle \mathbf{V} \cdot \nabla \varphi_j, \varphi_i \rangle_{\Omega_h}. \tag{7.31}$$

Elemento a elemento, temos:

$$\langle \mathbf{V} \cdot \nabla \varphi_j, \varphi_i \rangle_e = \int_e \mathbf{V} \cdot \left\{ [\mathbf{J}^{-1}(\boldsymbol{\xi})]^T \nabla \hat{\varphi}_j \right\} \, \hat{\varphi}_i \, |\mathbf{J}(\boldsymbol{\xi})| \, d\boldsymbol{\xi}. \tag{7.32}$$

Observamos que os termos presentes em (7.32) já foram usados anteriormente e encontram-se na seção 7.2.1. A implementação é direta e bastante similar às já apresentadas, podendo ser conferida de forma resumida na listagem 7.5, a exemplo do que foi feito na listagem 7.4.

```
1  template< int D, class T, class E > inline typename
2  UnaryReturn< Vector< D, T, E >, CteAdvective_op< basis_t, mapping_t > >::Type_t
3  operator()( const Vector< D, T, E >& a ) const {
4     typedef typename UnaryReturn< Vector< D, T, E >,
5        CteAdvective_op< basis_t, mapping_t > >::Type_t Type_t;
6
7     return outerProduct( dot( V,
8              dot( transpose( M->inverseMap().j( a ) ),
9                 B->grad( a ) ) ), B->eval( a ) ) *
10          fabs( M->directMap().jD( a ) );
11 }
```

Listagem 7.5: Operador local de advecção, com **V** constante.

7.4.2 Advecção com variação espacial do campo de velocidade

O tratamento da variante do operador de advecção com variação espacial $\mathbf{V} : \Omega \subset \mathbb{R}^n \to \mathbb{R}^n$ é em tudo similar aos já apresentados para difusão e decaimentos espaciais. A única diferença reside no fato de que a função que implementa $\alpha(\boldsymbol{x})$ ou $\sigma(\boldsymbol{x})$ retorna um número real, enquanto que a que implementa $\mathbf{V}(\boldsymbol{x})$ retorna um vetor de dimensão igual à de Ω. Sua implementação está na listagem 7.6.

7.5 Operador de interpolação

Um dos operadores disponíveis em Osiris, e que não é do tipo diferencial, é o chamado *operador de interpolação*. Através dele, podemos usar aproximações numéricas, obtidas a partir de um modelo, em outro.

```
1  template< int D, class T, class E > inline typename
2  UnaryReturn< Vector< D, T, E >, SpatialAdvective_op< basis_t, mapping_t, coef_t > >::Type_t
3  operator()( const Vector< D, T, E >& a ) const {
4     typedef typename UnaryReturn< Vector< D, T, E >,
5        SpatialAdvective_op< basis_t, mapping_t, coef_t > >::Type_t Type_t;
6
7     return outerProduct( dot( (*V)( M->directMap().eval( a ) ),
8              dot( transpose( M->inverseMap().j( a ) ),
9                 B->grad( a ) ) ), B->eval( a ) ) *
10          fabs( M->directMap().jD( a ) );
11 }
```

Listagem 7.6: Operador local de advecção, com **V** espacialmente variável.

O operador de interpolação é simples: dada uma solução numérica u_h definida para uma malha, em outras palavras, um campo vetorial discreto, e um ponto $\boldsymbol{x} \in \Omega$ *não necessariamente coincidente com os vértices da malha* este operador avalia $u_h(\boldsymbol{x})$.

Devemos notar que graças à natureza local das bases de elementos finitos, esta interpolação também pode ser feita elemento a elemento. Assim, sendo e um elemento da malha e $\boldsymbol{x} \in e$ temos que $u_h(\boldsymbol{x})$ é dado por (7.33), onde $ndof$ é o número de graus de liberdade no elemento e c_k são os valores da aproximação nestes graus de liberdade.

$$u_h(\boldsymbol{x}) = \sum_{k=1}^{ndof} c_k \hat{\varphi}_k(\boldsymbol{x}). \tag{7.33}$$

Este operador está definido em duas classes diferentes:

- Em `FieldOrg_op` procedemos com o que denominamos *organização da informação* do campo. Neste passo, dado um número de elemento na malha e um campo, é construído um vetor com os valores do campo nos graus de liberdade deste elemento. Na listagem 7.7 podemos ver o núcleo desta operação que por já estar no padrão do mecanismo de expressões templates, serão omitidos detalhes não essenciais. Notamos que é um loop simples que busca os índices do elemento de `idx`, lê a posição correspondente no campo (`Field`) e armazena em `Ret` (usualmente um vetor do Pooma).

- Em `FieldInterp_op` efetuamos a interpolação propriamente dita (listagem 7.8). A operação é simples, pois consiste de uma avaliação da base seguida de um produto com o vetor resultante da operação anterior (`FieldOrg_op`).

Vale aqui um comentário repetido: nas listagens 7.1 até 7.8, temos passos de Osiris cuja originalidade maior reside na configuração algorítmica que reduz de modo surpreendente o tempo de execução, somando à vantagem de integração elemento a elemento dos elementos finitos, aquela das expressões templates aqui definidas.

```
1   inline typename UnaryReturn< uinteger_t, FieldOrg_op< Field_t > >::Type_t
2   operator()( uinteger_t idx ) const {
3      typedef typename UnaryReturn< uinteger_t, FieldOrg_op< Field_t > >::Type_t Type_t;
4
5      Type_t Ret;
6
7      for( uinteger_t ii = 0; ii < Type_t::d1; ii++ )
8      {
9         Ret( ii ) = Field->read(
10                    Field->mesh().elements()( idx ).theReference( ii ) );
11     }
12
13     return Ret;
14  }
15
16  template< class field_t >
17  struct UnaryReturn< uinteger_t, FieldOrg_op< field_t > >
18  {
19     typedef typename field_t::Mesh_t::Element_t Temp_t;
20
21     typedef Vector< Temp_t::ndof, real_t, Full > Type_t;
22  };
```

Listagem 7.7: Primeira parte do operador de interpolação.

```
1   template< int D, class T, class E > inline typename
2   BinaryReturn< uinteger_t, Vector< D, T, E >,
3                 FieldInterp_op< Field_t, Basis_t > >::Type_t
4   operator()( uinteger_t idx, const Vector< D, T, E >& v ) const
5   {
6      typedef typename
7      BinaryReturn< uinteger_t, Vector< D, T, E >, FieldInterp_op< Field_t, Basis_t >
8                  >::Type_t Type_t;
9
10     return dot( B->eval( v ), FieldOrg_op< Field_t >( *Field )( idx ) );
11  }
12
13  template< class field_t, class basis_t, int D, class T, class E >
14  struct BinaryReturn< uinteger_t, Vector< D, T, E >, FieldInterp_op< field_t, basis_t > >
15  {
16     typedef real_t Type_t;
17  };
```

Listagem 7.8: Segunda parte do operador de interpolação.

Parte IV

Galeria

CAPÍTULO 8

Diversos exemplos e aplicações

Neste capítulo exploraremos diversas aplicações e exemplos dos operadores definidos no capítulo 6, mostrando a robustez e a variedade de aplicações desta abordagem. Cada modelo será seguido de alguns cenários criados para ilustrar a flexibilidade dos configuradores já apresentados. Cada cenário muda um ou mais aspectos da aproximação numérica, como a geometria dos elementos utilizados, a ordem de aproximação (base) ou ainda a dimensão espacial.

Executamos todos os exemplos em três máquinas, configuradas de acordo com a tabela 8.1.

Máquinas				Sistema	
Nome	Processador	Memória	Disco	GNU/Linux	Compilador
`honey`	Athlon XP 1800+	256M DDR 266	40Gb ATA100	Gentoo 2004.0	GCC 3.3.2
`crucio`	Athlon XP 2600+	512M DDR 333	40Gb ATA100	Gentoo 2004.1	GCC 3.3.3
`gambrinus`	2x Athlon MP 2000+	1G DDR 333	80Gb ATA100	Gentoo 2004.0	GCC 3.3.2

Tabela 8.1: Máquinas usadas nas simulações.

8.1 A equação de Poisson

O primeiro modelo a ser implementado usando Osiris é o de Poisson, dado pelo sistema em (8.1). Este é o modelo escolhido por ser simples, conhecido e por modelar diversos fenômenos, como por exemplo difusão de calor em um meio ou a dispersão espacial de uma população [Sos03].

$$\begin{aligned} -\alpha\Delta u &= f, \quad \text{em } \Omega \subset \mathbb{R}^n \\ u &= g, \quad \text{em } \partial\Omega. \end{aligned} \tag{8.1}$$

Parâmetros	$n = 2$ $\alpha = 10^{-4}$ $f = f(x, y) = 2\alpha[x(1-x) + y(1-y)]$ $g = 0$ em $\partial\Omega$
Malha	quadrada $[0, 1]^2$ (figura 8.1) 516600 elementos triangulares 1996 elementos de fronteira 259299 vértices base de Lagrange, ordem 1 (3 nós por elemento)
Nós para quadratura	1 ponto para o operador 3 pontos para o termo independente

Tabela 8.2: Cenário 1.

8.1.1 Cenário 1

Para testar os procedimentos, neste cenário 1, escolhemos $f(x, y)$ de forma que a solução seja exatamente definida a priori por

$$u(x, y) = xy(1-x)(1-y).$$

Observamos que, com esta escolha, $u = 0$ em $\partial\Omega$, como exigido, com $g \equiv 0$.

O sistema linear resultante da discretização de 8.1 foi resolvido usando-se uma biblioteca especializada em métodos de Krylov para grandes sistemas chamada *PETSc*[1], versão 2.2.0. PETSc (Portable, Extensible Toolkit for Scientific Computation) é fortemente orientada ao paralelismo, embora a tenhamos usado de forma serial. Escolhemos o erro relativo entre iterações com precisão de 10^{-5} como critério de parada.

Para problemas simétricos como os de Poisson usamos o método dos Gradientes Conjugados, com pré-condicionamento de Jacobi. Para problemas gerais sem simetria, utilizamos Gradientes Bi-Conjugados, também com pré-condicionamento de Jacobi.

Vamos verificar passo a passo neste primeiro exemplo, todas as configurações que compõem a simulação. As listagens deste exemplo podem ser apreciadas no apêndice C.

Configurando o elemento

Na listagem 8.1 é feita a *configuração do elemento* utilizado na simulação. Notemos que foram feitas de fato três declarações. Na primeira declaramos a *classe geradora do elemento* (`FeGenerator_t`), na segunda declaramos o próprio *elemento* (`Element_t`) e finalmente recuperamos, a partir do próprio elemento, a dimensão espacial do problema.

[1] `http://www.mcs.anl.gov/petsc`

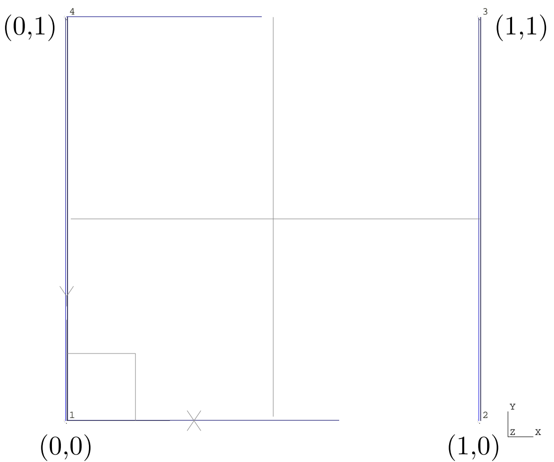


Figura 8.1: Geometria para os Cenários 1, 2 e 3.

```
// Declarando a classe que exporta o elemento.
typedef FE_GENERATOR<
  FE_Family< lagrange< 1 > >, // Lagrange de primeira ordem.
  FE_Geometry< triangle > // Geometria:  triângulo.
> FeGenerator_t;

// Declarando o elemento.
typedef FeGenerator_t::RET Element_t;

// A partir do elemento recuperamos a dimensão do problema.
const int D = Element_t::spatialDimensions;
```

Listagem 8.1: Declaração do elemento usado no Cenário 1.

Malha e campo

As próximas definições a serem tratadas são a *malha* e o *campo*. Deve-se lembrar que, na nomenclatura do OSIRIS, um campo ou *field* consiste de uma geometria — usualmente a malha — e de valores associados a esta geometria. Na listagem 8.2 podemos ver que ambas as definições são simples, não envolvendo configurações automáticas.

```
// A malha (geometria)
typedef UnstructuredMesh< D, Element_t > Mesh_t;

// O campo.
typedef OsirisField< Mesh_t, Array< 1 > > Field_t;
```

Listagem 8.2: Malha e campo para o Cenário 1.

A malha é um objeto do tipo `UnstructuredMesh` com dois parâmetros templates: um correspondente

à dimensão da malha e outro ao elemento que a compõe. Esta definição de malha é baseada na da Pooma [PT98], e foi criada por razões de compatibilidade com versões anteriores da biblioteca.

A declaração do campo também é simples, consistindo novamente de dois parâmetros templates. O primeiro é a geometria (malha) e o segundo é a estrutura que armazenará valores sobre a malha, no caso um array unidimensional simples.

Base e transformação padrão ↔ real (referência)

Com o elemento e a malha prontos (listagens 8.1 e 8.2), podemos configurar a base e a transformação padrão ↔ real. A listagem 8.3 mostra as duas configurações.

```
// A base de elementos finitos.
typedef BASIS_GENERATOR<
   BASIS_Fe< Element_t >, // O elemento onde será construída a base.
   BASIS_Type< grad_basis > // O tipo de base.
>::RET Basis_t;

// A transformação padrão ↔ real.
typedef L2G_MAP_GENERATOR<
   L2G_FeGenerator< FeGenerator_t >, // O gerador do elemento.
   L2G_Geometry< Mesh_t >, // A malha subjacente.
   L2G_MapKind< both_map > // O tipo de transformação.
>::RET Mapping_t;
```

Listagem 8.3: Declaração da base e da transformação padrão ↔ real para o Cenário 1.

É interessante notarmos que *apenas com o tipo do elemento* e *o tipo de base* obtemos um objeto que define a base de Lagrange de primeira ordem para o triângulo 8.2 (funções e gradientes). Da mesma forma, com alguma informação, obtemos um objeto plenamente funcional com as transformações direta e inversa como definidas na seção A.1.

$$
\begin{array}{rclcrcl}
\hat{\varphi}_1(x,y) & = & 1-x-y & \qquad & \nabla\hat{\varphi}_1(x,y) & = & [-1,-1] \\
\hat{\varphi}_2(x,y) & = & x & & \nabla\hat{\varphi}_2(x,y) & = & [1,0] \\
\hat{\varphi}_3(x,y) & = & y & & \nabla\hat{\varphi}_3(x,y) & = & [0,1]
\end{array} \tag{8.2}
$$

Outro ponto a que devemos chamar atenção é o uso de `FeConfigurator_t` na configuração da transformação padrão ↔ real. Este fato ilustra a cooperação entre configuradores menores em detrimento de um único configurador monolítico, como exposto em [CE99] e [CE00]. Em outras palavras, as informações de configuração, ou "DNA", como apelidado na seção 5.3.3, do elemento são usadas na criação do "DNA" da transformação padrão ↔ real.

O operador diferencial

A configuração do operador diferencial que corresponde ao da equação 8.1 pode ser conferida na listagem 8.4. Notemos que a especificação mais uma vez é feita em alto nível. Por exemplo, quando declaramos `LOCOP_Model< divgrad< D > >` estamos nos referindo a operadores do tipo $\nabla\cdot(\alpha\nabla)$, que

serão particularizados por `LOCOP_SubModel< cte_parameter >` dentro da hipótese de α constante. Também são especificados o campo sobre o qual o operador atuará e o integrador numérico.

```
// Operador laplaciano.
typedef LOCOP_GENERATOR<
   LOCOP_Class< galerkin >, // Método de Galerkin.
   LOCOP_Model< divgrad< D > >, // Operador tipo ∇·(α∇).
   LOCOP_SubModel< cte_parameter >, // α = cte.
   LOCOP_Field< Field_t >, // O campo.
   LOCOP_Integrator< Integrator_t > // O integrador numérico.
>::RET CLaplacian_t;
```

Listagem 8.4: Declaração do operador laplaciano para o Cenário 1.

Condição de contorno

A condição de contorno escolhida (Dirichlet nula em todo $\partial\Omega$) tem sua construção exemplificada na listagem 8.5. A declaração é bastante simples, pois devemos especificar somente o tipo e o campo onde a condição será aplicada.

```
// Dirichlet nulo.
typedef BC_GENERATOR<
   BC_Type< null_dirichlet< D > >,
   BC_Field< Field_t > >::RET DirichletBC_t;
```

Listagem 8.5: Condição de contorno de Dirichlet nula para o Cenário 1.

O lado direito

O lado direito do problema 8.1

$$f(x,y) = 2\alpha[x(1-x) + y(1-x)],$$

está implementado na listagem do apêndice C.3, usando o mesmo modelo apresentado na seção 7.2.2 através do mecanismo de expressões templates.

Resultados

A *solução aproximada* para o Cenário 1 pode ser apreciada na figura 8.2, tanto na forma de curvas de contorno como na forma de gráfico de densidade.

A diferença entre a solução esperada $u(x,y)$ avaliada nos pontos da malha e a aproximada $u_h(x,y)$ é $\|u(x,y) - u_h(x,y)\| = 5.88 \times 10^{-5}$.

A tabela 8.3 resume os tempos de execução para o Cenário 1.

Como podemos notar, a avaliação dos operadores gastou o menor tempo entre todas as operações do programa.

Tarefa	Tempo
Leitura e processamento da malha	28.15s
Avaliação dos operadores e construção de matrizes e vetores	4.64s
Resolução de sistemas lineares	3m22s
Exportação para visualização	10.62s
Total	**4m5s**

Tabela 8.3: Tempos de execução para o Cenário 1 (máquina `honey`).

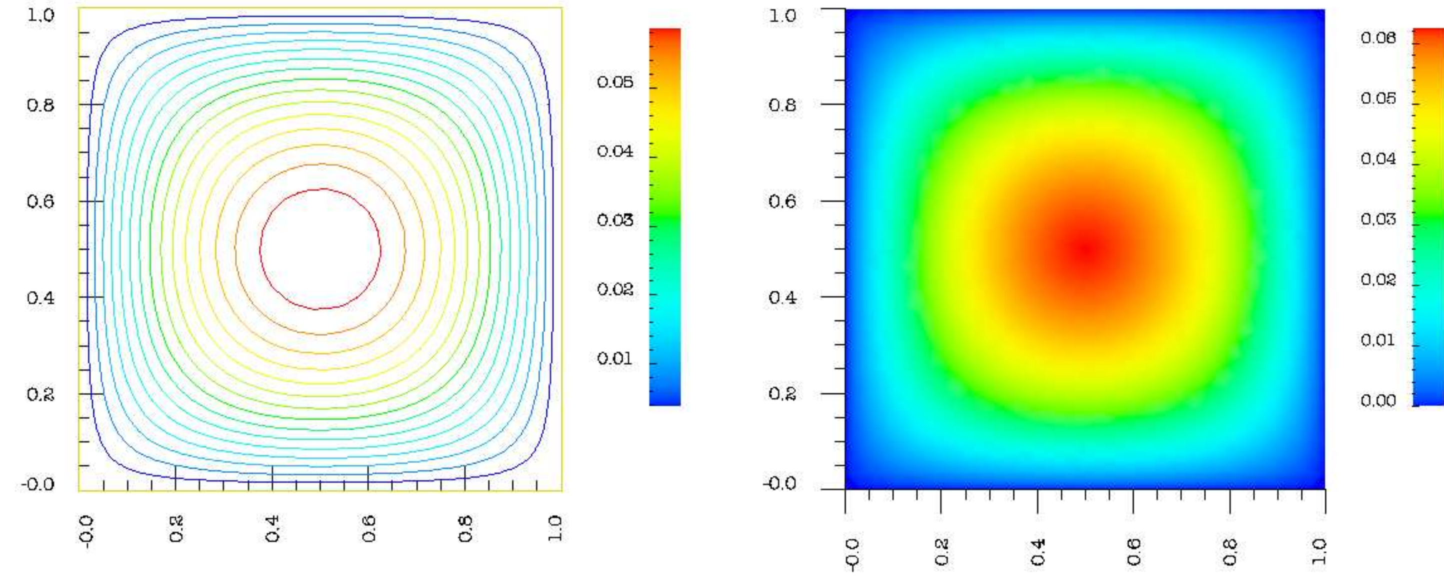


Figura 8.2: Curvas de nível e densidade para a solução aproximada dos Cenários 1, 2 e 3.

