Pontifícia Universidade Católica
do Rio de Janeiro

Carlos Augusto Teixeira Mendes

GeMA, um novo framework para a prototipação, desenvolvimento e integração de simulações multifísicas e multiescalas em grupos multidisciplinares

Tese de Doutorado

Tese apresentada ao Programa de Pós–graduação em Informática do Departamento de Informática da PUC–Rio como requisito parcial para obtenção do título de Doutor em Informática

Orientador: Prof. Marcelo Gattass

Rio de Janeiro
Abril de 2016

Pontifícia Universidade Católica
do Rio de Janeiro

Carlos Augusto Teixeira Mendes

GeMA, um novo framework para a prototipação, desenvolvimento e integração de simulações multifísicas e multiescalas em grupos multidisciplinares

Tese apresentada ao Programa de Pós–graduação em Informática do Departamento de Informática do Centro Técnico Científico da PUC–Rio como requisito parcial para obtenção do título de Doutor em Informática. Aprovada pela Comissão Examinadora abaixo assinada.

Prof. Marcelo Gattass
Orientador
Departamento de Informática — PUC–Rio

Prof. Hélio Côrtes Vieira Lopes
Departamento de Informática — PUC–Rio

Profa. Deane de Mesquita Roehl
PUC–Rio

Dr. Márcio Arab Murad
LNCC

Dr. João Luiz Elias Campos
IBM Research

Prof. Waldemar Celes Filho
Departamento de Informática — PUC–Rio

Prof. Márcio da Silveira Carvalho
Coordenador Setorial do Centro Técnico Científico — PUC–Rio

Rio de Janeiro, 01 de Abril de 2016

Carlos Augusto Teixeira Mendes

Possui graduação em Engenharia de Computação (PUC-Rio, 1996) e mestrado em Informática com ênfase em redes de computadores e sistemas distribuídos (PUC-Rio, 1999). É sócio desde 1997 da K2 Sistemas, possuindo experiência em projetos de desenvolvimento de software voltados para a área científica. Atualmente trabalha no Instituto Tecgraf/PUC-Rio, onde atua na concepção e implementação de um novo *framework* para suporte ao desenvolvimento de simulações multi-físicas.

Ficha Catalográfica

Mendes, Carlos Augusto Teixeira

GeMA, um novo framework para a prototipação, desenvolvimento e integração de simulações multifísicas e multiescalas em grupos multidisciplinares / Carlos Augusto Teixeira Mendes; orientador: Prof. Marcelo Gattass. — Rio de Janeiro : PUC–Rio, Departamento de Informática, 2016.

v., 168 f: il. ; 29,7 cm

1. Tese (doutorado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática.

Inclui referências bibliográficas.

1. Informática – Tese. 2. Simulação. 3. Framework. 4. Multifísica. 5. Multiescala. I. Gattass, Marcelo. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

À Flávia, Carolina e Mariana.

Agradecimentos

A minha esposa, Flávia, pelo apoio, paciência e compreensão.

A minhas filhas, Carolina e Mariana, para quem o tempo demorou a passar.

A toda minha família e amigos, que de uma forma ou de outra me estimularam ou me ajudaram.

Ao meu orientador, professor Marcelo Gattass, pelo apoio, atenção, conselhos e incentivo durante toda a realização deste trabalho.

Aos colegas do sexto andar no Instituto Tecgraf, Erwan Renaut, Jéferson Coêlho e Murillo Santana pela constante troca de ideias, sugestões e contribuições ao *framework* GeMA.

A Carlos Henrique Levy, principal incentivador deste trabalho.

À K2 Sistemas, ao CNPq, à PUC-Rio e ao Instituto Tecgraf, pelos auxílios concedidos, sem os quais este trabalho não poderia ter sido realizado.

Resumo

Mendes, Carlos Augusto Teixeira; Gattass, Marcelo. **GeMA, um novo framework para a prototipação, desenvolvimento e integração de simulações multifísicas e multiescalas em grupos multidisciplinares**. Rio de Janeiro, 2016. 168p. Tese de Doutorado — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

A exploração e produção de petróleo é uma tarefa complexa onde a utilização de modelos físicos é fundamental para minimizar riscos exploratórios e maximizar o retorno do capital investido durante a etapa de produção dos campos descobertos. Com o passar do tempo, estes modelos vêm se tornando cada vez mais complexos, apresentando uma grande tendência de integração entre simuladores distintos e dando origem à necessidade de novas simulações multifísicas, onde modelos físicos isolados são resolvidos conjuntamente de maneira acoplada. Este trabalho apresenta o *framework* GeMA (Geo Modelling Analysis framework), uma biblioteca para suporte ao desenvolvimento de novos simuladores multifísicos, permitindo tanto o acoplamento de novos modelos construídos tendo o *framework* como base, quanto a integração com simuladores já existentes. Seu objetivo é promover a utilização de técnicas de engenharia de software tais como extensibilidade, reusabilidade, modularidade e portabilidade na construção de modelos físicos para engenharia, permitindo que engenheiros estejam livres para se concentrarem na formulação física do problema, uma vez que o *framework* se encarrega do gerenciamento de dados e das funções de suporte necessárias, agilizando a produção de código. Construído para auxiliar durante todo o fluxo de trabalho de uma simulação multifísica, a arquitetura do *framework* suporta múltiplos paradigmas de simulação e acoplamento de físicas, com especial ênfase no método de elementos finitos, sendo capaz de representar o domínio espacial através de múltiplas discretizações (malhas) e efetuar a troca de valores entre as mesmas. O *framework* implementa ainda conceitos importantes de extensibilidade, através do uso combinado de "plugins" e interfaces abstratas, bem como orquestração configurável e prototipação rápida através do uso da linguagem Lua. Além da descrição do *framework*, este trabalho apresenta ainda um conjunto de testes aplicados para testar sua corretude e expressividade, com especial ênfase em um modelo 2D de modelagem de bacias acoplando cálculo não linear de temperatura baseado em elementos finitos, compactação mecânica, maturação e geração de hidrocarbonetos.