Por outro lado, a resolução do sistema linear com o PETSc deixou a desejar, talvez por ser um software paralelo sendo usado de modo serial ou por uma má escolha do método de pré-condicionamento (pré-condionador de Jacobi). Maiores investigações são necessárias para podermos determinar a real causa de tanto tempo para a resolução dos sistemas lineares.

8.1.2 Cenário 2

A tabela 8.4 resume os dados de simulação para o Cenário 2.

Parâmetros	$n = 2$
	$\alpha = 10^{-4}$
	$f = f(x, y) = 2\alpha[x(1-x) + y(1-x)]$
	$g = 0$ em $\partial\Omega$
Malha	quadrada $[0, 1]^2$ (figura 8.1)
	516600 elementos triangulares
	1996 elementos de fronteira
	1035197 vértices
	base de Lagrange, ordem 2 (6 nós por elemento)
Nós para quadratura	2 pontos para o operador
	3 pontos para o termo independente

Tabela 8.4: Cenário 2.

Configurando o elemento

A alteração introduzida no Cenário 2 consiste no aumento da ordem da base, passando de linear para quadrática.

```
// Declarando a classe que exporta o elemento.
typedef FE_GENERATOR<
   FE_Family< lagrange< 2 > >, // Lagrange de segunda ordem.
   FE_Geometry< triangle > // Geometria:  triângulos.
> FeGenerator_t;
```

Listagem 8.6: Declaração do elemento usado no Cenário 2.

A *única* mudança no código é na configuração do elemento (listagem 8.6). O resto do programa *permanece exatamente como antes*. A simples troca de `lagrange< 1 >` por `lagrange< 2 >` reconfigurou completamente operadores, transformações, base, malha e campo, *sem intervenção do usuário*.

Resultados

Felizmente, e como era de se esperar, a solução aproximada concorda graficamente com aquela do Cenário 1, e portanto não vamos reproduzir a figura.

A diferença entre a solução esperada $u(x, y)$ avaliada nos pontos da malha e a aproximada $u_h(x, y)$ é $\|u(x, y) - u_h(x, y)\| = 3.38 \times 10^{-5}$.

A tabela 8.5 resume os tempos de execução para o Cenário 2. Não conseguimos detectar as causas do péssimo desempenho do PETSc na solução deste sistema.

Tarefa	Tempo
Leitura e processamento da malha	28.68s
Avaliação dos operadores e construção de matrizes e vetores	12.7s
Resolução de sistemas lineares	60m22s
Exportação para visualização	27.85s
Total	**61m34s**

Tabela 8.5: Tempos de execução para o Cenário 2 (máquina `gambrinus`).

É interessante observarmos que o tempo de avaliação dos operadores foi aproximadamente três vezes maior que no Cenário 1. É natural esperarmos essa diferença se levarmos em conta que o número de funções de base dobrou, bem como sua complexidade, resultando em um número maior de operações de ponto flutuante. De acordo com o exposto em capítulo anteriores, observamos que as operações do Cenário 1 são feitas entre tensores com 6 componentes (3×3, porém simétrico). No Cenário 2 temos tensores com 21 componentes (6×6, também simétricos), numa relação de $3.5 = 21 : 6$, bastante próxima da relação de tempos entre os dois cenários[2].

8.1.3 Cenário 3

A tabela 8.6 resume os dados de simulação para o Cenário 3.

Parâmetros	$n = 2$ $\alpha = 10^{-4}$ $f = f(x, y) = 2\alpha[x(1 - x) + y(1 - x)]$ $g = 0$ em $\partial\Omega$
Malha	quadrada $[0, 1]^2$ (figura 8.1) 488601 elementos quadrados 2796 elementos de fronteira 490000 vértices base de Lagrange, ordem 2 incompleta (4 nós por elemento)
Nós para quadratura	3 pontos para o operador 4 pontos para o termo independente

Tabela 8.6: Cenário 3.

[2]De fato, a máquina `gambrinus` é pouca coisa mais rápida que `honey`

Configurando o elemento

O Cenário 3 é novamente uma variação dos Cenários 1 e 2. A única mudança consiste na troca da geometria do elemento, passando de triângulos para quadriláteros. Desta forma, a *única* mudança no código é na configuração do elemento (listagem 8.7)

```
// Declarando a classe que exporta o elemento.
typedef FE_GENERATOR<
  FE_Family< lagrange< 1 > >, // Lagrange de segunda ordem incompleta.
  FE_Geometry< rectangle > // Geometria:  quadriláteros.
> FeGenerator_t;
```

Listagem 8.7: Declaração do elemento usado no Cenário 3.

Mais uma vez o restante da listagem permanece igual.

A solução aproximada pode ser vista na figura 8.2. Novamente, e de acordo com a expectativa, ela coincide visualmente com aquela do Cenário 1, e portanto não vamos reproduzi-la aqui.

A diferença entre a solução esperada $u(x, y)$ avaliada nos pontos da malha e a aproximada $u_h(x, y)$ é $\|u(x, y) - u_h(x, y)\| = 5.38 \times 10^{-5}$.

A tabela 8.7 resume os tempos de execução para o Cenário 3.

Tarefa	Tempo
Leitura e processamento da malha	33.62s
Avaliação dos operadores e construção de matrizes e vetores	5.97s
Resolução de sistemas lineares	1m47s
Exportação para visualização	14.44s
Total	**2m42s**

Tabela 8.7: Tempos de execução para o Cenário 3 (máquina `honey`).

Observamos que o tempo de avaliação dos operadores foi aproximadamente 1.3 vezes maior que no Cenário 1. Como já mencionamos, as operações do Cenário 1 são feitas entre tensores com 6 componentes, enquanto no Cenário 3 temos tensores com 10 componentes (4×4, simétricos), numa relação de $1.67 = 10 : 6$, novamente próxima da relação de tempos entre os dois cenários.

8.1.4 Cenário 4

Configurando o elemento

O Cenário 4 traz como novidade a passagem de problemas bidimensionais para problemas tridimensionais, sempre ressaltando as mudanças mínimas no código fonte.

De fato, a mudança ocorre mais uma vez no elemento, simplesmente através da troca de geometria, como podemos verificar na listagem 8.8.

Mais uma vez o restante da listagem permanece igual.

Parâmetros	$n = 3$
	$\alpha = 10^{-4}$
	$f = f(x, y, z) = 2\alpha[yz(1-y)(1-z) + xz(1-x)(1-z) + xy(1-x)(1-y)]$
	$g = 0$ em $\partial\Omega$
Malha	cúbica $[0, 1]^3$ (figura 8.3)
	556522 elementos tetraédricos
	19776 elementos de fronteira
	92193 vértices
	base de Lagrange, ordem 1 (4 nós por elemento)
Nós para quadratura	2 pontos para o operador
	4 pontos para o termo independente

Tabela 8.8: Cenário 4.

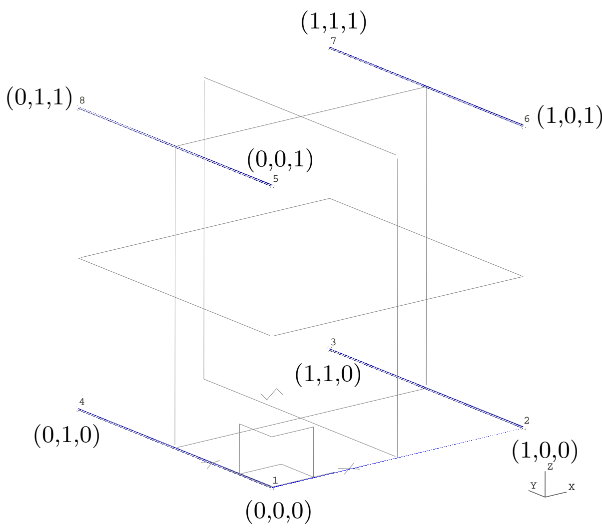


Figura 8.3: Geometria para o Cenário 4.

Resultados

Os resultados aproximados para o Cenário 4 constam da figura 8.4. A primeira parte da figura é um corte nas superfícies de nível da solução, enquanto que a segunda corresponde a três cortes planos perpendiculares a $[0, 0, 1]^T$ respectivamente nas alturas $z = 0.25$, $z = 0.5$ e $z = 0.75$.

A diferença entre a solução esperada $u(x, y, z) = xyz(1-x)(1-y)(1-z)$ avaliada nos pontos da malha e a aproximada $u_h(x, y, z)$ é $\|u(x, y, z) - u_h(x, y, z)\| = 1.43 \times 10^{-3}$.

A tabela 8.9 resume os tempos de execução para o Cenário 4.

Note que o tempo de avaliação dos operadores é similar ao observado no Cenário 2. Apesar dos tamanhos diferentes – 6 funções no Cenário 2 e 4 no Cenário 4 – e da maior simplicidade da base no tetraedro, as operações são feitas com mais pontos de integração, levando a uma compensação de tempo.

```
// Declarando a classe que exporta o elemento.
typedef FE_GENERATOR<
  FE_Family< lagrange< 1 > >, // Lagrange de primeira ordem.
  FE_Geometry< tetra > // Geometria:  tetraedros.
> FeGenerator_t;

// Declarando o elemento.
typedef FeGenerator_t::RET Element_t;

// A partir do elemento recuperamos a dimensão do problema.
const int D = Element_t::spatialDimensions;
```

Listagem 8.8: Declaração do elemento usado no Cenário 4.

Tarefa	Tempo
Leitura e processamento da malha	30.64s
Avaliação dos operadores e construção de matrizes e vetores	14.3s
Resolução de sistemas lineares	13.4s
Exportação para visualização	5.21s
Total	**1m5s**

Tabela 8.9: Tempos de execução para o Cenário 4 (máquina `honey`).

Apresentamos na tabela 8.10 um resumo dos Cenários de 1 a 4. Dela constam as dimensões dos problemas (número de elementos e de nós), tempo de construção das matrizes, que é onde o OSIRIS de fato mostra suas capacidades e tempo total de execução. Ressaltamos mais uma vez que o gargalo é a resolução de sistemas usando o PETSc.

	# Elementos	# Nós	Δt Ops	Δt Total
Cenário 1	516600	259299	4.64s	4m58s
Cenário 2	516600	1035197	12.7s	61m34s
Cenário 3	488601	490000	5.97s	2m42s
Cenário 4	556522	92193	13.3s	1m5s

Tabela 8.10: Resumo dos Cenários de 1 a 4.

Notemos que os tempos de montagem dos operadores devem depender basicamente do número de elementos e da complexidade da base. Como escolhemos problemas com uma discretização bastante similar neste quesito, os tempos de construção dos operadores devem recair na complexidade da avaliação da base e no seu tamanho, exatamente como comentamos na comparação entre os Cenários 1, 2 e 3. O caso do Cenário 4 é um pouco mais complexo, pois há também um aumento no número de pontos de integração graças a dimensão extra.

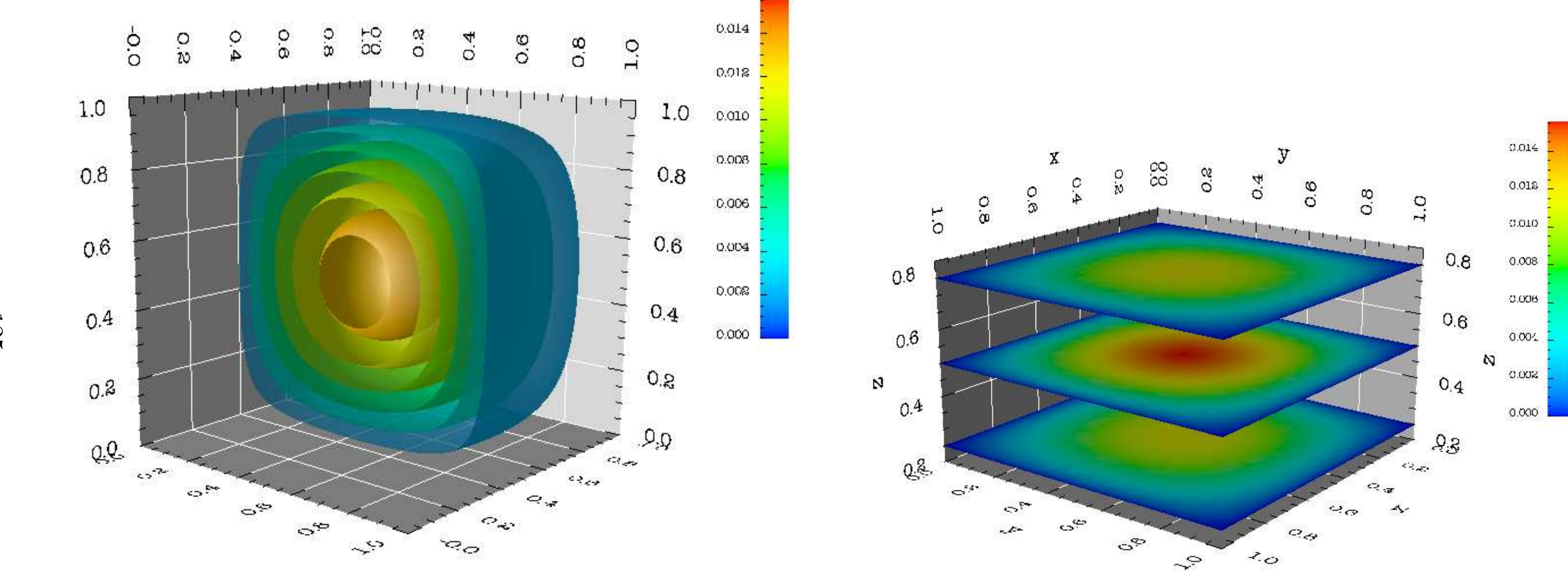


Figura 8.4: Corte nas superfícies de nível e densidade para a solução aproximada do Cenário 4.

8.2 Difusão espacial

Uma evolução natural da equação de Poisson (seção 8.1) é a consideração de termos de difusão espacialmente variáveis [dAP02], na forma (8.3).

$$\alpha : \Omega \subset \mathbb{R}^n \longrightarrow \mathbb{R}. \tag{8.3}$$

Desta forma o modelo básico de Poisson (8.1) evolui para (8.4).

$$\begin{aligned} \nabla \cdot (-\alpha(\boldsymbol{x})u) &= f, \quad \text{em } \Omega \subset \mathbb{R}^n \\ u &= g, \quad \text{em } \partial\Omega. \end{aligned} \tag{8.4}$$

8.2.1 Cenário 1

Parâmetros	$n = 2$
	$\alpha = \alpha(\boldsymbol{x}) = \begin{cases} 2 \times 10^{-5} & \text{se } \boldsymbol{x} \in Q_1 \\ 10^{-4} & \text{c.c.} \end{cases}$
	$f = f(x, y) = 2\alpha[x(1-x) + y(1-x)]$
	$g = 0$ em $\partial\Omega$
Malha	quadrada $[0, 1]^2$ (figura 8.1)
	516600 elementos triangulares
	1996 elementos de fronteira
	259299 vértices
	base de Lagrange, ordem 1 (3 nós por elemento)
Nós para quadratura	2 pontos para o operador
	3 pontos para o termo independente

Tabela 8.11: Cenário 1.

Seja Q_1 o quadrilátero de coordenadas $(0.5, 0.5)$, $(1.0, 0.5)$, $(1.0, 1.0)$ e $(0.5, 1.0)$.

Do ponto de vista puramente geométrico, o Cenário 1 da tabela 8.11 é bastante similar ao da tabela 8.2 (seção 8.1). Já pelo lado dos operadores, notamos que agora o coeficiente de difusão α é dependente de $\boldsymbol{x}$ (coordenada espacial).

A implementação do coeficiente em si segue as linhas gerais daquela apresentada em 7.2.2. As configurações do elemento, da malha, do campo, da condição de contorno e do lado direito permanecem as mesmas; a única mudança sensível é na construção do operador diferencial.

O operador diferencial

A listagem 8.9 mostra a declaração para esta nova versão do operador. Notemos que a única mudança é a troca de `LOCOP_SubModel< cte_parameter >` para `LOCOP_SubModel< spatial_parameter >`.

Obviamente a chamada de avaliação do operador elemento a elemento neste caso deve incluir[3] o

[3] Uma possível melhoria futura seria a inclusão desta operação como um parâmetro template, como é feito no próximo cenário no caso da condição de contorno de Dirichlet, listagem 8.11.

```
// Operador Laplaciano.
typedef LOCOP_GENERATOR<
  LOCOP_Class< galerkin >, // Método de Galerkin.
  LOCOP_Model< divgrad< D > >, // Operador tipo ∇·(α∇).
  LOCOP_SubModel< spatial_parameter >, // α = cte.
  LOCOP_Field< Field_t >, // O campo.
  LOCOP_Integrator< Integrator_t > // O integrador numérico.
>::RET CLaplacian_t;
```

Listagem 8.9: Declaração do operador difusão para o Cenário 1.

operador que calcula α, como na listagem 8.10.

```
// Função de difusão (‘‘corner’’ = canto).
class Corner {
  public:
    inline Corner() {}
    inline ~Corner() {}

    // Comportamento da função em um único elemento do vetor.
    template< class T, class E > inline typename
    UnaryReturn< Vector< 2, T, E >, Corner >::Type_t
    operator()( const Vector< 2, T, E >& v ) const {
      if( ( fabs( v( 0 ) - 0.75 ) <= 0.25 ) && ( fabs( v( 1 ) - 0.75 ) <= 0.25 ) ) {
        return 2e-5;
      }

      return 1e-4;
    }

    template< int D1, class E1, class T2, class E2 > inline typename
    MakeReturn< UnaryNode< Corner,
    typename CreateLeaf< Array< D1, Vector< D, T2, E2 >, E1 > >::Leaf_t > >::Expression_t
    operator()( const Array< D1, Vector< D, T2, E2 >, E1 >& l ) const
    {
       typedef UnaryNode< Corner,
         typename CreateLeaf< Array< D1, Vector< D, T2, E2 >, E1 > >::Leaf_t > Tree_t;

       return MakeReturn< Tree_t >::make(
         Tree_t( CreateLeaf< Array< D1, Vector< D, T2, E2 >, E1 > >::make( l ) ) );
    }
};

// Finalmente, associamos entrada e saída.
template< class T, class E > struct UnaryReturn< Vector< 2, T, E >, Corner > {
    typedef double Type_t;
};
```

Listagem 8.10: Função $\alpha(\boldsymbol{x})$ para o Cenário 1.

Resultados

A *solução aproximada* para o Cenário 1 pode ser apreciada na figura 8.5, tanto na forma de curvas de contorno como na forma de gráfico de densidade.

A tabela 8.12 resume os tempos de execução para o Cenário 1.

Tarefa	Tempo
Leitura e processamento da malha	26.72s
Avaliação dos operadores e construção de matrizes e vetores	4.91s
Resolução de sistemas lineares	4m38s
Exportação para visualização	10.86s
Total	**5m21s**

Tabela 8.12: Tempos de execução para o Cenário 1 (máquina `honey`).

Podemos verificar que a diferença entre a avaliação do operador com difusão constante (tabela 8.3) ou variável mudou pouco o tempo de execução.

Outro ponto a ser notado é a coerência da solução obtida com o tipo de coeficiente de difusão imposto. No quadrante Q_1 temos uma difusão menor do que no resto do domínio, resultando na clara distorção observada na figura 8.5.

8.2.2 Cenário 2

Parâmetros	$n = 2$
	$\alpha = \alpha(\boldsymbol{x}) = \begin{cases} 2 \times 10^{-5} & \text{se } \boldsymbol{x} \in Q_1 \\ 10^{-4} & \text{c.c.} \end{cases}$
	$f = f(x, y) = 2\alpha[x(1-x) + y(1-x)]$
	$g = \begin{cases} \frac{1}{2}y(1-y) & \text{em } \partial\Omega_1 \\ 0 & \text{em } \partial\Omega_2 \end{cases}$
Malha	quadrada $[0, 1]^2$ (figura 8.1)
	488601 elementos quadrados
	2796 elementos de fronteira
	490000 vértices
	base de Lagrange, ordem 2 incompleta (4 nós por elemento)
Nós de quadratura	2 pontos para o operador
	3 pontos para o termo independente

Tabela 8.13: Cenário 2.

Q_1 é o mesmo quadrilátero do Cenário 1, $\partial\Omega_1$ corresponde às fronteiras $x = 0$ e $x = 1$ e $\partial\Omega_2$ corresponde às fronteiras $y = 0$ e $y = 1$.

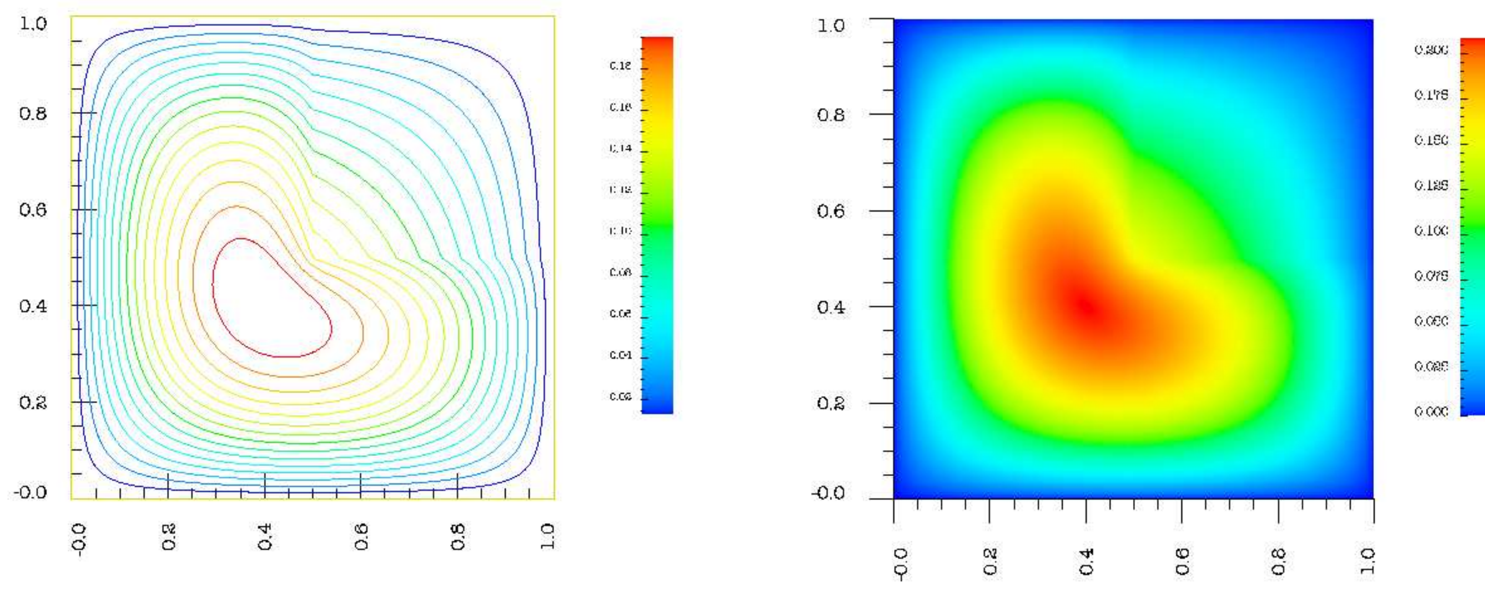


Figura 8.5: Curvas de nível e densidade para a solução aproximada do Cenário 1 com α espacialmente variável.

Condição de contorno

A condição de contorno escolhida (Dirichlet espacialmente variável) tem sua construção exemplificada na listagem 8.11. A declaração é similar à de Dirichlet nulo 8.5, com a inclusão do parâmetro `BC_DirichletEvaluator< DoubleArch >`. `DoubleArch` é a classe que implementa a avaliação da condição de contorno. Esta avaliação será chamada no momento conveniente dentro do mecanismo de expressões templates que calcula a condição de contorno.

```
// Dirichlet espacialmente variável.
typedef BC_GENERATOR<
  BC_DirichletEvaluator< DoubleArch >, // DoubleArch implementa a expressão ½y(1 − y).
  BC_Type< spatial_dirichlet< D > >,
  BC_Field< Field_t > >::RET SpDirichletBC_t;
```

Listagem 8.11: Condição de contorno de Dirichlet variável no espaço para o Cenário 2.

Resultados

A *solução aproximada* para o Cenário 2 pode ser apreciada na figura 8.6, tanto na forma de curvas de contorno como na forma de gráfico de densidade.

A tabela 8.14 resume os tempos de execução para o Cenário 2.

Tarefa	Tempo
Leitura e processamento da malha	34.5s
Avaliação dos operadores e construção de matrizes e vetores	5.23s
Resolução de sistemas lineares	3m18s
Exportação para visualização	16.83s
Total	**4m15s**

Tabela 8.14: Tempos de execução para o Cenário 2 (máquina `honey`).

Podemos notar o pequeno acréscimo no tempo de cálculo dos operadores devido à introdução da condição de Dirichlet espacialmente variável (como era esperado) e também pelo fato de que a base de ordem 2, incompleta para elementos quadrados, possui mais graus de liberdade do que a triangular.

Apresentamos também a figura 8.7, que mostra uma representação tridimensional da solução. Desta forma podemos observar que o contorno foi corretamente recuperado, resultando na parábola desejada $p(x) = \frac{1}{2}y(1-y)$.

8.2.3 Cenário 3

Para encerrar esta seção, apresentamos um último exemplo com difusão espacialmente variável em três dimensões.

A configuração dos componentes do problema (elemento, malha, base) é muito similar àquela apresentada no Cenário 4 da seção 8.1. É interessante notarmos que até a definição do operador diferencial,

Parâmetros	$n = 3$
	$\alpha = \alpha(\boldsymbol{x}) = \begin{cases} 2 \times 10^{-5} & \text{se } z < 0.5 \\ 10^{-4} & \text{c.c.} \end{cases}$
	$f = f(x, y, z) = 2\alpha[yz(1-y)(1-z) + xz(1-x)(1-z) + xy(1-x)(1-y)]$
	$g = 0$ em $\partial\Omega$
Malha	cúbica $[0, 1]^3$ (figura 8.3)
	556522 elementos tetraédricos
	19776 elementos de fronteira
	92193 vértices
	base de Lagrange, ordem 1 (4 nós por elemento)
Nós de quadratura	2 pontos para o operador
	4 pontos para o termo independente

Tabela 8.15: Cenário 3.

agora com α espacialmente variável, é idêntica àquela feita no Cenário 1 desta seção.

A função que calcula α será omitida por ser muito similar à apresentada na listagem 8.10.

Resultados

A *solução aproximada* para o Cenário 3 pode ser apreciada na figura 8.8 na forma de superfícies de nível.

A tabela 8.16 resume os tempos de execução para o Cenário 3.

Tarefa	Tempo
Leitura e processamento da malha	30.64s
Avaliação dos operadores e construção de matrizes e vetores	14.3s
Resolução de sistemas lineares	13m4s
Exportação para visualização	10.3s
Total	**1m5s**

Tabela 8.16: Tempos de execução para o Cenário 3 (máquina `honey`).

O resumo dos Cenários de 1 a 3 está na tabela 8.17.

	# Elementos	# Nós	Δt Ops	Δt Total
Cenário 1	516600	259299	4.91s	5m21s
Cenário 2	488601	490000	5.23s	4m15s
Cenário 3	556522	92193	13.4s	1m5s

Tabela 8.17: Resumo dos Cenários de 1 a 3.

É interessante observarmos novamente a influência do PETSc no tempo de execução. O tempo de aproximadamente 81 minutos para a avaliação dos operadores parece invalidar tudo o que foi dito

até agora sobre a eficiência da implementação com expressões templates. Porém, uma nova rodada da simulação, mas desta vez *apenas avaliando os operadores, sem introduzi-los na estrutura de matriz esparsa do PETSc*, obtemos um tempo de apenas 4.62s! Ou seja, praticamente todo o tempo de avaliação dos operadores é gasto alimentandop a matriz esparsa global do PETSc. Novamente não sabemos se esta é uma deficiência do pacote ou simplesmente um mau uso de suas funcionalidades.

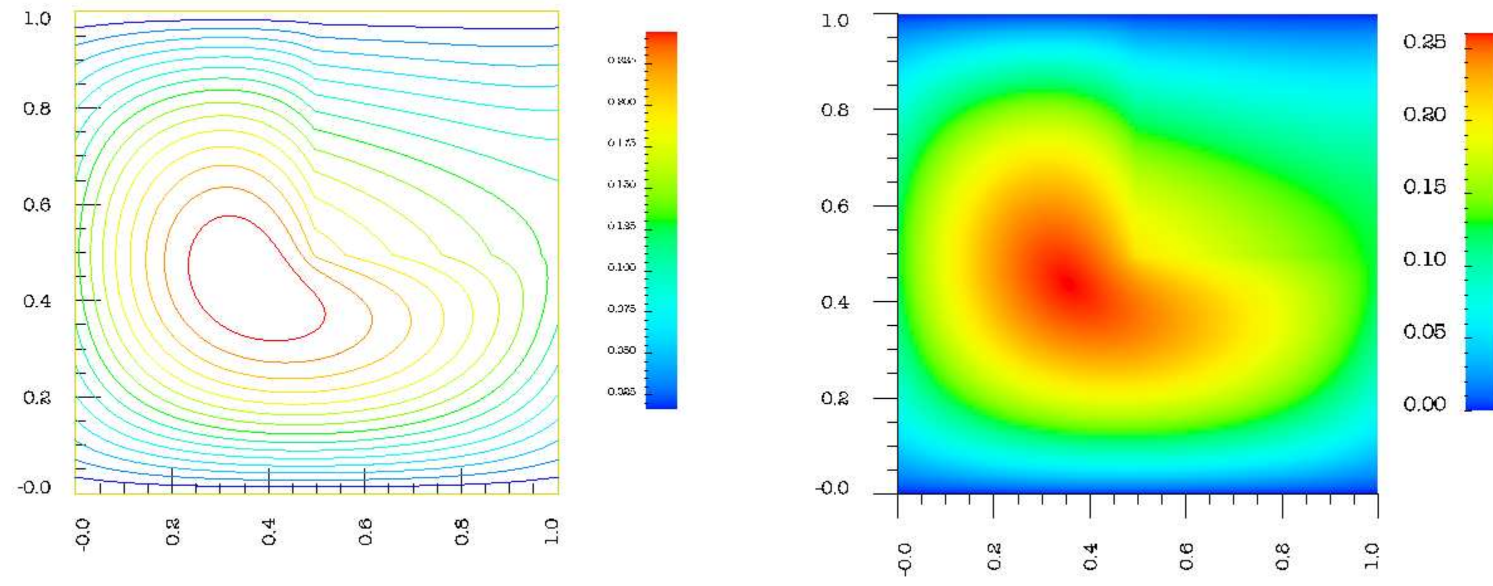


Figura 8.6: Curvas de nível e densidade para a solução aproximada do Cenário 2.

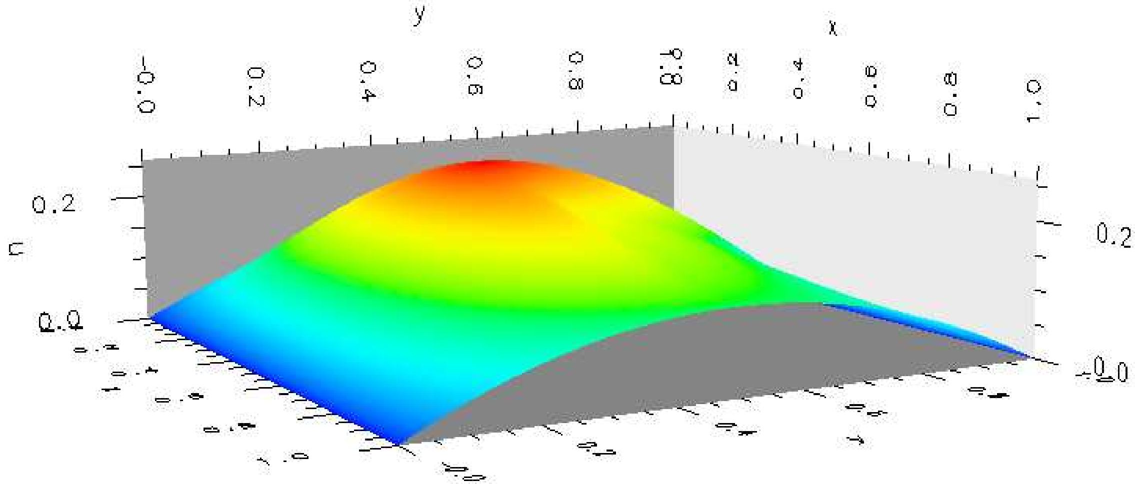


Figura 8.7: Representação 3D para a solução aproximada do Cenário 2.

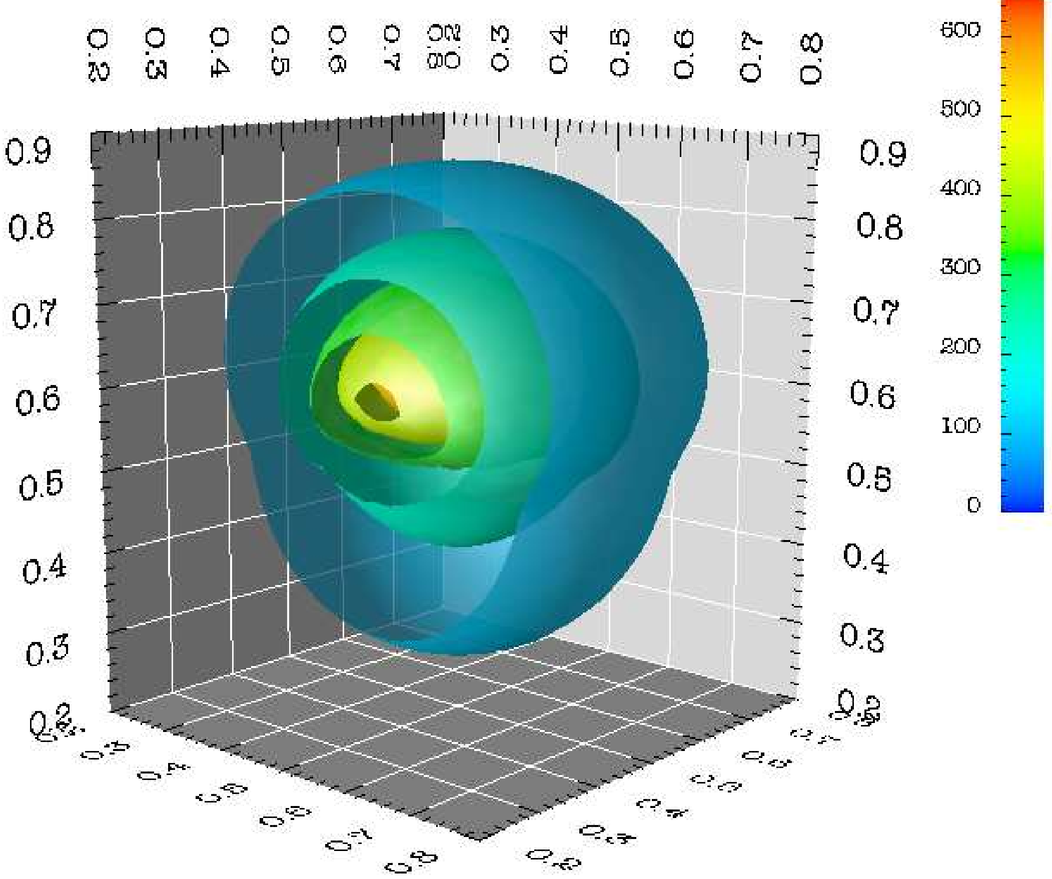


Figura 8.8: Superfícies de nível com corte para a solução aproximada do Cenário 3.

8.3 Interação poluente ↔ espécie animal

O próximo exemplo é uma aplicação de modelos de difusão-advecção a dois casos distintos, porém interativos:

- No primeiro modelo fizemos a simulação de um poluente que entra por um dos lados do domínio e se espalha através de um processo difusivo-advectivo, com velocidade constante.
- No segundo modelo simulamos a dispersão de uma população fictícia que vai de encontro ao poluente. A interação entre os dois casos se faz através do *coeficiente de mortalidade* σ da população, que será uma função espacial da concentração de poluente do item anterior.

Este modelo é uma *enorme simplificação* ilustrativa para o caso estudado em [Sos03] relativo à dispersão de materiais impactantes e seus efeitos na dinâmica de populações na Lagoa de Iberá.

É também interessante notarmos que este exemplo mostra a possibilidade de conexão entre dois modelos distintos através do operador de interpolação descrito na seção 7.5. A solução aproximada da equação do poluente — que é nodal — será usada através desta interpolação para o cálculo da mortalidade na equação da espécie.

8.3.1 O poluente

O sistema de equações de difusão-advecção usado para a modelagem da concentração u de poluente pode ser escrito como em (8.5). Ele considera uma variação espacial na difusão α_p, decaimento σ_p constante e campo de velocidades advectivo $\boldsymbol{V}$ constante. A condição de fronteira é Dirichlet nula em todo $\partial\Omega$.

$$\begin{aligned} \nabla\cdot(-\alpha_p(\boldsymbol{x})\nabla u) + \nabla\cdot(\boldsymbol{V}u) + \sigma_p u = f \qquad & \text{em } \Omega \\ u = 0 \qquad & \text{em } \partial\Omega \end{aligned} \tag{8.5}$$

A implementação dos operadores em (8.5) segue exatamente o padrão descrito no capítulo 7.

8.3.2 A população

O sistema de equações para a população c (8.6) é bastante similar à do poluente (8.5). As diferenças residem na difusão constante e na mortalidade (equivalente ao decaimento para o poluente), que agora depende de u.

$$\begin{aligned} -\alpha_e\Delta c + \nabla\cdot(\boldsymbol{U}c) + \sigma_e(u)c = g \qquad & \text{em } \Omega \\ c = 0 \qquad & \text{em } \partial\Omega, \end{aligned} \tag{8.6}$$

com $\sigma_e(u) = k\,u$.

Novamente, a implementação destes segue o padrão descrito no capítulo 7.

8.3.3 O domínio

Visando recuperar a experiência adquirida em outros ensaios (*cf* [Sos03], [Ber03], o domínio escolhido é retangular, de vértices $(0,0)$, $(5,0)$, $(5,1)$ e $(0,5)$ (conforme representado na figura 8.9).

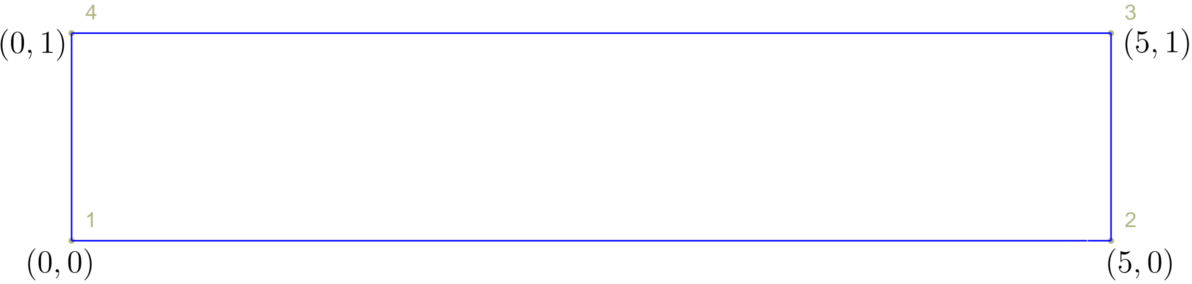


Figura 8.9: Domínio $[0,5] \times [0,1]$ para os cenários 1 e 2 (interação poluente-população).

8.3.4 Cenário 1

O primeiro cenário considera um ambiente completamente isolado, tanto para o poluente quanto para a população (o que equivale a uma condição de fronteira de Dirichlet nula). Há uma fonte poluente próxima ao canto direito do domínio, em $(x,y) = (3.75, 0.25)$ e uma fonte de indivíduos na esquerda. O poluente é transportado por advecção na direção da população e vice-versa. A intensidade de interação entre eles é dada pelo coeficiente k em $\sigma_e(u) = k\,u$ na equação 8.6.