Palavras–chave

Simulação; Framework; Multifísica; Multiescala;

Abstract

Mendes, Carlos Augusto Teixeira; Gattass, Marcelo (advisor). **GeMA, a new framework for prototyping, development and integration of multiphysics and multiscale simulations in multidisciplinary groups**. Rio de Janeiro, 2016. 168p. PhD Thesis — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Petroleum exploration and production is a complex task where the use of physical models is imperative to minimize exploration risks and maximize the return on the invested capital during the production phase of new oil fields. Over time, these models have become more and more complex, giving rise to a tendency of integration between several simulators and the need for new multiphysics simulations, where single-physics models are solved together in a coupled way. This work presents the GeMA (Geo Modelling Analysis) framework, a library to support the development of new multiphysics simulators, allowing both the coupling of new models built with the framework as a base and the integration with pre-existing simulators. Its objective is to promote the use of software engineering techniques, such as extensibility, reusability, modularity and portability in the construction of engineering physical models, allowing engineers to focus on the physical problem formulation since the framework takes care of data management and other necessary support functions, speeding up code development. Built to aid during the entire multiphysics simulation workflow, the framework architecture supports multiple simulation and coupling paradigms, with special emphasis given to finite element methods. Being capable of representing the spatial domain by multiple discretizations (meshes) and exchanging values between them, the framework also implements some important concepts of extensibility, through the combined use of plugins and abstract interfaces, configurable orchestration and fast prototyping through the use of the Lua language. This work also presents a set of test cases used to assess the framework correctness and expressiveness, with particular emphasis given to a 2D basin model that couples FEM non-linear temperature calculations based on finite elements, mechanical compaction and hydrocarbon maturation and generation.

Keywords

Simulation; Framework; Multiphysics; Multiscale;

Sumário

Lista de figuras

Lista de tabelas

Lista de *scripts*

1
Introdução

Atualmente, simulações físicas são praticamente onipresentes nos mais diversos campos da ciência, sendo empregadas, tanto em âmbito acadêmico quanto industrial, para permitir a compreensão e previsão do comportamento de diversos fenômenos físicos do mundo real aplicados ao cenário em estudo.

Com a evolução das técnicas de simulação e o avanço no poder computacional disponível ao alcance de usuários comuns, as simulações se sofisticaram migrando de cenários inicialmente simples e unidimensionais para cenários complexos bidimensionais e tridimensionais. Simulações que antes modelavam o comportamento de um único fenômeno físico, agora modelam vários fenômenos ao mesmo tempo de maneira acoplada, dando origem às simulações multifísicas. Mais recentemente, problemas envolvendo fenômenos que ocorrem em escalas espaciais e/ou temporais distantes em várias ordens de grandeza, que originalmente só podiam ser resolvidos separadamente, são resolvidos de maneira acoplada nas simulações multiescala.

De acordo com relatório da Fundação Americana de Ciências (National Science Foundation) sobre engenharia baseada em simulação (Oden *et al*, 2006), estas simulações complexas são a chave para se alcançar o progresso em engenharia e ciências. O mesmo relatório coloca ainda que as simulações se tornaram indispensáveis não só para as diversas áreas da engenharia, mas também para a previsão do tempo e de mudanças climáticas.

Ao simular com precisão o comportamento de um novo cenário complexo que se encontra na fronteira dos conhecimentos da física e da engenharia, o entendimento sobre seu comportamento aumenta e, consequentemente, os custos e riscos associados diminuem. Isto melhora a chance de sucesso do projeto, quer este seja uma estrutura a ser construída, um procedimento médico a ser executado ou o lançamento de um foguete.

No âmbito da indústria de petróleo, simulações podem ser encontradas em todas as etapas do negócio, desde a exploração, passando pela produção, refino e distribuição. Apresentar todas as simulações importantes do setor de óleo e gás tornaria esta introdução muito longa, por isto serão citadas apenas algumas. Simuladores de bacias, por exemplo, têm papel importante na etapa de exploração,

ajudando a minimizar riscos exploratórios e a avaliar o potencial de campos ainda inexplorados. Simuladores de reservatório, por sua vez, têm papel fundamental no planejamento e na otimização do desenvolvimento de novos campos de produção. Com o custo de um poço em águas profundas podendo chegar na casa de centenas de milhões de dólares, a diminuição do risco de perfuração de um poço seco ou a minimização do número de poços necessários para o desenvolvimento de um novo campo permitem economia significativa nos custos envolvidos.

Durante a fase de produção, novamente o uso de simulações é importante para prever se os processos de injeção utilizados para estimular a produção são capazes de provocar aumento de tensões que podem comprometer a estabilidade estrutural dos poços, ou mesmo provocar o fraturamento das rochas capeadoras levando a graves acidentes com derramamento de óleo no meio ambiente.

Conforme será detalhado no capítulo 2, a construção de um novo simulador físico é composta por três etapas principais. Na primeira, o cenário a ser simulado é estudado e uma representação física do mesmo é gerada. Esta representação é então mapeada em um conjunto de equações que compõe o modelo matemático da solução, usualmente composto por equações diferenciais, ordinárias ou parciais, que deverão ser posteriormente discretizadas para que sua solução seja possível. Finalmente, em uma terceira etapa, é feita a construção de um sistema de simulação que implementa os modelos definidos nas etapas anteriores, permitindo a obtenção de uma resposta ao problema original.

Claramente, os conhecimentos e habilidades necessárias para definir a representação física apropriada e efetuar a modelagem matemática do problema em estudo são distintos daqueles necessários para transformar este modelo em um sistema de computação capaz de simular o fenômeno estudado. Esta dicotomia fica ainda mais pronunciada quando o objetivo deixa de ser a resolução de um cenário específico e passa a ser a resolução de uma classe de cenários, cuja parametrização é dada pelo usuário, de forma eficiente e robusta.