A tabela 8.18 resume os parâmetros para o Cenário 1.

Resultados

Fizemos testes com várias intensidades de interação entre poluente e espécie. Classificamos estas intensidades em inexistente ($k = 0$), fraca ($k = 10^{-3}$), média ($k = 10^{-2}$), forte ($k = 10^{-1}$) e extrema ($k = 1$). As respectivas soluções aproximadas para a espécie (o poluente não se altera) podem ser apreciadas nas figuras 8.11 e 8.12.

A tabela 8.19 resume os tempos de execução para o Cenário 1 levando em conta a intensidade de interação. Podemos notar que os tempos são pequenos pois a malha é pouco refinada; a intenção deste exemplo é mostrar flexibilidade e não desempenho.

É clara a influência da mortalidade na população de acordo com o aumento da intensidade desta.

É interessante notarmos também que com o aumento da intensidade de interação o tempo de resolução do sistema para a espécies diminui (tarefa E). Este fato deve-se, muito provavelmente, ao aumento da dominância da diagonal e da simetria da matriz, resultados da predominância do operador $\sigma_e(u)\,c$.

Parâmetros (poluente)	$n = 2$ $\alpha_p = \alpha_p(x, y) = 10^{-5}(1 + y^2)$ $f = f(x,y) = \begin{cases} 10^{-3} & \text{se } \lvert x - 4.25\rvert \leq 0.5 \text{ e } \lvert y - 0.2\rvert \leq 0.1 \\ 0 & \text{caso contrário} \end{cases}$ $\boldsymbol{V} = (-8 \times 10^{-4}, 2 \times 10^{-4})^T$ $\sigma_p = 10^{-7}$ $u = 0$ em $\partial\Omega$
Parâmetros (espécie)	$n = 2$ $\alpha_e = 10^{-4}$ $g = g(x,y) = \begin{cases} 10^{-2} & \text{se } x \leq 0.5 \\ 0 & \text{caso contrário} \end{cases}$ $\boldsymbol{U} = (10^{-3}, 0)^T$ $\sigma_e = k\,u$ $c = 0$ em $\partial\Omega$
Malha	retangular $[0,5] \times [0,1]$ (figura 8.9) 25930 elementos triangulares 596 elementos de fronteira 52457 vértices base de Lagrange, ordem 2
Nós de quadratura (poluente)	4 pontos para o difusivo espacial 2 pontos para o operador advectivo 2 pontos para o decaimento 2 pontos para o termo independente
Nós de quadratura (espécie)	2 pontos para o difusivo constante 2 pontos para o operador advectivo 2 pontos para o decaimento espacial 2 pontos para o termo independente

Tabela 8.18: Cenário 1.

	Tempo				
Tarefa	$k = 0$	$k = 10^{-3}$	$k = 10^{-2}$	$k = 10^{-1}$	$k = 1$
Leitura e processamento da malha	1.05s	1.05s	1.05s	1.06s	1.04s
Construção de matrizes e vetores para o poluente	1.51s	1.52s	1.51s	1.51s	1.52s
Resolução do sistema linear do poluente	27.43s	27.51s	27.47s	27.61s	27.35s
Construção de matrizes e vetores para a espécie	1.38s	1.39s	1.39s	1.39s	1.39s
Resolução do sistema linear da espécie	29.07s	30.29s	22.30s	11.81s	7.28s
Total	1m2.85s	1m4.15s	56.23s	45.78s	40.98s

Tabela 8.19: Tempos de execução para o Cenário 1 (máquina `crucio`).

8.3.5 Cenário 2

No Cenário 2 alteramos um pouco as condições de fronteira, introduzindo a divisão $\partial\Omega = \partial\Omega_1 \cup \partial\Omega_2$, com $\partial\Omega_1 = \{(x,y) \in \mathbb{R}^2 : x = 0\}$ e Ω_2 como sendo o resto da fronteira. Com esta divisão da fronteira fizemos

- Poluente:

$$\frac{\partial u}{\partial y} = 0 \text{ em } \partial\Omega_1 \qquad u = 0 \text{ em } \Omega_2$$

- Espécie:

$$c = 10^{-2} \text{ em } \partial\Omega_1 \qquad c = 0 \text{ em } \Omega_2$$

A fonte populacional também é desligada.

Resultados

As soluções aproximadas para a espécie do Cenário 2 podem ser conferidas nas figuras 8.13 8.14 e 8.15, bem como os tempos de execução na tabela 8.20.

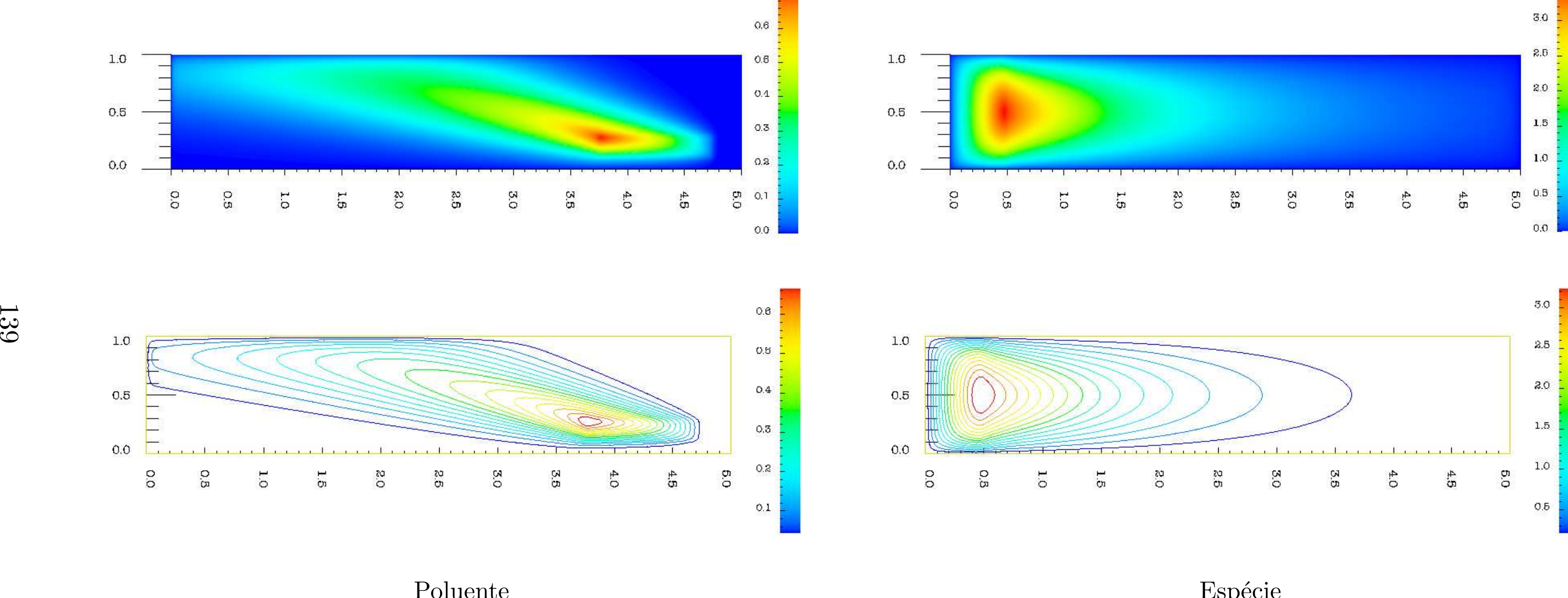
	Tempo				
Tarefa	$k = 0$	$k = 10^{-3}$	$k = 10^{-2}$	$k = 10^{-1}$	$k = 1$
A	1.25s	1.23s	1.22s	1.20s	1.18s
B	1.68s	1.90s	1.84s	1.91s	1.79s
C	34.12s	32.24s	39.13s	33.22s	33.73s
D	1.64s	1.49s	1.67s	1.37s	1.37s
E	26.61s	24.96s	18.61s	9.18s	6.06s
Total	1m8.06s	1m4.5s	1m5.82s	49.52s	46.76s

Tabela 8.20: Tempos de execução para o Cenário 2 (máquina `crucio`).

São seguidas as mesmas convenções da tabela 8.19.

Observações

Como seria de se esperar, à medida que cresce o valor de k, induzindo uma mortalidade maior na espécie afetada, os gráficos mostram essa população sobrevivendo em uma região cada vez mais restrita, onde o poluente não chega plenamente (note-se que está presente uma degradação desse poluente: $\sigma_p = 10^{-7}$).



Poluente Espécie

Figura 8.10: Curvas de nível e densidade para a solução aproximada do Cenário 1 com $k = 0$.

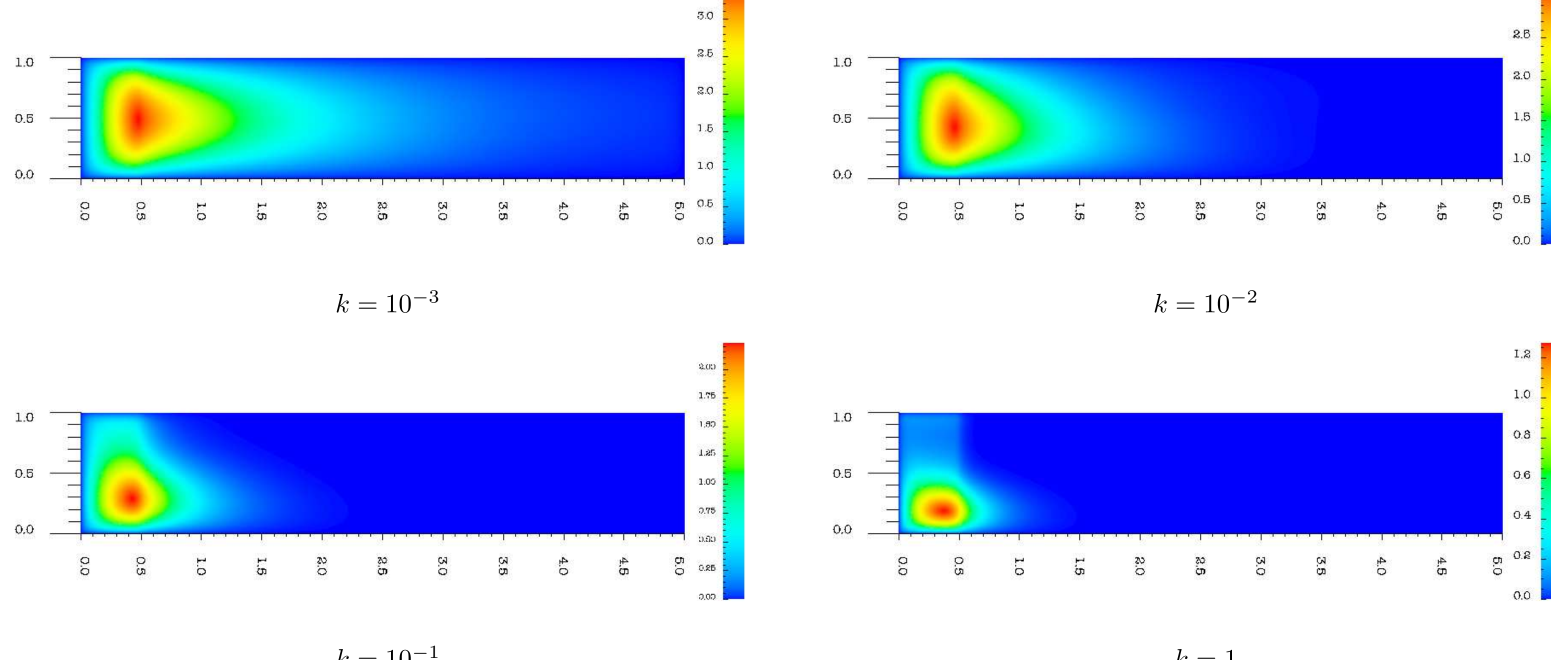
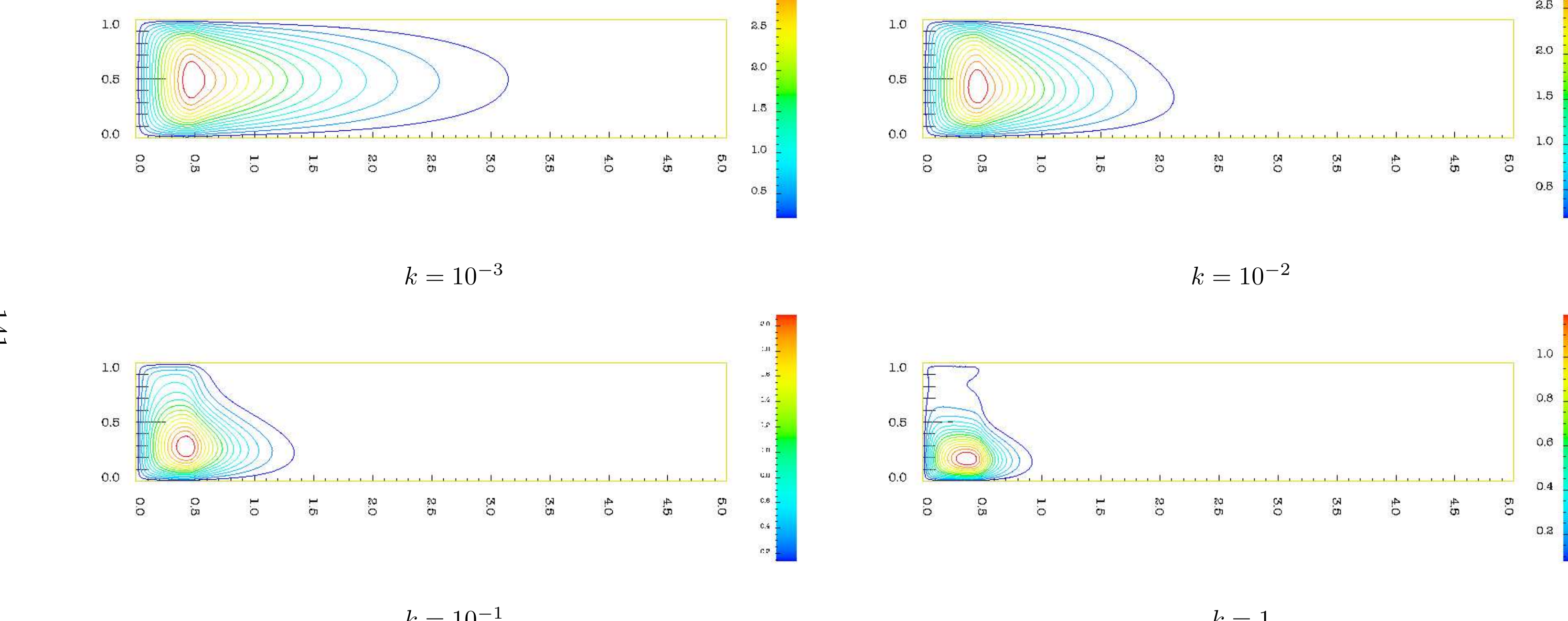


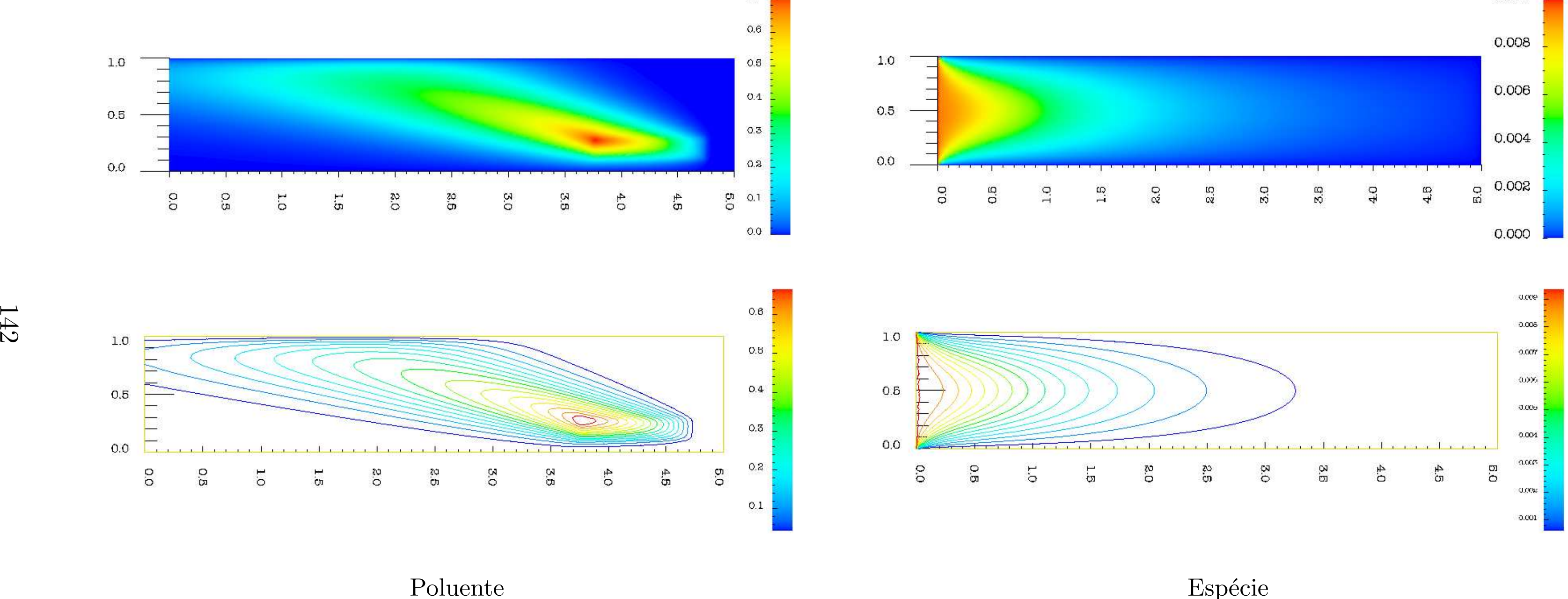
Figura 8.11: Gráfico de densidade para a solução aproximada da espécie do Cenário 1.



$k = 10^{-3}$ $k = 10^{-2}$

$k = 10^{-1}$ $k = 1$

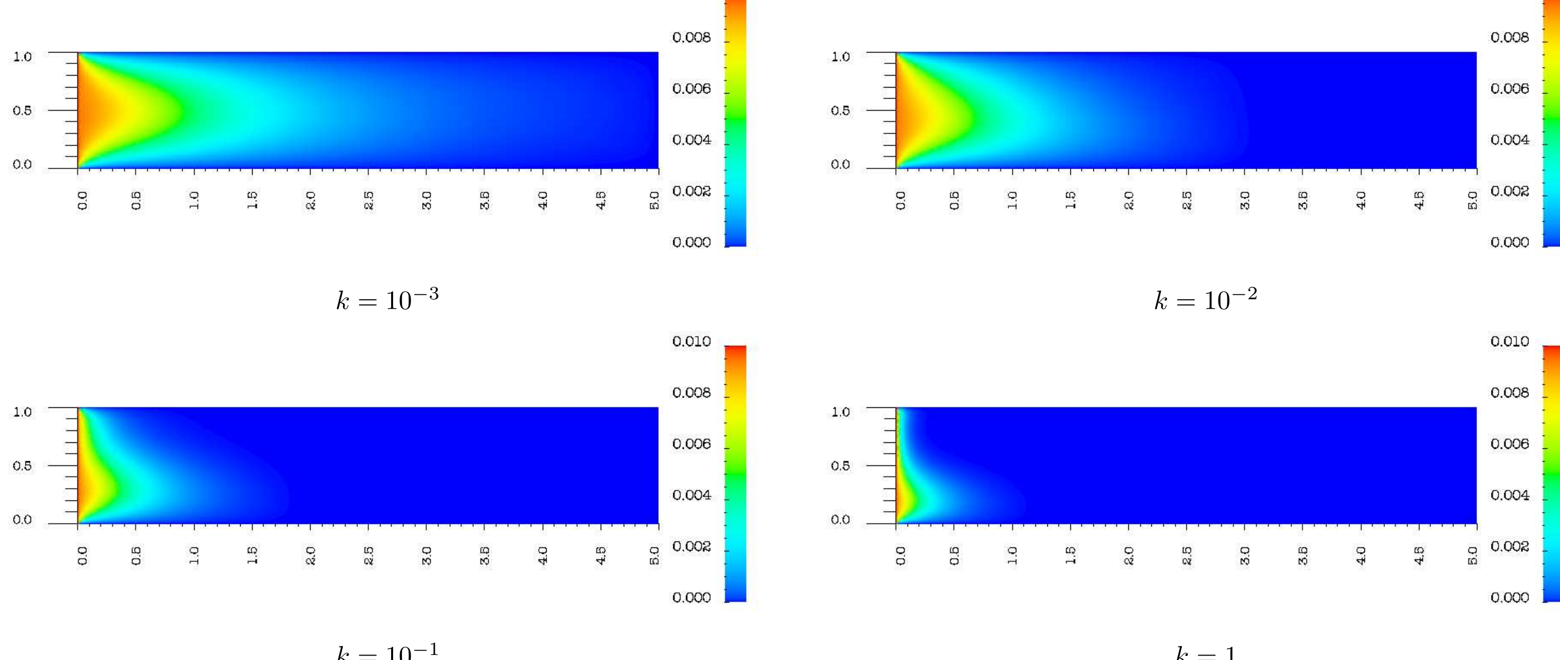
Figura 8.12: Gráficos de nível para a solução aproximada da espécie do Cenário 1.