Profissionais com os conhecimentos combinados necessários de física/engenharia, matemática e ciência da computação são raros. Em universidades e centros de pesquisa, é cada vez mais comum que novos engenheiros sejam treinados em técnicas de simulação e cálculo numérico com uso de ferramentas para avaliação simbólica e/ou numérica, tais como os pacotes Matemática, Maple ou Matlab, em substituição do uso de linguagens de programação de mais baixo-nível, tais como Fortran, C ou C++. Esta escolha justifica-se por manter o foco na física do problema e na prototipação rápida de novos métodos, porém é incompatível com a transformação destes métodos em simuladores eficientes e robustos para uso em cenários de produção.

Idealmente, simuladores deveriam ser desenvolvidos por equipes multidisciplinares, mas é comum que estes sejam implementados por pesquisadores com amplo conhecimento da física do problema, mas poucos conhecimentos de engenharia de software, tornando o sistema resultante mais propenso a erros e de difícil manutenção / extensão, principalmente se este extrapolar o ambiente de pesquisa e vier a ser usado em um ambiente de produção por outros usuários.

Segundo Rogers (1997), um *framework* é uma biblioteca de classes que captura padrões de interação entre objetos, sendo composto por um conjunto de classes concretas e abstratas, explicitamente projetadas para serem usadas em conjunto. Aplicações são desenvolvidas a partir de um *framework* através da implementação de suas classes abstratas.

Frameworks são voltados a um domínio específico de aplicações. Diferentemente de uma biblioteca de componentes, um *framework* define a arquitetura básica da aplicação, ditando sua estrutura global e como classes e objetos colaboram entre si. Em uma inversão de controle, o *framework* define o corpo principal do programa, sendo o responsável por utilizar as classes definidas pelo usuário para especializar suas funções. Desta forma, um *framework* captura as decisões de projeto comuns ao domínio da aplicação, favorecendo o reuso de decisões de projeto e não apenas o reuso de código (Gamma *et al*, 1994).

Conhecimentos de física e engenharia de *software* são mundos separados. Entretanto, a construção de um framework voltado ao desenvolvimento de simulações multifísicas cria uma "ponte" entre estes dois mundos, permitindo que o engenheiro possa focar-se na construção da representação física e do modelo matemático, sendo suportado e guiado a seguir as melhores práticas de programação pelo *framework*. Como consequência, novos simuladores podem ser criados com maior agilidade, robustez e qualidade compatíveis com ambientes de produção.

Embora todo *framework* seja voltado a um domínio de aplicações específico, seu foco pode ser mais ou menos amplo. Dentro do domínio de simulações físicas, um *framework* pode ser voltado exclusivamente a um método de discretização (ver capítulo 2) e/ou a um único tipo de problema físico, enquanto outro pode ser mais geral, suportando diversos métodos e diversos tipos de problemas.

O *framework* GeMA (Geo Modelling Analysis framework), desenvolvido em C++, tem como objetivo ser um ambiente geral de suporte para simulações multifísicas. Cabe notar porém que *frameworks* podem ser hierárquicos, assim um *framework* voltado, por exemplo, para a construção de simulações baseadas no método de elementos finitos pode, por sua vez, basear-se em um outro, de suporte a simulações físicas em geral.

1.1 Características desejáveis em um *framework* de multifísica

Na definição das principais características desejáveis para um *framework* geral de suporte à simulações multifísicas, é interessante levar em consideração as conclusões apresentadas em Post e Kendall (2004). Esse trabalho apresenta lições de projeto aprendidas ao longo de 8 anos de desenvolvimento de simuladores multifísicos voltados a simulações relativas ao armamento nuclear americano pelos laboratórios nacionais Lawrence Livermore, Los Alamos e Sandia.

Uma de suas conclusões diz: “Uma melhor física é muito mais importante do que uma melhor ciência da computação". No âmbito do desenvolvimento de um *framework*, o reconhecimento desta afirmação é de vital importância. Para os usuários do simulador final desenvolvido com seu auxílio, o importante é a acurácia da física simulada e não como o mesmo foi construído, ou seja, todos os benefícios que um *framework* traz ao desenvolvimento de um simulador só serão relevantes se o mesmo efetivamente cumprir seu papel. De nada adianta que o simulador tenha sido construído rapidamente, seja eficiente, extensível e utilize as melhores práticas da engenharia de *software* se o resultado da simulação for aquém do esperado.

Como consequência, um *framework* geral para simulações multifísicas deve abraçar seu papel de “ator coadjuvante” e exercê-lo tomando como princípio básico que o usuário do *framework* deve possuir toda a liberdade possível para definir como as físicas do problema serão simuladas. Desta forma, as primeiras características desejáveis são:

1. Suporte a múltiplos métodos de discretização das equações do modelo matemático, como por exemplo, os métodos de elementos finitos, volumes finitos, etc. Mesmo que o método desejado não esteja disponível diretamente através do suporte padrão oferecido pelo *framework*, este deve poder ser estendido para suportá-lo.

2. Suporte a simulações multifísicas, permitindo que o método de acoplamento utilizado (ver capítulo 2) para solução do problema seja definido pelo método de solução, e não pelo *framework*.

3. Suporte à orquestração configurável. Em uma visão macro, a execução de uma simulação multifísica complexa pode ser decomposta na execução de uma série de processos que cooperam para a obtenção do resultado final desejado. A esta execução coordenada é dado o nome de orquestração.

   O conjunto de passos necessários para a solução de cada problema multifísico é distinto. Assim, a definição dos passos a serem executados deve fazer parte do método de solução, sendo configurados como parte da descrição de como o problema multifísico será simulado.

Frequentemente, iniciativas de simulações multifísicas foram precedidas pela criação de simulações individuais das principais físicas envolvida no problema. Desta forma, é comum que as organizações envolvidas já possuam simuladores, comerciais ou construídos internamente, que resolvem partes do novo problema multifísico. Quando este pode ser resolvido de maneira fracamente acoplada (ver capítulo 2), há um apelo muito grande para a reutilização, ao menos parcial, dos simuladores existentes, preservando-se o investimento feito no passado. Assim, a quarta característica desejável para o *framework* consiste em:

4. Suporte à integração com simuladores pré-existentes.

Em modelos multiescala, dados e propriedades calculadas em uma escala refinada precisam ser continuamente transferidos de/para uma escala mais grosseira. Diversos modelos matemáticos existem para efetuar a ligação entre as escalas (Fish, 2006). Não cabe a um *framework* geral a implementação de métodos específicos, porém é interessante que este implemente uma ferramenta para transferência de dados entre malhas. Assim, a quinta característica desejável para o *framework* consiste em:

5. Suporte básico à simulações multiescala através da possibilidade de transferência de dados entre discretizações espaciais do domínio.

Para a solução de modelos massivos, onde a discretização espacial do problema conta com milhões de elementos, é importante que este possa ser resolvido em paralelo. Desta forma, a última das principais características desejáveis consiste em:

6. Suporte à paralelização da simulação em ambientes com memória compartilhada e/ou distribuição da simulação por conjuntos de máquinas.

1.2 O *framework* GeMA

O desenvolvimento do *framework* GeMA está inserido em um projeto maior de P&D, iniciado em 2015 pelo instituto Tecgraf/PUC-Rio. Dentre os objetivos deste projeto, incluem-se:

– Desenvolvimento de um protótipo de simulador numérico para problemas acoplados com suporte a interações hidro-termo-mecânica-químicas (THMC) para problemas de petróleo e gás, incluindo modelagem multifísica e multifase na capacidade de simulação, envolvendo escalas de tempo e de espaço alargadas.

– Modelagem grão-fluido acoplada: Simulação da fragmentação da rocha durante a produção (produção de sólidos).
– Modelagem da construção de poços: Simulação do processo de perfuração, garantindo a integridade do poço em reservatórios complexos.
– Modelagem de estimulação hidráulica: Otimização da produção / recuperação de hidrocarbonetos.

Dentro deste contexto, o *framework* GeMA compõe a base de suporte ao desenvolvimento do simulador THMC, que por sua vez suporta os demais tópicos de modelagem.

Previamente à decisão de criação de um novo *framework*, algumas das opções existentes foram estudadas, conforme apresentado no capítulo 3. Embora existam trabalhos maduros, com características sofisticadas, nenhum dos *frameworks* / simuladores estudados apresenta todas as características listadas na seção 1.1. As principais deficiências encontradas incluem:

– Escopo voltado para solução de problemas específicos;
– Suporte a apenas um ou a um conjunto limitado de métodos de discretização;
– Falta de suporte à integração com simuladores pré-existentes;
– Disponibilidade limitada ou licença permitindo seu uso apenas em aplicações acadêmicas, sem suporte a aplicações comerciais;

Excetuando-se o requisito de suporte à distribuição e paralelização da execução, que será abordado em trabalhos futuros, o *framework* GeMA tem como objetivo atender aos demais requisitos listados na seção 1.1.

Quando uma simulação é executada no ambiente GeMA, o primeiro passo consiste em carregar sua definição contendo os dados do modelo e a descrição do método de solução adotado. Estes definem, respectivamente, o que será simulado e como a simulação será feita. A seguir, um *script* de orquestração é executado para cálculo dos resultados (Figura 1.1).

O *script* de orquestração é o objeto central do método de solução. Sua função principal é permitir ao usuário descrever a sequência de processos que devem ser aplicados ao modelo para que os resultados desejados sejam corretamente calculados, fornecendo assim o *loop* principal da simulação. Este *script* é escrito pelo usuário através da utilização da linguagem Lua (Ierusalimschy *et al*, 1996).

A linguagem Lua é uma linguagem interpretada especialmente construída para ser incorporada em aplicações, permitindo assim que estas executem dinamicamente programas fornecidos pelos usuários, no caso em tela o próprio *script* de orquestração. Conforme Ierusalimschy (2003), a linguagem é considerada