Poluente

Espécie

Figura 8.13: Curvas de nível e densidade para a solução aproximada do Cenário 2 com $k = 0$.



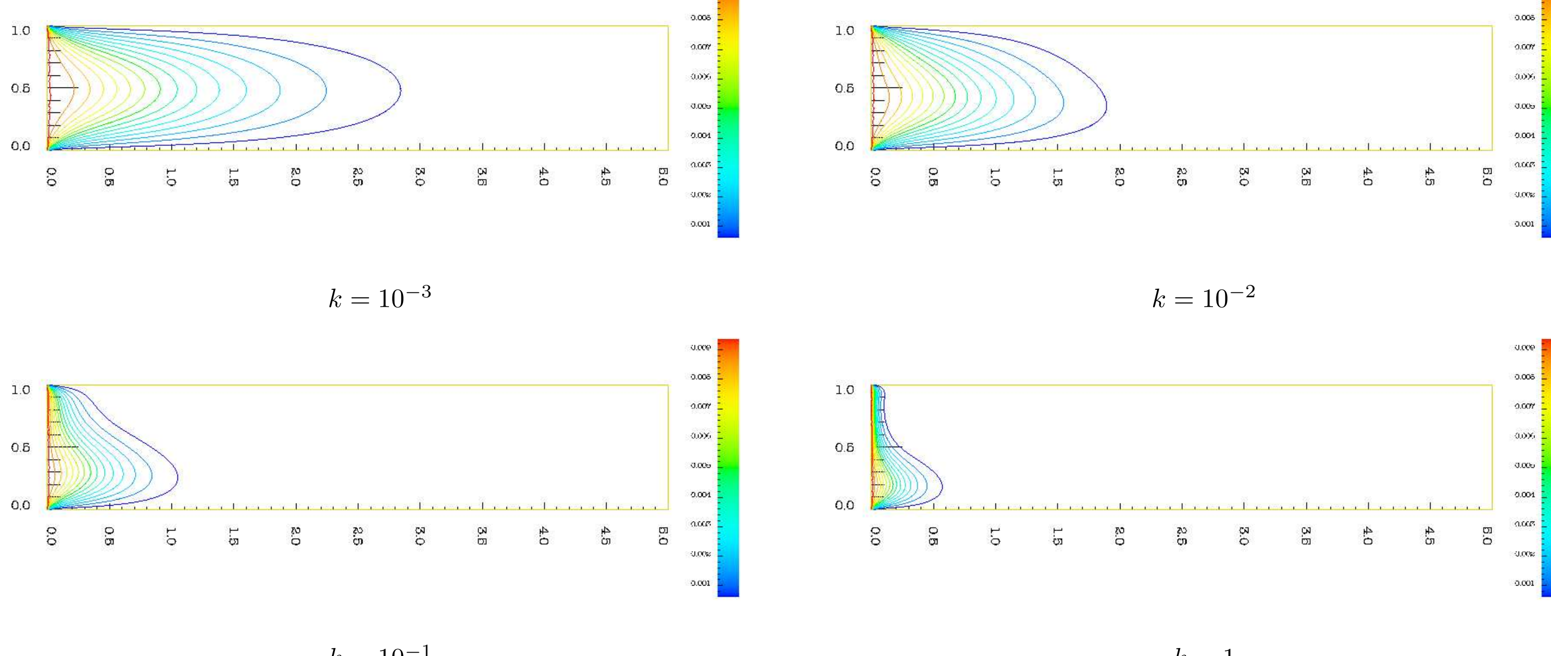
$k = 10^{-3}$

$k = 10^{-2}$

$k = 10^{-1}$

$k = 1$

Figura 8.14: Gráfico de densidade para a solução aproximada da espécie do Cenário 2.



$k = 10^{-3}$ $k = 10^{-2}$

$k = 10^{-1}$ $k = 1$

Figura 8.15: Gráficos de nível para a solução aproximada da espécie do Cenário 2.

Conclusões, faltas e trabalhos futuros

Conclusões gerais

Iniciamos este trabalho apresentando uma perspectiva sombria da utilização da linguagem C++ para aplicações de Computação Científica. As inúmeras facilidades trazidas pela filosofia da Orientação por Objetos acabavam por prejudicar o desempenho esperado neste tipo de aplicação.

Continuamos nosso trabalho mostrando as possibilidades inovadoras trazidas pela feliz descoberta de que o mecanismo de templates era muito mais do que uma conveniência sintática, o que praticamente tornou C++ uma metalinguagem, ou seja, uma linguagem para construir linguagens.

Estudamos o trabalho de Czarnecki e Eisenecker e *estendemos* os conceitos de repositórios de configurações ("objetos com DNA") para o de *interações entre repositórios de configurações*, criando assim um modelo ainda mais flexível para a construção de objetos de forma automática. Este modelo foi aplicado na criação de um código de elementos finitos válido para vários tipos de elemento, várias aproximações e dimensões.

Mostramos também que o mecanismo de expressões templates *pode ser usado em aplicações diversas* das usuais de álgebra linear computacional (operações entre matrizes e vetores). Naturalmente, como estamos operando em espaços discretos, intrinsicamente estamos fazendo estas operações, mas até onde sabemos, este uso parava aí. Nós o estendemos para uma aplicação de mais alto nível (elementos finitos) *sem a necessidade da criação de um mecanismo novo*, reaproveitando de forma integral o já presente na Pooma e, curiosamente, ampliando-o, sem contudo modificá-lo.

Por fim, acreditamos que as duas maiores contribuições trazidas por este trabalho foram a tentativa de explicar de forma didática todo o poder inerente a uma nova metodologia de desenvolvimento orientada a objetos, que não só contorna as deficiências conhecidas de C++ como também amplia seu poder de abstração, e a demonstração de que este tipo de desenvolvimento é viável e desejável.

Faltas

Durante grande parte deste trabalho mencionamos a palavra "desempenho". Apesar de números expressivos, como a avaliação do Laplaciano em 12 segundos numa malha com mais de 500000 elementos e mais de um milhão de graus de liberdade, serialmente e em um computador comum, o software aqui desenvolvido carece de comparações com outros softwares que oferecem características similares.

Esta comparação não foi feita por vários motivos. Dentre eles:

- Comparações desta espécie têm de ser feitas em nível de aplicação. Dado um determinado prob-

lema, quanto tempo e quanta memória são necessários para sua resolução?

- Existem dezenas de softwares de Elementos Finitos de todos os tipos (comerciais, livres, acadêmicos).
- Cada um destes softwares tem seu *modus operandi*: arquivos de entrada de malha, configurações, saída, etc. Mesmo de posse de toda a documentação para cada um, não há garantia total de que estaríamos resolvendo exatamente o mesmo problema.

Por outro lado, analisamos o código fonte de algumas bibliotecas que oferecem funcionalidades parecidas, ou seja, construção de simulações baseadas em elementos finitos. Apresentamos a seguir um breve resumo do que foi extraído dos códigos com destaque para as características que são implementadas de forma diferentes da de OSIRIS e que podem eventualmente reduzir a performance ou a generalidade.

deal.II A biblioteca deal.II[4] – desenvolvida na Universidade de Heidelberg (Alemanha) – oferece várias funcionalidades similiares as de OSIRIS, como várias dimensões e várias geometrias. Porém, as várias dimensões são definidas através de chaves de compilação e não através de técnicas generativas. Usa Orientação por Objetos no estilo clássico. Os conceitos por trás da simulação não são tão independentes quanto na Orientação por Aspectos. Não usa expressões templates e usa virtualidade.

OFELI A *Object Finite Element Library*[5] é ainda mais conservadora em termos de Orientação por Objetos. Os elementos são classificados não por aspectos geométricos como geometria ou dimensão, mas sim por tipo de problema (térmico, fluido ou sólido). Os conceitos de base e transformação local-global também não são independentes, fazendo parte da definição do elemento. Não usa expressões templates e não permite o encadeamento natural de operadores. Usa virtualidade de forma controlada. Não permite uma troca tão simples entre tipos de elemento ou dimensão do problema.

Rheolef Desenvolvida[6] no Laboratoire de Modélisation et Calcul em Grenoble é talvez a que mais se aproxime de OSIRIS no quesito modelagem de conceitos de elementos finitos. Ela define geometrias e espaços de forma bastante transparente para o usuário. Rheolef usa algumas técnicas avançadas de templates (traits, por exemplo), mas não utiliza expressões templates nem configuração automática de objetos. Dois pontos interessantes desta biblioteca são o pouco uso da virtualidade e a geração automática de código usando uma linguagem interpretada. Aparentemente usa ponteiros para funções em alguns pontos do código, o que pode vir a comprometer o desempenho.

Outro ponto em OSIRIS que merece ser destacado consiste na forma de imposição de condições de contorno. Da forma como está feito, para cada elemento no contorno são feitos testes para sabermos em qual condição ele se encontra. Este tipo de teste é feito em laços internos, causando uma perda desnecessária de eficiência computacional. Uma forma mais interessante de aplicação das condições de fronteira seria a extensão da estrutura de malha para o armazenamento por famílias. Assim, uma família do tipo Dirichlet não-nulo terá sua condição aplicada a todos os elementos a ela pertencentes de uma só vez, sem a necessidade de testes durante os cálculos.

[4] `http://gaia.iwr.uni-heidelberg.de/~deal`
[5] `http://ofeli.sourceforge.net`
[6] `http://www-lmc.imag.fr/lmc-edp/Pierre.Saramito/rheolef`

Trabalhos futuros

Como direções interessantes a serem tomadas, incluindo aí a possibilidade de futuras teses, citamos:

- A generalização dos operadores para campos vetoriais. Isto permitirá a resolução de problemas vetoriais (fluidos, por exemplo). Nós testamos uma versão de um programa que implementa um método para a resolução da equação de Stokes, mas cada velocidade tinha seu próprio campo e seu próprio conjunto de operadores, levando a uma duplicação desnecessária de código.

- A adaptação para uma estrutura de malhas mais inteligente. Nossa estrutura de malhas é bastante simples, baseada naquela presente na versão 2.3.0 do POOMA. Com a introdução da versão 2.4.0, nossa implementação ficou incompatível com a do POOMA, o que nos deixa livres para escolher uma mais robusta[7].

- A criação de novos operadores oriundos de outros problemas interessantes. Este teste é necessário para avaliar a flexibilidade da estrutura proposta.

- Testar a estrutura proposta com outros tipos de aproximação: Petrov-Galerkin, bolhas, elementos tipo P de alta ordem, etc.

- Recuperar a história de modelos desenvolvidos e testados dentro do grupo de Biomatemática do IMECC / UNICAMP ([Ber03], [Can98], [dAP02], [Sos03], [dO03], [Din03]). A maioria destes modelos foi desenvolvida usando-se alguma linguagem interpretada, notadamente Matlab. Com o uso de OSIRIS, estes mesmos modelos poderiam ser executados em malhas muito maiores, mais complexas, tridimensionais e com períodos de simulação maiores, podendo levar a conclusões mais abrangentes do que as exibidas nos citados trabalhos.

[7] De fato, a implementação do POOMA é para malhas estruturadas.

Referências Bibliográficas

[BBC+94] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA, 1994.

[Ber03] Mateus Bernardes. Poluição em corpos aquáticos de baixa circulação: Modelagem e simulação numérica. Master's thesis, Universidade Estadual de Campinas – UNICAMP, Instituto de Matemática, Estatística e Computação Científica – IMECC, 2003.

[Can98] Renato Fernandes Cantão. Simulação numérica de derrames de óleo: o caso do Canal de São Sebastião. Master's thesis, Universidade Estadual de Campinas – UNICAMP, Instituto de Matemática, Estatística e Computação Científica – IMECC, 1998.

[CCH+98a] James A. Crotinger, Julian C. Cummings, Scott W. Haney, William F. Humprey, Steve R. Karmesin, John V. W. Reynders, Stephen A. Smith, and Timothy J. Williams. Generic programming in POOMA and PETE. In *Dagstuhl Seminar on Generic Programming*. Schloβ Dagstuhl, apr 1998.

[CCH+98b] Julian C. Cummings, James A. Crotinger, Scott W. Haney, William F. Humprey, Steve R. Karmesin, John V. W. Reynders, Stephen A. Smith, and Timothy J. Williams. Rapid application development and enhanced code interoperability using the POOMA framework. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO'98)*. SIAM Press, oct 1998.

[CE99] Krzysztof Czarnecki and Ulrich W. Eisenecker. Synthesizing objects. In *Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP´99)*, pages 18–42. Department of Computer Science, University of Lisbon, jun 1999.

[CE00] K. Czarnecki and U. W. Eisenecker. *Generative Programming. Methods, Tools and Applications*. Addison-Wesley, 2000.

[CE01] Krzysztof Czarnecki and Ulrich. W. Eisenecker. Named parameters for configuration generators. Presented as a tutorial at the OOP'2001 Conference, jan. 2001. http://www.generative-programming.org/namedparams.

[Cun00] Maria Cristina Cunha. *Métodos numéricos*. Livro texto. Editora da UNICAMP, 2000.

[Cza00] Krysztof Czarnecki. Separating the configuration aspect to support architecture evolution. In *Proceedings of the ECOOP´2000 Workshop on "Aspects and dimensions of concerns"*, 2000.

[dAP02] Silvio de Alencastro Pregnolato. *Mal-das-cadeiras em capivaras: estudo, modelagem e simulação de um caso.* PhD thesis, Universidade Estadual de Campinas – UNICAMP, Faculdade de Engenharia Elétrica e Computação – FEEC, 2002.

[DD01] Harvey M. Deitel and Paul J. Deitel. *C++ Como Programar.* Bookman, 3a. edition, 2001.

[Din03] Geraldo Lúcio Diniz. *Dispersão de poluentes num sistema ar-água: modelagem, aproximação e aplicações.* PhD thesis, Universidade Estadual de Campinas – UNICAMP, Faculdade de Engenharia Elétrica e Computação – FEEC, 2003.

[dO03] Rosane Ferreira de Oliveira. *O comportamento evolutivo de uma mancha de óleo na Baía de Ilha Grande/RJ: modelagem, análise numérica e simulações.* PhD thesis, Universidade Estadual de Campinas – UNICAMP, Instituto de Matemática, Estatística e Computação Científica – IMECC, 2003.

[EBC00] Ulrich. W. Eisenecker, Frank Blinn, and Krzysztof Czarnecki. A solution to the constructor-problem of mixin-based programming in C++. Presented at the GCSE'2000 Workshop on C++ Template Programming, 2000.

[Eck95] Bruce Eckel. *Thinking in C++.* Prentice Hall, Englewood Cliffs, second edition, 1995. Also available Online at http://www.bruceeckel.com.

[GDL00] Thierry Géraud and Alexandre Duret-Lutz. Generic programming redesign of patterns. Technical report, EPITA Research and Development Laboratory, 2000.

[HC00] Scott Haney and James Crotinger. How templates enable high-performance scientific computing in C++. Technical report, Advanced Computing Laboratory, Los Alamos National Laboratory, 2000.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP).* Springer-Verlag, jun 1997.

[KN87] Hayrettin Kardestuncer and D. H. Norrie, editors. *Finite element handbook.* McGraw-Hill, 1987.

[KS99] George Em Karniadakis and Spencer J. Sherwin. *Spectral/hp element methods for CFD.* Numerical Mathematics and Scientific Computation. Oxford University Press, 1999.

[Mye95] Nathan C. Myers. Traits: a new and useful template technique. *C++ Report*, jun 1995.

[PT98] Los Alamos National Laboratory POOMA Team. POOMA tutorials. online, 1998.

[SB00] Yannis Smaragdakis and Don Batory. Mixin-based programming in C++. In *Proceedings of the Second International Symposium on Generative and Component-Based Software Engineering (GCSE´2000)*, pages 9–12, oct 2000.

[SL] Jeremy Siek and Andrew Lumsdaine. Concept Checking: binding parametric polymorphism in C++. Online.

[Sma99] Ioannis Smaragdakis. *Implementing Large-Scale Object-Oriented Components*. PhD thesis, Graduate School of The University of Texas at Austin, dec 1999.

[Sos03] Renata Cristina Sossae. *A presença evolutiva de um material impactante e seu efeito no transiente populacional de espécies interativas: modelagem e aproximação.* PhD thesis, Universidade Estadual de Campinas – UNICAMP, Instituto de Matemática, Estatística e Computação Científica – IMECC, 2003.

[Vel95a] Todd L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, jun 1995. Reprinted in C++ Gems, ed. Stanley Lippman.

[Vel95b] Todd L. Veldhuizen. Template metaprograms. *C++ Report*, 7(4):36–43, may 1995.

[Vel97] Todd L. Veldhuizen. Scientific computing: C++ versus Fortran: C++ has more than caught up. *Dr. Dobb's Journal of Software Tools*, 22(11):34, 36–38, 91, nov 1997.

[Vel00] Todd L. Veldhuizen. Techniques for scientific C++. Technical Report 542, Indiana University, Computer Science, aug 2000.

[VG98] Todd L. Veldhuizen and Dennis Gannon. Active libraries: Rethinking the roles of compilers and libraries. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Interoperable Scientific and Engineering Computing (OO'98)*. SIAM Press, oct 1998.

[VJ97] Todd L. Veldhuizen and M. Ed Jernigan. Will C++ be faster than Fortran? In *Proceedings of the 1st International Scientific Computing in Object Oriented Parallel Environments (ISCOPE´97)*, Lecture Notes in Computer Science. Springer-Verlag, 1997.

[VP96] Todd L. Veldhuizen and Kumaraswamy Ponnambalam. Linear algebra with C++ template metaprograms. *Dr. Dobb's Journal of Software Tools*, 21(8):38–44, aug 1996.

APÊNDICE A

Elementos de Referência

A.1 Triângulo de referência

O triângulo que utilizaremos como referência é o composto pelos vértices $(0,0)$, $(1,0)$ e $(0,1)$ no espaço (ξ,η), como representado na figura A.1. Ele será genericamente denotado por $\hat{e}$.

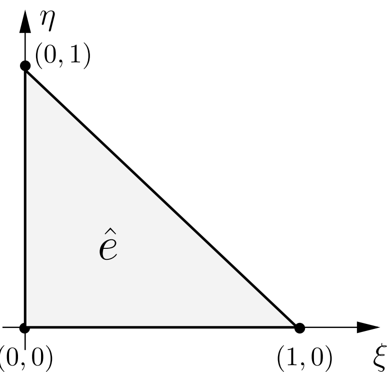


Figura A.1: Triângulo de referência.