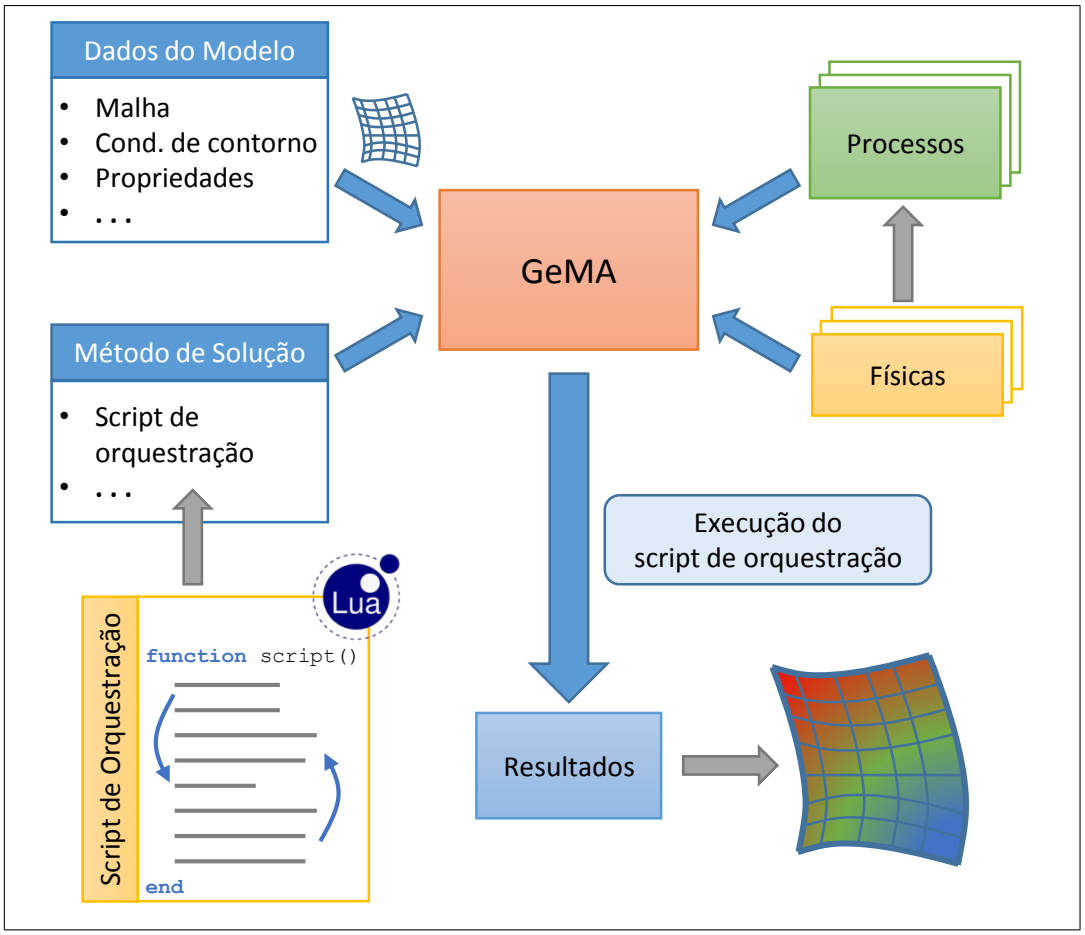


Figura 1.1: Passos principais de uma simulação GeMA.

uma linguagem simples, de fácil aprendizado, porém englobando conceitos poderosos tais como tipagem dinâmica, coleta de lixo, funções como objetos de primeira classe e uso de mapas associativos para construção de estruturas de dados. É uma linguagem extensível que permite a criação de "dialetos" voltados ao domínio da aplicação (*domain-specific languages*). Muito usada como uma linguagem para integração e extensão de aplicações, também é considerada como uma das linguagens interpretadas mais rápidas existentes.

Processos são a unidade básica utilizada para a descrição da solução, podendo ser escritos em C++ ou na própria linguagem Lua. Em geral são primitivas de alto nível que descrevem um ação completa, tais como a execução de uma análise através do método de elementos finitos, a transferência de dados de uma malha para outra, o refinamento adaptativo de uma malha, o salvamento de um conjunto de dados, etc.

Ao adotar uma linguagem de programação completa, com controle de fluxo, funções, suporte a estruturas de dados, etc, ao invés de uma solução simplificada que apenas identifique uma ordem de execução de processos, o *framework* permite que seu orquestrador tenha a liberdade de executar operações tão complexas quanto necessárias.

Para o *framework*, processos são interfaces abstratas cuja implementação concreta é dada por *plugins*. Malhas e outras entidades importantes são tratadas da mesma maneira. O uso de interfaces abstratas para modelar as principais entidades do *framework* promove sua extensibilidade. O uso de *plugins*, além de outras vantagens, força a existência de uma clara separação de conceitos e interdependências, garantindo a modularização do código.

A implementação atual do *framework* inclui processos para suporte à soluções baseadas no método de elementos finitos. Estes processos, por sua vez, definem interfaces denominadas de físicas, responsáveis por fornecer ao método a formulação matemática específica para um conjunto de equações. Outros métodos de discretização são suportados através da criação de novos processos.

Desta forma, a combinação da extensibilidade através de interfaces abstratas e *plugins* com a flexibilidade introduzida pelo *script* de orquestração permite que o modelador tenha toda a liberdade necessária para definir como a simulação será executada.

A possibilidade de implementar processos e físicas através de código em Lua permite que o *framework* seja usado para a prototipação rápida de novas ideias que, uma vez testadas e validadas, podem ser transformadas em código C++ para maior eficiência, se necessário.

A integração com simuladores pré-existentes pode ser feita em dois níveis distintos:

1. Integração através de trocas de arquivos. Sempre que for necessário executar um simulador pré-existente, um novo arquivo de dados é escrito em formato compreendido pela aplicação externa, esta é executada e seus resultados são lidos, interpretados e incluídos na estrutura da simulação.

2. Integração a nível de componentes. Neste cenário, o simulador externo é visto como um componente de *software*, a ser integrado ao ambiente GeMA através de uma camada de adaptação que permita a troca de dados entre ambos.

No cenário de trocas de arquivos, geralmente a integração pode ser alcançada através da expressividade natural da linguagem Lua durante a orquestração da solução, devidamente apoiada por processos para leitura e escrita de dados. Naturalmente este processo tem uma tendência a ser ineficiente, mas dependendo do cenário pode ser a única forma de integração possível (simuladores comerciais, por exemplo). Requer que os formatos de dados utilizados pelo simulador externo sejam conhecidos e que o mesmo possa ser executado de forma independente de uma interface com o usuário (algo geralmente suportado pela maioria dos simuladores).

O suporte à integração via componentes é dado pelas interfaces abstratas do *framework*. Desta forma, a camada de adaptação consiste em uma implementação

da interface esperada pelo *framework* GeMA com base nas estruturas internas do simulador que está sendo integrado. Para que este tipo de integração possa ocorrer, é necessário que exista acesso ao código fonte do simulador externo ou que este forneça uma API de programação. Sua adoção, quando possível, requer mais esforço do que uma integração via arquivos, mas geralmente traz resultados mais eficientes, sendo uma alternativa atraente para integração com outros simuladores de uma mesma instituição.

Por diversos motivos, modelos multifísicos podem ser compostos por mais de uma discretização do domínio espacial. Pode-se ter situações onde uma física requer um refinamento maior do que as outras. Em outros casos, o domínio de cálculo de cada física é distinto, com sobreposição parcial entre os domínios ou mesmo com interação entre estes apenas em suas bordas. Pode-se ainda estar trabalhando em um modelo que integra simuladores externos que operam com tipos de malhas incompatíveis (um deles trabalha, por exemplo, com malhas triangulares e o outro, com malhas de quadriláteros).

Para suportar estas situações, onde há necessidade de trabalho com múltiplas malhas, em múltiplas escalas, incluindo tipos heterogêneos de elementos e possivelmente representando partições distintas do domínio espacial, o *framework* GeMA permite que o modelo de simulação contenha um conjunto de malhas e implementa processos para transferência de dados entre estas, apoiados por estruturas de dados para indexação espacial.

Se for considerada exclusivamente a definição de *framework* dada anteriormente, os usuários do *framework* GeMA são aqueles programadores que usam diretamente as classes definidas por este para criarem novos simuladores. O *framework* GeMA porém, possui implementações padrão para todas as suas interfaces abstratas (na forma de *plugins*) e inclui uma aplicação para execução de simulações. Considerando este ambiente expandido, denominado de ambiente GeMA, é interessante analisar os casos de uso esperados e os tipos de usuários atendidos.

O suporte à criação de simulações multifísicas pode ocorrer em diversos níveis, atendendo a cenários e tipos distintos de problemas:

1. Configuração de uma simulação pré-existente para sua execução sobre um novo conjunto de dados do modelo;

2. Criação de uma nova simulação onde o método de solução tem como base métodos de cálculo pré-existentes, quer seja baseados em processos e físicas criadas com suporte do próprio *framework* ou em simuladores externos a serem integrados;

3. Criação de novos métodos de cálculo que serão implementados através de novos processos e/ou novas físicas;

Na solução de problemas reais, usualmente todos os papeis acima estão presentes, possivelmente associados a diversos usuários, de perfis multidisciplinares, conforme apresentado na figura 1.2.

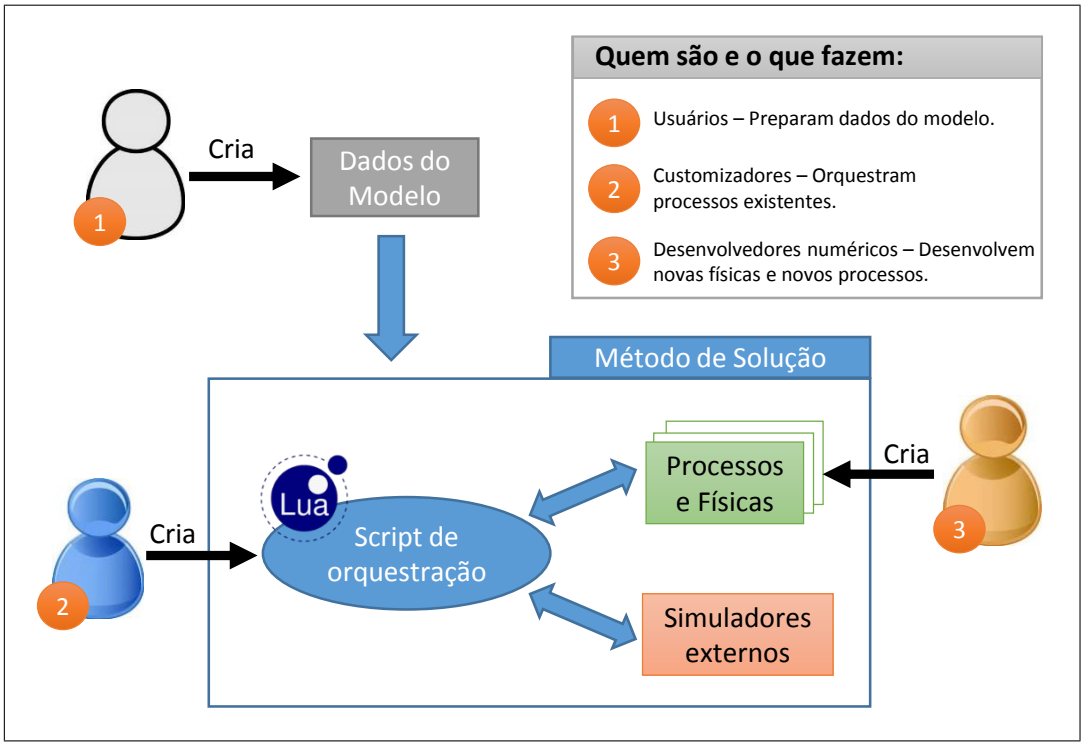


Figura 1.2: Níveis de suporte à criação de simulações.

O primeiro papel, configuração de modelos já existentes, geralmente está associado com o usuário que irá executar e interpretar as informações obtidas como resultado do modelo. Este usuário, conhecedor do domínio da aplicação, não precisa conhecer a fundo o modelo matemático da simulação, apenas os tipos de dados de entrada necessários e suas inter-relações. Sua responsabilidade consiste em alimentar o modelo com os dados corretos (malhas, atributos, condições de contorno, etc) e interpretar os resultados obtidos.

No segundo papel, o usuário, conhecedor do modelo matemático da simulação, descreve através do *script* de orquestração como os processos existentes serão compostos, contribuindo em conjunto para o resultado final da simulação. Este tipo de usuário não precisa ter grandes conhecimentos de desenvolvimento de sistemas. Habilidades básicas são suficientes para a construção do *script* de orquestração.

Finalmente, o terceiro cenário de uso contempla a necessidade de criação de processos e físicas para suporte a novos métodos de discretização e/ou novas equações para os métodos existentes. Este é o usuário "tradicional" de um

framework, com conhecimentos tanto do modelo matemático a ser implementado quanto de desenvolvimento de sistemas, que utiliza as interfaces e classes de suporte fornecidas para estender o *framework* e criar novos tipos de simuladores.

1.3 Avaliação do *framework*

Para avaliar a corretude e a expressividade do *framework*, foram implementadas diversas simulações de teste. Entretanto, é importante salientar que o foco do presente trabalho consiste na definição da arquitetura e na implementação básica do ambiente GeMA, e não no resultado das simulações *per si* ou em sua relevância individual.

Testes básicos foram efetuados através de simulações utilizando o método de elementos finitos para cálculo de tensões e de temperatura. Foram efetuadas simulações com tensões lineares e não-lineares para barras e para elementos em estado plano de tensões. Simulações com temperatura foram baseadas na condução de calor, em regimes permanente e transiente, com diversos tipos de condições de contorno e a possibilidade de uso de funções do usuário para definir parâmetros de condutividade dependentes da própria temperatura, tornando o problema não-linear. Mudanças de fase foram exploradas através do método de capacidade térmica efetiva (Lewis *et al*, 2004). Os resultados obtidos foram comparados com modelos analíticos e/ou com resultados encontrados na literatura.