Seja um triângulo de vértices (x_i, y_i), $i = 1, 2, 3$, no espaço (x, y). Este triângulo será representado por e. A *transformação* que mapeia pontos $(\xi,\eta) = \boldsymbol{\xi}$ dentro de $\hat{e}$ para pontos $(x, y) = \boldsymbol{x}$ em e pode ser escrita como $\boldsymbol{x} = \boldsymbol{J}(\boldsymbol{\xi})\boldsymbol{\xi} + \boldsymbol{b}$, com $\boldsymbol{b} \in \mathbb{R}^2$. De forma mais explícita, temos (A.1).

$$\begin{bmatrix} x(\xi,\eta) \\ y(\xi,\eta) \end{bmatrix} = \begin{bmatrix} x_2 - x_1 & x_3 - x_1 \\ y_2 - y_1 & y_3 - y_1 \end{bmatrix} \begin{bmatrix} \xi \\ \eta \end{bmatrix} + \begin{bmatrix} x_1 \\ y_1 \end{bmatrix}, \tag{A.1}$$

com $\boldsymbol{b} = [x_1, y_1]^T$. Desta forma, o *jacobiano* $\boldsymbol{J}(\boldsymbol{\xi})$ da transformação (A.1) é dado por (A.2)

$$\boldsymbol{J}(\boldsymbol{\xi}) = \begin{bmatrix} x_2 - x_1 & x_3 - x_1 \\ y_2 - y_1 & y_3 - y_1 \end{bmatrix}, \tag{A.2}$$

com respectivo *determinante* (A.3)

$$|\boldsymbol{J}(\boldsymbol{\xi})| = (x_2 - x_1)(y_3 - y_1) - (x_3 - x_1)(y_2 - y_1). \tag{A.3}$$

A *transformação inversa*, que mapeia o triângulo real e no triângulo padrão $\hat{e}$ pode ser escrita como $\boldsymbol{\xi} = \boldsymbol{J}^{-1}(\boldsymbol{x})(\boldsymbol{x} - \boldsymbol{b})$. Explicitamente podemos escrever a transformação inversa como em (A.4).

$$\begin{bmatrix} \xi(x,y) \\ \eta(x,y) \end{bmatrix} = \frac{1}{|\boldsymbol{J}(\boldsymbol{\xi})|} \begin{bmatrix} y_3 - y_1 & x_1 - x_3 \\ y_1 - y_2 & x_2 - x_1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} - \frac{1}{|\boldsymbol{J}(\boldsymbol{\xi})|} \begin{bmatrix} x_1 \\ y_1 \end{bmatrix}. \tag{A.4}$$

Portanto, o *jacobiano* da transformação inversa (A.4) é dado por (A.5)

$$\boldsymbol{J}^{-1}(\boldsymbol{x}) = \frac{1}{|\boldsymbol{J}(\boldsymbol{\xi})|} \begin{bmatrix} y_3 - y_1 & x_1 - x_3 \\ y_1 - y_2 & x_2 - x_1 \end{bmatrix}. \tag{A.5}$$

com respectivo *determinante* (A.6).

$$|\boldsymbol{J}^{-1}(\boldsymbol{x})| = |\boldsymbol{J}(\boldsymbol{x})|^{-1} = \frac{1}{(x_2 - x_1)(y_3 - y_1) - (x_3 - x_1)(y_2 - y_1)}. \tag{A.6}$$

A.2 Quadrilátero de referência

O quadrilátero de referência é composto pelos vértices $(-1,-1)$, $(1,-1)$, $(1,1)$ e $(-1,1)$ no espaço (ξ,η), como representado na figura A.2.

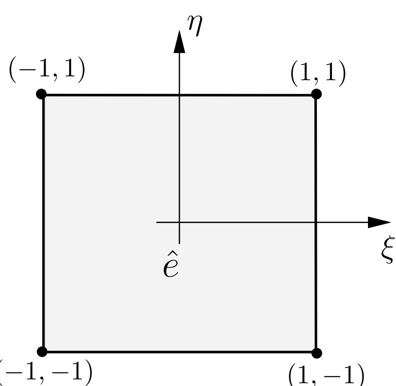


Figura A.2: Quadrilátero de referência.

Seja um quadrilátero de vértices (x_i, y_i), $i = 1, 2, 3, 4$, no espaço (x, y), representado por e. A transformação que mapeia pontos $(\xi,\eta) = \boldsymbol{\xi}$ dentro de $\hat{e}$ para pontos $(x,y) = \boldsymbol{x}$ em e pode ser escrita como $\boldsymbol{x} = \boldsymbol{F}(\boldsymbol{\xi})$, com $\boldsymbol{b} \in \mathbb{R}^2$. De forma mais explícita, temos (A.7).

$$\begin{bmatrix} x(\xi,\eta) \\ y(\xi,\eta) \end{bmatrix} = \begin{bmatrix} a + b\xi + c\eta + d\,\xi\eta \\ e + f\xi + g\eta + h\,\xi\eta \end{bmatrix}. \tag{A.7}$$

Agrupando-se os coeficientes a, b, ..., h de forma que $\boldsymbol{c}_x = [a, b, c, d]^T$ e $\boldsymbol{c}_y = [e, f, g, h]^T$ e agrupando-se as coordenadas de e como $\boldsymbol{e}_x = [x_1, x_2, x_3, x_4]^T$ e $\boldsymbol{e}_y = [y_1, y_2, y_3, y_4]^T$ temos:

$$\boldsymbol{c}_x = \frac{1}{4}\boldsymbol{A}^T\boldsymbol{e}_x \quad \text{e} \quad \boldsymbol{c}_y = \frac{1}{4}\boldsymbol{A}^T\boldsymbol{e}_y,$$

onde

$$\boldsymbol{A}^T = \begin{bmatrix} 1 & 1 & 1 & 1 \\ -1 & 1 & 1 & -1 \\ -1 & -1 & 1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix}.$$

Desta forma, o *jacobiano* $\boldsymbol{J}(\boldsymbol{\xi})$ da transformação (A.7) é dado por (A.8)

$$\boldsymbol{J}(\boldsymbol{\xi}) = \begin{bmatrix} b + d\,\eta & c + d\,\xi \\ f + h\,\eta & g + h\,\xi \end{bmatrix}, \tag{A.8}$$

com respectivo *determinante* (A.9)

$$|\boldsymbol{J}(\boldsymbol{\xi})| = bg - cf + (bh - df)\xi + (dg - ch)\eta. \tag{A.9}$$

Para escrevermos a transformação inversa, que mapeia o quadrilátero real e no quadrilátero padrão $\hat{e}$ usaremos a mesma notação de A.7.

$$\begin{bmatrix} \xi(x,y) \\ \eta(x,y) \end{bmatrix} = \begin{bmatrix} a + bx + cy + dxy \\ e + fx + gy + hxy \end{bmatrix}. \tag{A.10}$$

Neste caso, para determinarmos os coeficientes $\boldsymbol{c}_x$ e $\boldsymbol{c}_y$ (correspondentes a a, b, ..., h), devemos resolver os sistemas

$$\boldsymbol{B}\boldsymbol{c}_x = \boldsymbol{r}_x \quad \text{e} \quad \boldsymbol{B}\boldsymbol{c}_y = \boldsymbol{r}_y,$$

onde

$$\boldsymbol{B} = \begin{bmatrix} 1 & x_1 & y_1 & x_1y_1 \\ 1 & x_2 & y_2 & x_2y_2 \\ 1 & x_3 & y_3 & x_3y_3 \\ 1 & x_4 & y_4 & x_4y_4 \end{bmatrix},$$

$\boldsymbol{r}_x = [-1, 1, 1, -1]^T$ e $\boldsymbol{r}_y = [-1, -1, 1, 1]^T$.

O jacobiano da transformação inversa (A.10) é dado por (A.11)

$$\boldsymbol{J}^{-1}(\boldsymbol{x}) = \begin{bmatrix} b + dy & c + dx \\ f + hy & g + hx \end{bmatrix}. \tag{A.11}$$

com respectivo *determinante* (A.12).

$$|\boldsymbol{J}^{-1}(\boldsymbol{x})| = bg - cf + (bh - df)x + (dg - ch)y. \tag{A.12}$$

A.3 Tetraedro de referência

O tetraedro de referência é composto pelos vértices $(0,0,0)$, $(1,0,0)$, $(0,1,0)$ e $(0,0,1)$, no espaço (ξ, η, ζ), como representado na figura A.3.

Seja um tetraedro de vértices (x_i, y_i, z_i), $i = 1,2,3,4$, no espaço (x, y, z), também representado por e como no caso do triângulo. A *transformação* que mapeia pontos $(\xi, \eta, \zeta) = \boldsymbol{\xi}$ dentro de $\hat{e}$ para

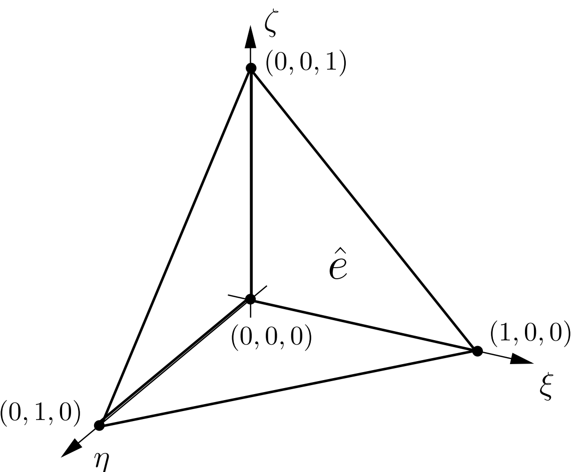


Figura A.3: Tetraedro de referência.

pontos $(x, y, z) = \boldsymbol{x}$ em e é escrita de forma análoga à do caso triangular: $\boldsymbol{x} = \boldsymbol{J}(\boldsymbol{\xi})\boldsymbol{\xi} + \boldsymbol{b}$, com $\boldsymbol{b} \in \mathbb{R}^3$. Especificamente, temos a expressão dada por (A.13).

$$\begin{bmatrix} x(\xi,\eta,\zeta) \\ y(\xi,\eta,\zeta) \\ z(\xi,\eta,\zeta) \end{bmatrix} = \begin{bmatrix} x_2 - x_1 & x_3 - x_1 & x_4 - x_1 \\ y_2 - y_1 & y_3 - y_1 & y_4 - y_1 \\ z_2 - z_1 & z_3 - z_1 & z_4 - z_1 \end{bmatrix} \begin{bmatrix} \xi \\ \eta \\ \zeta \end{bmatrix} + \begin{bmatrix} x_1 \\ y_1 \\ z_1 \end{bmatrix} \tag{A.13}$$

Desta forma, o *jacobiano* da transformação (A.13) é dado por (A.14)

$$\boldsymbol{J}(\boldsymbol{\xi}) = \begin{bmatrix} x_2 - x_1 & x_3 - x_1 & x_4 - x_1 \\ y_2 - y_1 & y_3 - y_1 & y_4 - y_1 \\ z_2 - z_1 & z_3 - z_1 & z_4 - z_1 \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}, \tag{A.14}$$

com respectivo *determinante* (A.15)

$$|\boldsymbol{J}(\boldsymbol{\xi})| = g\,(bf - ce) + h\,(cd - fh) + i\,(ae - bd), \tag{A.15}$$

usando as letras representativas em (A.14).

A *transformação inversa* de (A.13), seu jacobiano e respectivo determinante podem ser apreciados em (A.16), (A.17) e (A.18) (ainda usando as letras representativas de (A.14)).

$$\begin{bmatrix} \xi(x,y,z) \\ \eta(x,y,z) \\ \zeta(x,y,z) \end{bmatrix} = \frac{1}{|\boldsymbol{J}(\boldsymbol{\xi})|} \begin{bmatrix} ei - fh & ch - bi & bf - ce \\ fg - di & di - cg & cd - af \\ dh - eg & bg - ah & ae - bd \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} - \frac{1}{|\boldsymbol{J}(\boldsymbol{\xi})|} \begin{bmatrix} x_1 \\ y_1 \\ z_1 \end{bmatrix}, \tag{A.16}$$

$$\boldsymbol{J}^{-1}(\boldsymbol{x}) = \begin{bmatrix} ei - fh & ch - bi & bf - ce \\ fg - di & di - cg & cd - af \\ dh - eg & bg - ah & ae - bd \end{bmatrix}, \tag{A.17}$$

$$|\boldsymbol{J}^{\boldsymbol{-1}}(\boldsymbol{x})| = \frac{1}{|\boldsymbol{J}(\boldsymbol{x})|} = \frac{1}{g\,(bf - ce) + h\,(cd - fh) + i\,(ae - bd)}. \tag{A.18}$$

APÊNDICE B

Listagens

B.1 Meta SWITCH/CASE

```
// (C) Copyright Krzysztof Czarnecki & Ulrich Eisenecker 1998-2000.
// Permission to copy, use, modify, sell and distribute this software
// is granted provided this copyright notice appears in all
// copies.  In case of modification, the modified files should carry
// a notice stating that you changed the files.  This software is
// provided "as is"without express or implied warranty, and with no
// claim as to its suitability for any purpose.
//
// Cosmetic modifications by Renato Fernandes Cantão (2001).

#include "IF.h" // Meta IF.
#include "EQUAL.h" // Meta comparação de igualdade (EQUAL).

const int NilValue   = -32768;
const int DefaultValue = -32767;

struct NilCase {
     enum { tag = NilValue, foundCount = 0, defaultCount = 0 };

     typedef NilCase RET;
     typedef NilCase DEFAULT_RET;
};

class MultipleCaseHits {};
class MultipleDefaults {};

template< int Tag, class statement_t, class next_t = NilCase >
struct CASE {
     enum { tag = Tag };

     typedef statement_t Statement_t;
     typedef next_t Next_t;
```

```
32  };

    template< class statement_t, class next_t = NilCase >
34  struct DEFAULT {
        enum { tag = DefaultValue };
36
        typedef statement_t Statement_t;
38      typedef next_t Next_t;
    };


40  template< int Tag, class aCase_t, bool acceptMultipleCaseHits = false >
    struct SWITCH {
42      typedef typename aCase_t::Next_t NextCase_t;

44      enum { tag = aCase_t::tag, nextTag = NextCase_t::tag,
               found = (tag == Tag), isDefault = (tag == DefaultValue) };
46
        typedef typename IF< !EQUAL< nextTag, NilValue >::RET,
48          SWITCH< Tag, NextCase_t >,
            NilCase >::RET NextSwitch_t;
50
        enum { foundCount = found    + NextSwitch_t::foundCount,
52             defaultCount = isDefault + NextSwitch_t::defaultCount };

54      typedef typename IF< isDefault,
            typename aCase_t::Statement_t,
56          typename NextSwitch_t::DEFAULT_RET >::RET DEFAULT_RET;

58      typedef typename IF< found,
            typename IF< EQUAL< foundCount, 1 >::RET ||
60              EQUAL< acceptMultipleCaseHits, true >::RET,
                typename aCase_t::Statement_t,
62              MultipleCaseHits >::RET,
            typename IF< !EQUAL< foundCount, 0 >::RET,
64              typename NextSwitch_t::RET,
                DEFAULT_RET >::RET
66      >::RET ProvisionalRET;

68      typedef typename
        IF< EQUAL< defaultCount, 0 >::RET || EQUAL< defaultCount, 1 >::RET,
70          ProvisionalRET,
            MultipleDefaults >::RET RET;
72  };
```

Listagem B.1: Implementação do meta SWITCH/CASE.

B.2 Meta comparações

```
// Meta comparação de igualdade.
template< int n1, int n2 >
struct EQUAL
{
    enum { RET = (n1 == n2) };
};

// Meta comparação de desigualdade.
template< int n1, int n2 >
struct UNEQUAL
{
    enum { RET = (n1 != n2) };
};

// Meta comparação ‘‘menor que’’.
template< int T1, int T2 >
struct LT
{
    enum { RET = (T1 < T2) };
};

// Meta comparação ‘‘maior que’’.
template< int T1, int T2 >
struct BT
{
    enum { RET = (T1 > T2) };
};
```

Listagem B.2: Implementação de meta comparações.

APÊNDICE C

Listagens para os exemplos de Poisson

C.1 SolversCommon.h

O conteúdo de `SolversCommon.h` corresponde às configurações feitas na seção 8.1. As que mudam de exemplo para exemplo estão indicadas nos comentários na própria listagem.

```
1  /* SolversCommon.h
2   * Common includes and definitions for all example solvers.
3   */
4
5  #ifndef PANTHEON_SOLVERS_COMMON_INCLUDED
6  #define PANTHEON_SOLVERS_COMMON_INCLUDED
7
8  #include <iostream>
9
10 // Main Pantheon include.
11 #include "Pantheon.h"
12
13 // Numerical integration.
14 #include "CommonIntegrator.h"
15
16 // Mesh stuff.
17 #include "CommonMesh.h"
18 #include "MeshDataAccess.h"
19 #include "OsirisField.h"
20 #include "FieldInterpolator.h"
21
22 // Operators and alikes.
23 #include "Basis.generator.h"
24 #include "L2G_Map.generator.h"
25 #include "LocOp.generator.h"
26 #include "BC.generator.h"
27 #include "GradDotGrad.h"
28 #include "FunctionalRhs.h"
29
```

```
30  // Sparse engines.
31  #include "SparseArray.h"
32  #include "CompressedRowEngine.h"
33  #include "CanonicScather.h"
34
35  // Utilities and filters.
36  #include "CommandLine.h"
37  #include "Timers.h"
38  #include "StreamFilter.plain.h"
39  #include "PoomaExtensions.h"
40  #include "Greetings.h"
41
42  // Right hand sides.
43  #include "Rhs1.h" // Para Cenários 1, 2 e 3 (Poisson).
44  #include "Rhs2.h" // Para Cenário 4 (Poisson).
45
46  // Diffusion coefficients.
47  #include "DiffusionCoeffs.h"
48
49  using namespace std;
50
51  /*---------------------------------------------< Problem definitions >--*/
52
53  // Now we must include the element definition.
54  #include "ElementDef.h" // C.2
55
56  // Spatial dimension.
57  const int D = Element_t::spatialDimensions;
58
59  // The mesh (geometry).
60  typedef CommonMesh< Element_t > Mesh_t;
61
62  // Osiris field (problem variable).
63  typedef OsirisField< Mesh_t, Array< 1 > > Field_t;
64
65  // The FE basis.
66  typedef BASIS_GENERATOR< BASIS_Fe< Element_t >,
67                        BASIS_Type< grad_basis > >::RET Basis_t;
68
69  // Mapping.
70  typedef L2G_MAP_GENERATOR< L2G_FeGenerator< FeGenerator_t >,
71                          L2G_Geometry< Mesh_t >,
72                          L2G_MapKind< both_map > >::RET Mapping_t;
73
74  #endif // PANTHEON_SOLVERS_COMMON_INCLUDED
```

C.2 ElementDef.h

Em SolversCommon.h temos a definição do tipo de Elemento Finito que usamos nos cenários da seção 8.1. Esta é uma listagem padrão para um tipo de elemento; para outros tipos de elemento,

consulte a seção apropriada no capítulo 8.

```
1   /* ElementDef.h */
2
3   #ifndef PANTHEON_OSIRIS_ELDEF_INCLUDED
4   #define PANTHEON_OSIRIS_ELDEF_INCLUDED
5
6   // The element generator.
7   typedef FE_GENERATOR< FE_Family< lagrange< 1 > >,
8                         FE_Geometry< triangle > > FeGenerator_t;
9
10  // Other elements.
11
12  // Second order Lagrange triangle.
13  // typedef FE_GENERATOR< FE_Family< lagrange< 2 > >,
14  // FE_Geometry< triangle > > FeGenerator_t;
15
16  // First oreder Lagrange tetrahedral.
17  // typedef FE_GENERATOR< FE_Family< lagrange< 1 > >,
18  // FE_Geometry< tetra > > FeGenerator_t;
19
20  // First order Lagrange rectangle.
21  // typedef FE_GENERATOR< FE_Family< lagrange< 1 > >,
22  // FE_Geometry< rectangle > > FeGenerator_t;
23
24  // The element itself.
25  typedef FeGenerator_t::RET Element_t;
26
27  #endif
```

C.3 Rhs_1.h e Rhs_2.h

Em Rhs_1.h e Rhs_2.h estão os lados direitos usados nos exemplos da seção 8.1.