Modelos multifísicos foram testados através do acoplamento do cálculo de tensões com temperatura e através de um cenário mais complexo composto por um modelo 2D para modelagem de bacias sedimentares. Este modelo engloba fenômenos de sedimentação com compactação mecânica, história térmica, maturação e geração de hidrocarbonetos. A evolução da bacia no tempo requer a utilização de uma malha dinâmica para acompanhar a deposição das camadas sedimentares e eventuais intrusões ígneas, cuja presença leva à necessidade de utilização de passos adaptativos de tempo na simulação. Cálculos de temperatura foram efetuados com base no processo de elementos finitos implementado em C++. Os demais cálculos, inclusive a evolução da malha no tempo, foram implementados em Lua para avaliação da potencialidade do ambiente.

A integração com aplicações externas e os processos de transferência de dados entre malhas foram testados através da criação de um modelo de simulação para acoplar cálculos de poro-pressões com cálculos de tensões, utilizando malhas distintas para cada física, com domínios e escalas diferentes, e simuladores externos.

Finalmente, a possibilidade de prototipação rápida foi avaliada através da comparação entre a física para cálculo de temperaturas pelo método de elementos finitos implementada em C++ com uma versão em Lua da mesma.

1.4 Contribuições

O *framework* GeMA apresenta uma arquitetura baseada no uso combinado de interfaces abstratas com *plugins* e na adoção da linguagem de extensão Lua como parte de seu núcleo, permeando todas as etapas de definição e execução das simulações. Esta arquitetura é inovadora e seus principais benefícios serão abordados ao longo deste trabalho.

Como resultado da arquitetura proposta, o *framework* GeMA apresenta uma série de características que, em conjunto, o distinguem dos demais trabalhos existentes:

- Suporte a processos e físicas adicionados pelo usuário, incluindo a possibilidade de inclusão de novos métodos de discretização do modelo matemático;
- Orquestração configurável baseada em uma linguagem de *script* completa, flexível e eficiente;
- Suporte à integração com simuladores pré-existentes;
- Suporte à prototipação rápida de novas físicas;
- Suporte a múltiplos tipos de usuários.

1.5 Organização da tese

O restante desta tese está organizado em seis capítulos adicionais. O capítulo 2 apresenta uma breve introdução a simulações físicas. Seu objetivo principal consiste em estabelecer a terminologia utilizada nos demais capítulos. No capítulo 3 são apresentados alguns trabalhos relacionados selecionados que tiveram influências no desenvolvimento do *framework* GeMA.

O capítulo 4 apresenta a arquitetura global do *framework*, suas principais entidades e pontos de extensão, detalhando os mecanismos adotados para o atendimento aos principais requisitos apontados na seção 1.1. São apresentados também alguns pontos importantes de sua implementação.

No capítulo 5 são apresentados os modelos matemáticos para as simulações utilizadas na avaliação do *framework*. Os cenários de teste, os resultados obtidos e sua discussão são apresentados no capítulo 6. Finalmente, o capítulo 7 apresenta uma discussão das conclusões obtidas e de possíveis trabalhos futuros.

2
Simulações físicas

Uma simulação é definida por Banks (1999) como uma "imitação da operação de um processo ou sistema do mundo real no tempo". Uma simulação física pode ser entendida então como uma simulação, que tem por base leis e princípios físicos.

Simulações são importantes pois permitem tentar entender o passado e prever o futuro, respondendo a perguntas do tipo "e se" aplicadas ao modelo de simulação, perguntas estas que muitas vezes não podem ser respondidas na prática devido ao custo associado ou mesmo à impossibilidade, já que o objeto da simulação pode nem existir na realidade.

Simulações baseiam-se em modelos. Segundo Felippa (2004), um modelo é "um dispositivo simbólico construído para simular e prever *aspectos* do comportamento de um sistema". Como "todos os modelos estão errados, mas alguns são úteis" (Box, 1979), é importante notar a ênfase em aspectos do comportamento, já que a descrição do comportamento completo de um sistema só é possível através do próprio sistema real.

Segundo Peiró e Sherwin (2005), um modelo computacional é composto por três etapas: a definição do problema, sua modelagem matemática e por final a simulação em um computador. Já para Felippa (2004), o processo de simulação é composto por um sistema físico, um modelo matemático, um modelo discreto e uma solução discreta. A visão do processo de simulação adotado no presente trabalho é inspirada nestas duas definições e ilustrada na figura 2.1(a).

O primeiro passo do processo de simulação consiste na idealização do fenômeno do mundo real a ser estudado, gerando uma representação física do mesmo. Isto envolve a definição das grandezas e propriedades físicas (escalares, vetoriais ou tensoriais) envolvidas, bem como seus domínios espacial e temporal, ou seja, a região do espaço e em quais momentos estas grandezas serão estudadas. Também é importante que esta idealização represente um problema bem posto.

A figura 2.1(b) ilustra a representação física de uma placa de metal submetida a diferentes temperaturas em suas bordas. Neste problema deseja-se estudar a temperatura em cada ponto da placa, em equilíbrio térmico. O domínio espacial do problema é representado pela própria placa.