C.3.1 Rhs_1.h

```
1   /*  Rhs_1.h
2    *  RHS functions for Poisson / Scenarios 1, 2 and 3.
3    */
4
5   #include <iostream>
6
7   // Main Pantheon include.
8   #include "Pantheon.h"
9
10  // Functions.
11  #include "VectorToScalarFunctions.h"
12
13  using namespace std;
```

```
14  using namespace Pantheon;
15
16  class F1 {
17     public:
18        inline F1() {}
19
20        inline F1( real_t a, const Vector< 2 >& v = Vector< 2 >( 0.0, 0.0 ) ) {
21           alpha = a;
22
23           center = v;
24        }
25
26        inline ~F1() {}
27
28        V2S_UNARY_FUNCTION_ATOM( F1 ) {
29            return 2.0 * alpha * dot( v - center, 1.0 - ( v - center ) );
30        }
31
32        V2S_UNARY_FUNCTION_REGISTER( F1 );
33
34     private:
35        static real_t alpha;
36        static Vector< 2 > center;
37  };
38
39  real_t F1::alpha = 0.0;
40  Vector< 2 > F1::center;
41
42  V2S_UNARY_FUNCTION_IO( F1, real_t );
```

C.3.2 Rhs_2.h

```
1   /*  Rhs_2.h
2    *  RHS functions for Poisson / Scenario 4.
3    */
4
5   #include <iostream>
6
7   // Main Pantheon include.
8   #include "Pantheon.h"
9
10  // Functions.
11  #include "VectorToScalarFunctions.h"
12
13  using namespace std;
14  using namespace Pantheon;
15
16  class F13D {
17     public:
18        inline F13D() {}
19
```

```
20          inline F13D( real_t a,
21                   const Vector< 3 >& v = Vector< 3 >( 0.0, 0.0, 0.0 ) ) {
22             alpha = a;
23
24             center = v;
25          }
26
27          inline ~F13D() {}
28
29          V2S_UNARY_FUNCTION_ATOM( F13D ) {
30             Vector< 3 > p( v * ( 1-v ) );
31
32             return 2.0 * alpha * ( p( 1 ) * p( 2 ) + p( 0 ) * p ( 2 ) +
33                p( 0 ) * p( 1 ) );
34          }
35
36          V2S_UNARY_FUNCTION_REGISTER( F13D );
37
38       private:
39          static real_t alpha;
40          static Vector< 3 > center;
41    };
42
43    real_t F13D::alpha = 0.0;
44    Vector< 3 > F13D::center;
45
46    V2S_UNARY_FUNCTION_IO( F13D, real_t );
```

C.4 PoissonCteCoeff.h

A listagem `PoissonCteCoeff.h` define a classe `PoissonSolver`, que initializa todos os componentes da simulação, além de prover os métodos para a criação das matrizes globais do problema e sua posterior inversão com PETSc.

```
1     /*  PoissonCteCoeff.h
2      *  Definitions for the basic Poisson solver.
3      */
4
5     #include "SolversCommon.h"
6
7     #include "petscksp.h" // PETSc includes.
8     #include "PETSc_ConjugateGradient.h" // Linear system stuff.
9
10    typedef PETSc_ConjugateGradient::PETSc_Vector_t Solution_t;
11    typedef PETSc_ConjugateGradient::PETSc_Matrix_t SparseArray_t;
12
13    /*---------------------------------------------< Specific definitions >--*/
14
15    // Numerical integration.
16    typedef CommonIntegrator< DIFF_PTS, D > Integrator_t;
17    typedef CommonIntegrator< RHS_PTS, D > RhsIntegrator_t;
```

```
18
19    // Local Laplacian operator.
20    typedef LOCOP_GENERATOR<
21      LOCOP_Class    < galerkin     >,
22      LOCOP_Model    < divgrad< D > >,
23      LOCOP_SubModel < cte_parameter >,
24      LOCOP_Field    < Field_t      >,
25      LOCOP_Integrator< Integrator_t > >::RET CLaplacian_t;
26
27    // Boundary condition (null Dirichlet).
28    typedef BC_GENERATOR<
29      BC_Type< null_dirichlet< D > >,
30      BC_Field< Field_t > >::RET DirichletBC_t;
31
32    // RHS approximation method.
33    typedef FunctionalRHS< ForcingFunction_t, RhsIntegrator_t > RHS_t;
34
35    /*----------------------------------------------< Main class problem >--*/
36
37    class PoissonSolver {
38      public:
39        inline PoissonSolver( const char* f, real_t diff, uinteger_t NZ = 50 )
40          :
41          NSum(),     // Laplacian numerical integrator.
42          RhsNSum(),  // RHS numerical integrator.
43          mesh( f ),  // Finite element mesh.
44          mD( mesh ), // Mesh support class.
45          nEl( mD.numberOfElements() ), // Number of elements.
46          nC( mD.numberOfPositions() ), // Number of coordinates.
47          Map( mesh ), // L2G / G2L mapping.
48          u( mesh ),  // The solution field.
49          Laplacian( u, NSum, diff ), // Laplacian operator.
50          bc1( u, 1 ),  // Dirichlet boundary condition.
51          forcingFunc( diff ), // RHS.
52          Rhs( forcingFunc, RhsNSum ), // RHS evaluator.
53          A(),  // Sparse matrix.
54          b()  // RHS vector.
55        {
56          integer_t ierr = MatCreateSeqAIJ( PETSC_COMM_SELF, nC, nC, NZ,
57                                            PETSC_NULL, &A );
58
59          PETSC_ERR( ierr );
60
61          // Initializing PETSc vectors.
62
63          ierr = VecCreateSeq( PETSC_COMM_SELF, nC, &x ); PETSC_ERR( ierr );
64          ierr = VecDuplicate( x, &b );              PETSC_ERR( ierr );
65        }
66
67        // Destructor.
68        inline ~PoissonSolver() {}
```

```
69
70          // Assembles the FEM system.
71          inline void assembly() {
72            CanonicScather scather;
73
74            cout << "Basis␣NDOF␣=␣" << B.ndof << endl;
75            cout << "nC␣=␣" << nC << endl;
76            cout << "nEl␣=␣" << nEl << endl;
77
78            for( uinteger_t kk = 0; kk < nEl; kk++ ) {
79              Map.setup( kk ); // Never forget to prepare the mappings!
80
81              S = Laplacian.eval( B, Map ); // Evaluating the Laplacian.
82              V = Rhs.eval( B, Map ); // Evaluating the RHS.
83
84              bc1.eval( kk, S, V, B ); // Fixing boundary conditions.
85
86              // Assembling global matrix/vector.
87              scather.eval( A, S, b, V, mD, kk, B.ndof );
88            }
89
90            VecAssemblyBegin( b );
91            VecAssemblyEnd( b );
92
93            MatAssemblyBegin( A, MAT_FINAL_ASSEMBLY );
94            MatAssemblyEnd( A, MAT_FINAL_ASSEMBLY );
95
96          }
97
98          // Solves the resulting linear system.
99          inline void solve() {
100           PETSc_ConjugateGradient solver( A ); // Preparing the solver.
101           solver.solve( x, b ); // Solve linear system.
102           solver.destroy(); // Destroys the solver.
103         }
104
105         // Exports the solution to OpenDX.
106         inline void dx( const char* f ) {
107           const uinteger_t size = u.discrete().length( 0 );
108           PetscScalar* wp;
109           wp = new PetscScalar[ size ];
110           VecGetArray( x, &wp );
111
112           for( uinteger_t ii = 0; ii < size; ii++ ) u( ii ) = wp[ ii ];
113
114           VecRestoreArray( x, &wp );
115           SaveToDX( u, f );
116         }
117
118         // Destroy all PETSc contexts.
119         inline void destroy() {
```

```
120            integer_t ierr;
121
122            ierr = VecDestroy( x ); PETSC_ERR( ierr );
123            ierr = VecDestroy( b ); PETSC_ERR( ierr );
124            ierr = MatDestroy( A ); PETSC_ERR( ierr );
125          }
126
127      private:
128          // Numerical integrators.
129          Integrator_t NSum;
130          RhsIntegrator_t RhsNSum;
131          Mesh_t mesh; // The FEM mesh.
132          MeshDataAccess< Mesh_t > mD; // Mesh information extractor.
133          uinteger_t nEl, nC; // Number of elements and coordinates.
134          Mapping_t Map; // L2G map.
135          Field_t u; // Mesh + values.
136          CLaplacian_t Laplacian; // The differential operator.
137          DirichletBC_t bc1; // Boundary condition.
138          ForcingFunction_t forcingFunc; // RHS.
139          RHS_t Rhs;
140          Basis_t B; // FEM basis.
141          SparseArray_t A; // The sparse array.
142          Solution_t b, x; // RHS and solution vectors.
143          Tensor< Basis_t::ndof, real_t, Symmetric > S; // Local discrete operator.
144          Vector< Basis_t::ndof, real_t, Full  > V; // Local discrete RHS.
145    };
146
147    #endif // PANTHEON_POISSON_CTE_COEFF_INCLUDED
```

C.5 PoissonCteCoeff.cc

Finalmente chegamos ao arquivo que centraliza todas as chamadas necessárias para a execução da simulação do problema de Poisson.

```
1    /*  PoissonCteCoeff.cc
2     *  Solves \f$-\alpha \Delta u = f\f$, \f$u = 0\f$ in \f$\partial\Omega\f$.
3     */
4
5    #include "PoissonCteCoeff.h"
6
7    static char help[] = "Nothing\n\n";
8
9    /*-------------------------------------------------------------< MAIN >--*/
10
11   int PantheonMain( PantheonParameters& parameters )
12   {
13     // Hello!
14     PANTHEON_GREETINGS( "Osiris", "Poisson" );
15
16     // PETSc initialization.
17     PetscInitialize( &parameters._argc, &parameters._argv,
```

```
18                          static_cast< char* >( 0 ), help );
19
20      int ierr;
21
22      cout << "Problem␣setup..." << endl;
23      PoissonSolver Poisson( parameters.MeshFile, parameters.Diffusivity,
24                             parameters.NZ );
25
26      cout << "System␣assembly..." << endl;
27      Poisson.assembly();
28
29      cout << "System␣solution..." << endl;
30      Poisson.solve();
31
32      cout << "Exporting␣to␣OpenDX..." << endl;
33      Poisson.dx( parameters.DXFile );
34
35      cout << "Memory␣deallocation..." << endl;
36      Poisson.destroy();
37
38      ierr = PetscFinalize(); CHKERRQ( ierr );
39
40      PANTHEON_GOODBYE;
41
42      return 0;
43  }
```

APÊNDICE D

Listagens para os exemplos com difusão espacial

D.1 SolversCommon.h

O arquivo `SolversCommon.h` para os exemplos com difusão espacial (8.2) é praticamente o mesmo apresentado em C.1, à exceção das diretivas de inclusão de arquivo (**`#include`**), que apontam para os arquivos com os termos forçantes pertinentes.

D.2 ElementDef.h

É idêntico ao mostrado em C.2.

D.3 Rhs_1.h e BallSrc.h

O arquivo `Rhs_1.h` é idêntico ao mostrado em C.3. Para o Cenário 3 (tridimensional) usamos o termo forçante presente no arquivo `BallScr.h`, que representa uma difusão constante dentro de uma esfera específica, e zero fora dela.

D.3.1 BallSrc.h

```
1   /*  BallSrc.h
2    *  RHS function for Diffusion / Scenario 3.
3    */
4
5   #ifndef EX_BALL_SRC_H
6   #define EX_BALL_SRC_H
7
8   #include <iostream>
9
10  // Main Pantheon include.
11  #include "Pantheon.h"
12
```

```
13    // Functions.
14    #include "VectorToScalarFunctions.h"
15
16    using namespace std;
17    using namespace Pantheon;
18
19    class Ball
20    {
21      public:
22        inline Ball() {}
23
24        inline Ball( real_t a,
25                     const Vector< 3 >& v = Vector< 3 >( 0.5, 0.5, 0.5 ) ) {
26          alpha = a;
27          center = v;
28        }
29
30        inline ~Ball() {}
31
32        V2S_UNARY_FUNCTION_ATOM( Ball ) {
33          return ( sqrt( dot( center - v, center - v ) ) < 0.2 ? alpha : 0.0 );
34        }
35
36        V2S_UNARY_FUNCTION_REGISTER( Ball );
37
38      private:
39        static real_t alpha;
40        static Vector< 3 > center;
41    };
42
43    real_t Ball::alpha = 0.0;
44    Vector< 3 > Ball::center;
45
46    V2S_UNARY_FUNCTION_IO( Ball, real_t );
47
48    #endif
```

D.4 `Corner.h` e `HalfCube.h`

Nos cenários do capítulo 8.2 são utilizadas duas funções de difusão, uma para os Cenários 1 e 2 (arquivo `Corner.h`) e outra para o Cenário 3 (arquivo `HalfCube.h`).

D.4.1 `Corner.h`

```
1    /*  Corner.h
2     *  Diffusion functions with a corner form.
3     */
4
5    #ifndef EX_CORNER_H
6    #define EX_CORNER_H
```

```
7
8   #include <iostream>
9
10  // Main Pantheon include.
11  #include "Pantheon.h"
12
13  // Functions.
14  #include "VectorToScalarFunctions.h"
15
16  using namespace std;
17  using namespace Pantheon;
18
19  class Corner {
20    public:
21       inline Corner() {}
22
23       inline Corner( real_t in, real_t out ) {
24          Inner = in;
25          Outer = out;
26       }
27
28       inline ~Corner() {}
29
30       VD2S_UNARY_FUNCTION_ATOM( Corner, 2 ) {
31          if( ( fabs( v( 0 ) - 0.75 ) <= 0.25 ) &&
32              ( fabs( v( 1 ) - 0.75 ) <= 0.25 ) ) {
33             return Inner;
34          }
35
36          return Outer;
37       }
38
39       VD2S_UNARY_FUNCTION_REGISTER( Corner, 2 );
40
41    private:
42       static real_t Inner;
43       static real_t Outer;
44  };
45
46  real_t Corner::Inner = 0.0;
47  real_t Corner::Outer = 0.0;
48
49  VD2S_UNARY_FUNCTION_IO( Corner, 2, real_t );
50
51  #endif
```

D.4.2 HalfCube.h

```
1   /* HalfCube.h
2    * Diffusion function with half cube form.
3    */
4
```

```
5	#ifndef EX_HALF_CUBE_H
6	#define EX_HALF_CUBE_H
7
8	#include <iostream>
9
10	// Main Pantheon include.
11	#include "Pantheon.h"
12
13	// Functions.
14	#include "VectorToScalarFunctions.h"
15
16	using namespace std;
17	using namespace Pantheon;
18
19	class HalfCube {
20	   public:
21	      inline HalfCube() {}
22
23	      inline HalfCube( real_t in, real_t out ) {
24	         Inner = in;
25	         Outer = out;
26	      }
27
28	      inline ~HalfCube() {}
29
30	      VD2S_UNARY_FUNCTION_ATOM( HalfCube, 3 ) {
31	         return ( v( 2 ) < 0.5 ? Inner : Outer );
32	      }
33
34	      VD2S_UNARY_FUNCTION_REGISTER( HalfCube, 3 );
35
36	   private:
37	      static real_t Inner;
38	      static real_t Outer;
39	};
40
41	real_t HalfCube::Inner = 0.0;
42	real_t HalfCube::Outer = 0.0;
43
44	VD2S_UNARY_FUNCTION_IO( HalfCube, 3, real_t );
45
46	#endif
```

D.5 DoubleArch.h

No Cenário 2 usamos uma condição de contorno dada pela função g, conforme a tabela 8.13. Esta função está implementada no arquivo DoubleArch.h.

```
1	/* DoubleArch.h
2	 * Dirichlet evaluator for boundary conditions.
3	 */
4
```

```
5   #include <iostream>
6
7   // Main Pantheon include.
8   #include "Pantheon.h"
9
10  // Functions.
11  #include "DirichletEvaluator.h"
12
13  using namespace std;
14  using namespace Pantheon;
15
16  class DoubleArch : public DirichletEvaluator {
17     public:
18        typedef DirichletEvaluator Parent_t;
19
20        template< class field_t, class mesh_t >
21        inline DoubleArch( field_t& u, const mesh_t& m, uinteger_t Id ) :
22           Parent_t( u, m, Id ) {
23           eval( u, m, Id );
24        }
25
26        template< class field_t, class mesh_t >
27        inline void eval( field_t& u, const mesh_t& m, uinteger_t Id ) {
28           u.discrete() = where(
29              m.vertexFlags() == Id,
30              0.5 * m.vertexPositions().comp( 1 ) *
31              ( 1.0 - m.vertexPositions().comp( 1 ) ),
32              0.0 );
33        }
34  }; // end of class DoubleArch
```

D.6 Diffusion.h

A listagem `Diffusion.h` define a classe `DiffusionSolver`, que initializa todos os componentes da simulação, além de prover os métodos para a criação das matrizes globais do problema e sua posterior inversão com PETSc.

```
1   /*  Diffusion.h
2    *  Definitions for Diffusion.cc
3    */
4
5   #ifndef PANTHEON_DIFFUSION_INCLUDED
6   #define PANTHEON_DIFFUSION_INCLUDED
7
8   #include "SolversCommon.h"
9   #include "StreamFilter.octave.h"
10
11  #include "petscksp.h" // PETSc includes.
12  #include "PETSc_ConjugateGradient.h"
```

```
13
14  // Typedefs.
15  typedef PETSc_ConjugateGradient::PETSc_Vector_t Solution_t;
16  typedef PETSc_ConjugateGradient::PETSc_Matrix_t SparseArray_t;
17
18  /*---------------------------------------------< Specific definitions >--*/
19
20  // Numerical integration.
21  typedef CommonIntegrator< DIFF_PTS, D > Integrator_t;
22  typedef CommonIntegrator< RHS_PTS, D > RhsIntegrator_t;
23
24  // Local Laplacian operator.
25  typedef LOCOP_GENERATOR<
26    LOCOP_Class    < galerkin          >,
27    LOCOP_Model    < divgrad< D >      >,
28    LOCOP_SubModel < spatial_parameter >,
29    LOCOP_Field    < Field_t           >,
30    LOCOP_Integrator< Integrator_t    > >::RET CLaplacian_t;
31
32  // Boundary conditions.
33  typedef BC_GENERATOR< // Para Cenários 1 e 3.
34    BC_Type< null_dirichlet< D > >,
35    BC_Field< Field_t > >::RET DirichletBC_t;
36
37  #include "DoubleArch.h" // Para Cenário 2.
38
39  typedef BC_GENERATOR< // Para Cenário 2.
40    BC_DirichletEvaluator< DoubleArch >,
41    BC_Type< spatial_dirichlet< D > >,
42    BC_Field< Field_t > >::RET SpDirichletBC_t;
43
44  typedef BC_GENERATOR< // Para Cenário 2.
45    BC_Type< null_dirichlet< D > >,
46    BC_Field< Field_t > >::RET DirichletBC_t;
47
48  #include "HalfCube.h" // Para Cenário 3.
49
50  typedef BC_GENERATOR<  // Para Cenário 3.
51    BC_Type< null_dirichlet< D > >,
52    BC_Field< Field_t > >::RET DirichletBC_t;
53
54  // RHS approximation method.
55  typedef FunctionalRHS< ForcingFunction_t, RhsIntegrator_t > RHS_t;
56
57  /*-----------------------------------------------< Main class problem >--*/
58
59  // Dumb class to gather all information concerning a problem.
60
61  class DiffusionSolver {
62    public:
63      // Cenários 1 e 2.
```

```
64          inline DiffusionSolver( const char* f, uinteger_t NZ = 50 )
65          // Cenário 3.
66          inline DiffusionSolver( const char* f, real_t alpha, uinteger_t NZ = 50 )
67            :
68            NSum(),     // Laplacian numerical integrator.
69            RhsNSum(),  // RHS numerical integrator.
70            mesh( f ),  // Finite element mesh.
71            mD( mesh ), // Mesh support class.
72            nEl( mD.numberOfElements() ), // Number of elements.
73            nC( mD.numberOfPositions() ), // Number of coordinates.
74            Map( mesh ), // L2G / G2L mapping.
75            u( mesh ),  // The solution field.
76            Laplacian( u, NSum ), // Laplacian operator.
77            bc1( u, 1 ),   // Dirichlet boundary condition.
78            // Cenários 1 e 2.
79            forcingFunc( 1e-4 ), // RHS.
80            diffusion( 1e-4, 0.2e-4 ), // Diffusion coefficient.
81            // Cenário 3.
82            forcingFunc( 1.0 ), // RHS.
83            diffusion( alpha, alpha / 10.0 ), // Diffusion coefficient.
84            bc2( u, 2 ), // Spatial Dirichlet Cenário 2.
85            Rhs( forcingFunc, RhsNSum ), // RHS evaluator.
86            A(),   // Sparse matrix.
87            b()   // RHS vector.
88          {
89            integer_t ierr = MatCreateSeqAIJ( PETSC_COMM_SELF, nC, nC, NZ,
90                                              PETSC_NULL, &A );
91            PETSC_ERR( ierr );
92
93            // PETSc vectors.
94
95            ierr = VecCreateSeq( PETSC_COMM_SELF, nC, &x ); PETSC_ERR( ierr );
96            ierr = VecDuplicate( x, &b );              PETSC_ERR( ierr );
97          }
98
99          // Destructor.
100         inline ~DiffusionSolver() {}
101
102         // Assembles the FEM system.
103         inline void assembly() {
104           CanonicScather scather;
105
106           cout << "Basis␣NDOF␣=␣" << B.ndof << endl;
107           cout << "nC␣=␣" << nC << endl;
108           cout << "nEl␣=␣" << nEl << endl;
109
110           for( uinteger_t kk = 0; kk < nEl; kk++ ) {
111             Map.setup( kk ); // Never forget to prepare the mappings!
112
113             // Evaluating the Laplacian.
114             S = Laplacian.eval( B, Map, diffusion );
```

```
115             V = Rhs.eval( B, Map );
116
117             bc1.eval( kk, S, V, B ); // Fixing boundary conditions.
118             // Cenário 2.
119             bc2.eval( kk, S, V, B ); // Fixing boundary conditions.
120             scather.eval( A, S, b, V, mD, kk, B.ndof );
121           }
122
123           VecAssemblyBegin( b );
124           VecAssemblyEnd( b );
125
126           MatAssemblyBegin( A, MAT_FINAL_ASSEMBLY );
127           MatAssemblyEnd( A, MAT_FINAL_ASSEMBLY );
128         }
129
130         // Solves the resulting linear system.
131         inline void solve() {
132           PETSc_ConjugateGradient solver( A );
133           solver.solve( x, b );
134           solver.destroy();
135         }
136
137         // Exports the solution to OpenDX.
138         inline void dx( const char* f ) {
139           const uinteger_t size = u.discrete().length( 0 );
140           PetscScalar* wp;
141           wp = new PetscScalar[ size ];
142           VecGetArray( x, &wp );
143
144           for( uinteger_t ii = 0; ii < size; ii++ ) u( ii ) = wp[ ii ];
145
146           VecRestoreArray( x, &wp );
147           SaveToDX( u, f );
148         }
149
150         // Destroy all PETSc contexts.
151         inline void destroy() {
152           integer_t ierr;
153
154           ierr = VecDestroy( x ); PETSC_ERR( ierr );
155           ierr = VecDestroy( b ); PETSC_ERR( ierr );
156           ierr = MatDestroy( A ); PETSC_ERR( ierr );
157         }
158
159       private:
160         Integrator_t NSum; // Numerical integrators.
161         RhsIntegrator_t RhsNSum;
162         Mesh_t mesh;  // The FEM mesh.
163         MeshDataAccess< Mesh_t > mD; // Mesh information extractor.
164         uinteger_t nEl, nC; // Number of elements and coordinates.
165         Mapping_t Map; // L2G map.
```

```
166        Field_t u; // Mesh + values.
167        CLaplacian_t Laplacian; // The differential operator.
168        DirichletBC_t bc1; // Boundary condition.
169        SpDirichletBC_t bc2; // Dirichlet bc (Cenário 2.)
170        ForcingFunction_t forcingFunc; // RHS.
171        RHS_t Rhs;
172        Diffusion_t diffusion; // Diffusion.
173        Basis_t B; // FEM basis.
174        SparseArray_t A; // The sparse array.
175        Solution_t b, x; // RHS and solution vectors.
176        Tensor< Basis_t::ndof, real_t, Symmetric > S; // Local discrete operator.
177        Vector< Basis_t::ndof, real_t, Full  > V;  // Local discrete RHS.
178    };
179
180    #endif // PANTHEON_POISSON_CTE_COEFF_INCLUDED
```

D.7 Diffusion.cc

Finalmente chegamos ao arquivo que centraliza todas as chamadas necessárias para a execução da simulação de um problema com difusão espacialmente variável. Note a semelhança com seu equivalente de Poisson, `PoissonCteCoeff.cc`.

```
1     /*  Diffusion.cc
2      *  Solves \f$-\alpha(x) \Delta u = f\f$, \f$u = 0\f$ in \f$\partial\Omega\f$.
3      */
4
5     #include "Diffusion.h"
6
7     static char help[] = "Nothing\n\n";
8
9     /*------------------------------------------------------------< MAIN >--*/
10
11    int PantheonMain( PantheonParameters& parameters )
12    {
13      // Hello!
14      PANTHEON_GREETINGS( "Osiris", "Diffusion" );
15
16      // PETSc initialization.
17      PetscInitialize( &parameters._argc, &parameters._argv,
18                       static_cast< char* >( 0 ), help );
19
20      int ierr;
21
22      cout << "Problem␣setup..." << endl;
23
24      // Para Cenrios 1 e 2.
25      DiffusionSolver Diffusion( parameters.MeshFile, parameters.NZ );
26      // Para Cenrio 3.
27      DiffusionSolver Diffusion( parameters.MeshFile, parameters.alpha,
28                                 parameters.NZ );
```

```
29
30      cout << "System␣assembly..." << endl;
31      Diffusion.assembly();
32
33      cout << "System␣solution..." << endl;
34      Diffusion.solve();
35
36      cout << "Exporting␣to␣OpenDX..." << endl;
37      Diffusion.dx( parameters.DXFile );
38
39      cout << "Memory␣deallocation..." << endl;
40      Diffusion.destroy();
41
42      ierr = PetscFinalize(); CHKERRQ( ierr );
43
44      PANTHEON_GOODBYE;
45
46      return 0;
47  }
```

APÊNDICE E

Listagem para os exemplos de Interação Espécie/Poluente

Os exemplos do capítulo 8.3 foram criados com um único arquivo fonte `PollutantSpecies.cc`, que contém todas as definições necessárias tanto para o poluente quanto para a espécie.

E.1 PollutantSpecies.cc

```
1  /* PollutantSpecies.cc */
2
3  #include "Pantheon.h"
4
5  // Numerical integration.
6  #include "CommonIntegrator.h"
7
8  // Mesh stuff.
9  #include "CommonMesh.h"
10 #include "MeshDataAccess.h"
11 #include "OsirisField.h"
12 #include "FieldInterpolator.h"
13
14 // Operators and alikes.
15 #include "Basis.generator.h"
16 #include "L2G_Map.generator.h"
17 #include "LocOp.generator.h"
18 #include "BC.generator.h"
19 #include "GradDotGrad.h"
20 #include "FunctionalRhs.h"
21 #include "CanonicScather.h"
22
23 // Utilities and filters.
24 #include "CommandLine.h"
25 #include "Timers.h"
```

```
26  #include "StreamFilter.plain.h"
27  #include "StreamFilter.dx.h"
28  #include "PoomaExtensions.h"
29  #include "Greetings.h"
30
31  // Right hand sides.
32  #include "Rhs.h"
33
34  // Diffusions.
35  #include "Alpha.h"
36
37  #include "petscksp.h" // PETSc includes.
38  #include "PETSc_ConjugateGradient.h"
39
40  // Typedefs.
41  typedef PETSc_ConjugateGradient::PETSc_Vector_t Solution_t;
42  typedef PETSc_ConjugateGradient::PETSc_Matrix_t SparseArray_t;
43
44  static char help[] = "Nothing\n\n";
45
46  /*---------------------------------------------------< Problem definitions >--*/
47
48  // The element (geometric unit).
49  typedef FE_GENERATOR< FE_Family< lagrange< 2 > >,
50                     FE_Geometry< triangle > > FeGenerator_t;
51
52  typedef FeGenerator_t::RET Element_t;
53
54  // Spatial dimension.
55  const int D = Element_t::spatialDimensions;
56
57  // The mesh (geometry).
58  typedef CommonMesh< Element_t > Mesh_t;
59
60  // Osiris field (problem variable).
61  typedef OsirisField< Mesh_t, Array< 1 > > Field_t;
62
63  // Numerical integration.
64  typedef CommonIntegrator< 2, D > Integrator_t;
65
66  typedef CommonIntegrator< 4, D > RhsIntegrator_t;
67
68  // Mapping.
69  typedef L2G_MAP_GENERATOR< L2G_FeGenerator< FeGenerator_t >,
70                         L2G_Geometry< Mesh_t >,
71                         L2G_MapKind< both_map > >::RET Mapping_t;
72
73  // Variable diffusivity operator.
74  typedef LOCOP_GENERATOR<
75    LOCOP_Class    < galerkin     >,
76    LOCOP_Model    < divgrad< D > >,
```

```
77        LOCOP_SubModel < spatial_parameter >,
78        LOCOP_Field    < Field_t       >,
79        LOCOP_Integrator< RhsIntegrator_t > >::RET SpatialDiffusion_t;
80
81      // Constant identity operator.
82      typedef LOCOP_GENERATOR<
83        LOCOP_Class    < galerkin      >,
84        LOCOP_Model    < identity< D > >,
85        LOCOP_SubModel < cte_parameter >,
86        LOCOP_Field    < Field_t       >,
87        LOCOP_Integrator< Integrator_t > >::RET CteIdentity_t;
88
89      // Local advection operator.
90      typedef LOCOP_GENERATOR<
91        LOCOP_Class    < galerkin       >,
92        LOCOP_Model    < advective< D > >,
93        LOCOP_SubModel < cte_parameter >,
94        LOCOP_Field    < Field_t        >,
95        LOCOP_Integrator< Integrator_t > >::RET CteAdvection_t;
96
97      // Constant diffusivity operator.
98      typedef LOCOP_GENERATOR<
99        LOCOP_Class    < galerkin      >,
100       LOCOP_Model    < divgrad< D > >,
101       LOCOP_SubModel < cte_parameter >,
102       LOCOP_Field    < Field_t       >,
103       LOCOP_Integrator< Integrator_t > >::RET CteDiffusion_t;
104
105     // Variable identity operator.
106     typedef LOCOP_GENERATOR<
107       LOCOP_Class    < galerkin      >,
108       LOCOP_Model    < identity< D > >,
109       LOCOP_SubModel < spatial_parameter >,
110       LOCOP_Field    < Field_t       >,
111       LOCOP_Integrator< Integrator_t > >::RET SpatialIdentity_t;
112
113     // Dirichlet boundary condition.
114     typedef BC_GENERATOR<
115       BC_Type< null_dirichlet< D > >,
116       BC_Field< Field_t > >::RET NullDirichletBC_t;
117
118     // Use esta condição para Dirichlet não-nulo.
119     typedef BC_GENERATOR<
120       BC_Type< cte_dirichlet< D > >,
121       BC_Field< Field_t > >::RET CteDirichletBC_t;
122
123     typedef BASIS_GENERATOR< BASIS_Fe< Element_t >,
124                             BASIS_Type< grad_basis > >::RET Basis_t;
125
126     /*--------------------------------------------------------< Velocity >--*/
127
```

```
128  // Reads a cte velocity field from a file.
129
130  Vector< D > Velocity( const char* FileName ) {
131     ifstream in( FileName );
132
133     if( !in ) {
134        PANTHEON_FATAL_ERROR( Osiris, "Unable␣to␣open␣file!" );
135     }
136
137     Vector< D > V;
138
139     for( int ii = 0; ii < D; ii++ ) in >> V( ii );
140
141     return V;
142  }
143
144  /*-----------------------------------------------< Pollutant diffusion >--*/
145
146  class PolDiffusion {
147     public:
148        inline PolDiffusion() {}
149        inline ~PolDiffusion() {}
150
151        VD2S_UNARY_FUNCTION_ATOM( PolDiffusion, 2 ) {
152           return 1e-5 + 1e-5 * v( 1 ) * v( 1 );
153        }
154
155        VD2S_UNARY_FUNCTION_REGISTER( PolDiffusion, 2 );
156  }; // end of class PolDiffusion
157
158  /*--------------------------------------------------< Pollutant source >--*/
159
160  class PolSrc {
161     public:
162        inline PolSrc() {}
163        inline ~PolSrc() {}
164
165        VD2S_UNARY_FUNCTION_ATOM( PolSrc, 2 ) {
166           if( ( fabs( v( 0 ) - 4.25 ) <= 0.5 ) &&
167               ( fabs( v( 1 ) - 0.20 ) <= 0.1 ) ) {
168              return 1e-3;
169           }
170
171           return 0.0;
172        }
173
174        VD2S_UNARY_FUNCTION_REGISTER( PolSrc, 2 );
175  }; // end of class PolSrc
176
177  /*----------------------------------------------------< Species source >--*/
178
```

```
179   class SpeciesSrc {
180     public:
181       inline SpeciesSrc() {}
182       inline ~SpeciesSrc() {}
183
184       VD2S_UNARY_FUNCTION_ATOM( SpeciesSrc, 2 ) {
185         // Retire este ''if'' para Dirichlet não-nulo.
186         if( v( 0 ) <= 0.5 ) {
187           return 1e-2;
188         }
189
190         return 0.0;
191       }
192
193       VD2S_UNARY_FUNCTION_REGISTER( SpeciesSrc, 2 );
194   }; // end of class PolSrc
195
196   /*-------------------------------------------------------------< MAIN >--*/
197
198   int PantheonMain( PantheonParameters& parameters )
199   {
200     // Hello!
201     PANTHEON_GREETINGS( "Osiris", "PollutantSpecies" );
202
203     // PETSc initialization.
204     PetscInitialize( &parameters._argc, &parameters._argv,
205                      static_cast< char* >( 0 ), help );
206
207     // Numeric integration.
208     Integrator_t NSum;
209
210     RhsIntegrator_t RhsNSum;
211
212     // Evaluating the basis on the integration points.
213     Basis_t B;
214
215     cout << "PROCESSING␣POLLUTANT␣PROBLEM..." << endl;
216
217     cout << "Mesh␣processing..." << endl;
218
219     // Reading a mesh from a GMSH file.
220     Mesh_t mesh( parameters.MeshFile );
221
222     // Standard mappings.
223     Mapping_t Map( mesh );
224
225     MeshDataAccess< Mesh_t > mD( mesh );
226
227     // Linking this mesh to a Osiris field (pollutant).
228     Field_t u( mesh );
229
```

```
230     // Pollutant diffusion function.
231     PolDiffusion cDiff;
232
233     // Pollutant decay coefficient.
234     const real_t cDecay = 1e-7;
235
236     // Declaring a variable diffusion operator.
237     SpatialDiffusion_t SpDiff( u, RhsNSum );
238
239     // Constant identity operator.
240     CteIdentity_t Decay( u, NSum, cDecay );
241
242     // Declaring the advective operator.
243     CteAdvection_t Advection( u, NSum, Velocity( parameters.PolVelFile ) );
244
245     // Null Dirichlet boundary condition.
246     NullDirichletBC_t bc1( u, 1 );
247
248     // RHS for the first equation.
249     PolSrc f1;
250
251     FunctionalRHS< PolSrc, Integrator_t > Rhs( f1, NSum );
252
253     // Some mesh quantities.
254
255     const uinteger_t nel = mD.numberOfElements();
256     const uinteger_t nc = mD.numberOfPositions();
257
258     SparseArray_t A;
259
260     integer_t ierr;
261
262     ierr = MatCreateSeqAIJ( PETSC_COMM_SELF, nc, nc, parameters.NZ,
263                             PETSC_NULL, &A );
264
265     PETSC_ERR( ierr );
266
267     Solution_t b, x;
268
269     ierr = VecCreateSeq( PETSC_COMM_SELF, nc, &b ); PETSC_ERR( ierr );
270     ierr = VecDuplicate( b, &x );               PETSC_ERR( ierr );
271
272     uinteger_t id1, id2;
273
274     Tensor< B.ndof, real_t, Full > S; // Local discrete operator.
275     Vector< B.ndof, real_t, Full > V;   // Local discrete RHS.
276
277     CanonicScather scather;
278
279     cout << "Operator␣processing..." << endl;
280
```

```
281     for( uinteger_t kk = 0; kk < nel; kk++ ) {
282       Map.setup( kk ); // Never forget to prepare the mappings!
283
284       S = SpDiff.eval( B, Map, cDiff ) + Decay.eval( B, Map ) +
285           Advection.eval( B, Map );
286
287       V = Rhs.eval( B, Map );
288       bc1.eval( kk, S, V, B ); // Fixing boundary conditions.
289
290       scather.eval( A, S, b, V, mD, kk, B.ndof );
291     }
292
293     VecAssemblyBegin( b );
294     VecAssemblyEnd( b );
295
296     MatAssemblyBegin( A, MAT_FINAL_ASSEMBLY );
297     MatAssemblyEnd( A, MAT_FINAL_ASSEMBLY );
298
299     cout << "Linear␣system␣solution..." << endl;
300
301     // Preparing the solver.
302     PETSc_ConjugateGradient solver( A );
303
304     // Solve linear system.
305     solver.solve( x, b );
306
307     // Destroys the solver.
308     solver.destroy();
309
310     cout << "PROCESSING␣SPECIES␣PROBLEM..." << endl;
311
312     // Species field.
313     Field_t s( mesh );
314
315     // Species cte diffusion.
316     const real_t cSpDiff = 1e-4;
317
318     // Scalling the pollutant.
319     u.discrete() *= parameters.Multiplier;
320
321     // Declaring a cte diffusion operator.
322     CteDiffusion_t CteDiff( s, NSum, cSpDiff );
323
324     // Pollutant interpolator.
325     typedef FieldInterpolator< Field_t, Basis_t > IField_t;
326
327     IField_t uI( u, B );
328
329     // Spatial decay.
330     SpatialIdentity_t SpDecay( s, NSum );
331
```

```
332     // Declaring the advective operator.
333     CteAdvection_t Advection2( s, NSum, Velocity( parameters.SpVelFile ) );
334
335     // RHS for the second equation.
336     SpeciesSrc f2;
337
338     FunctionalRHS< SpeciesSrc, Integrator_t > Rhs2( f2, NSum );
339
340     ierr = MatDestroy( A ); PETSC_ERR( ierr );
341     ierr = MatCreateSeqAIJ( PETSC_COMM_SELF, nc, nc, parameters.NZ,
342                             PETSC_NULL, &A ); PETSC_ERR( ierr );
343
344     Solution_t b2;
345
346     ierr = VecCreateSeq( PETSC_COMM_SELF, nc, &b2 ); PETSC_ERR( ierr );
347
348     // Use se Dirichlet for não-nulo.
349     CteDirichletBC_t bc2( u, 2, 1e-2 );
350
351     for( uinteger_t kk = 0; kk < nel; kk++ ) {
352        Map.setup( kk ); // Never forget to prepare the mappings!
353        uI.setup( kk );
354
355        S = CteDiff.eval( B, Map ) + SpDecay.eval( B, Map, uI ) +
356            Advection2.eval( B, Map );
357
358        V = Rhs2.eval( B, Map );
359        bc1.eval( kk, S, V, B ); // Fixing boundary conditions.
360        // Use se Dirichlet for não-nulo.
361        bc2.eval( kk, S, V, B ); // Fixing boundary conditions.
362
363        scather.eval( A, S, b2, V, mD, kk, B.ndof );
364     }
365
366     VecAssemblyBegin( b2 );
367     VecAssemblyEnd( b2 );
368
369     MatAssemblyBegin( A, MAT_FINAL_ASSEMBLY );
370     MatAssemblyEnd( A, MAT_FINAL_ASSEMBLY );
371
372     // Preparing the solver.
373     PETSc_ConjugateGradient solver2( A );
374
375     // Solve linear system.
376     solver2.solve( x, b2 );
377
378     // Destroys the solver.
379     solver2.destroy();
380
381     ierr = VecDestroy( x ); PETSC_ERR( ierr );
382     ierr = VecDestroy( b ); PETSC_ERR( ierr );
```

```
383        ierr = MatDestroy( A ); PETSC_ERR( ierr );
384
385        ierr = PetscFinalize(); CHKERRQ( ierr );
386
387        PANTHEON_GOODBYE;
388
389        return 0;
390     }
```