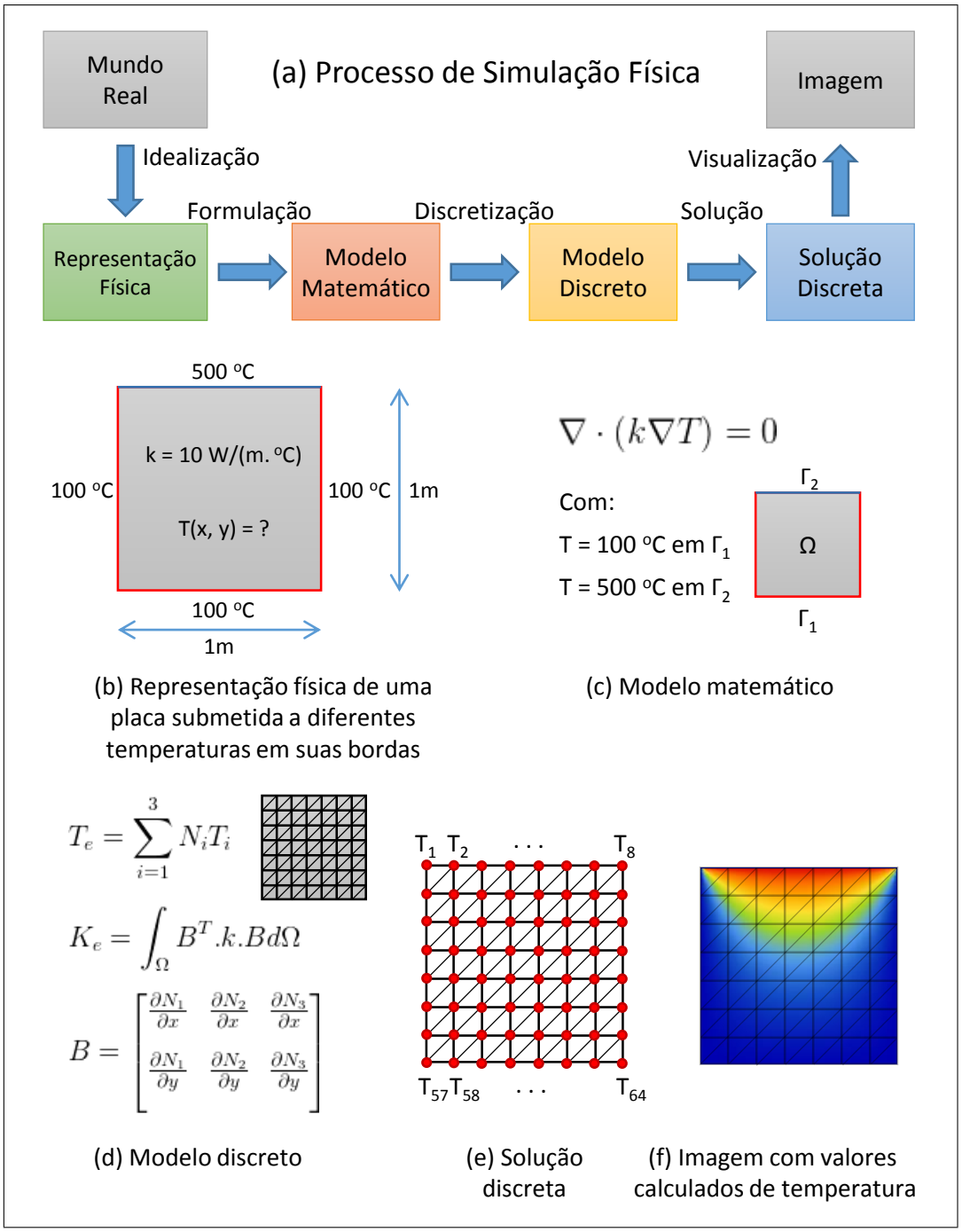


Figura 2.1: Processo de simulação física. (a) Etapas do processo de simulação; (b) Exemplo de representação física; (c) Modelo matemático; (d) Modelo discreto; (e) Solução discreta; (f) Imagem com valores calculados.

O segundo passo consiste em formular um modelo matemático para a idealização do mundo real obtida no passo anterior. Nesta etapa, o modelador, utilizando as leis e princípios físicos que regem as grandezas do problema em estudo, define o conjunto de equações e condições de contorno do modelo. Exemplos incluem as equações de elasticidade que relacionam deformações em objetos sólidos com as forças aplicadas, equações de Navier-Stokes para previsão do movimento de fluidos, equações de Maxwell para problemas de eletromagnetismo,

equações para transferência de calor, equações para tratamento de reações químicas, etc. Em problemas multifísicos, ou seja, problemas envolvendo mais de uma grandeza física, diversos conjuntos de equações são necessários, geralmente com acoplamento entre as grandezas.

No exemplo da figura 2.1(c), o problema em estudo é formulado através da equação bidimensional de condução de calor, uma expressão do princípio de conservação de energia, e das condições de contorno impostas na fronteira do domínio.

Em geral, os modelos matemáticos referentes aos problemas que se quer simular são complexos, envolvendo sistemas acoplados de equações diferenciais parciais, no tempo e no espaço, para os quais não há soluções analíticas conhecidas, ou estas existem apenas para geometrias regulares com condições de contorno simples.

Nos casos onde não é possível encontrar uma solução analítica que forneça os valores das grandezas em estudo para todos os infinitos pontos do domínio, faz-se necessária a adoção de soluções numéricas que envolvem, normalmente, a criação de um modelo discreto para que o problema possa ser resolvido.

Em um modelo discreto, as grandezas em estudo, também chamadas de graus de liberdade, são aproximadas por um conjunto de valores denominados de solução discreta. Considere uma função $u(x)$, onde x representa uma posição no espaço, e um conjunto de pontos $x_i; i = 1, \cdots, N$ pertencentes ao domínio. A solução discreta procurada será um conjunto de valores $\{u_1, \ldots, u_N\}$ tal que $u_i \approx u(x_i)$.

O processo de discretização do modelo matemático é composto por duas etapas principais. Na primeira, as equações contínuas do modelo matemático são transformadas em equações discretas, isto é, equações que têm como objetivo permitir o cálculo dos valores de u_i. Este passo é denominado método de discretização. Um breve resumo de alguns dos métodos existentes pode ser encontrado na seção 2.1. Cabe salientar que para simulações que evoluem no tempo, o método de discretização deve efetuar tanto a discretização espacial quanto temporal das equações.

A segunda etapa consiste na discretização do domínio espacial, correspondendo à definição dos pontos x_i onde a solução aproximada será buscada. A este processo é dado o nome de discretização espacial, e a seu resultado, domínio discreto. Quando o domínio discreto inclui, além dos pontos x_i, relações de vizinhança entre estes pontos (arestas, faces, etc.), denomina-se o mesmo de malha. A figura 2.2 ilustra estas etapas.

A figura 2.1(d) ilustra o modelo discreto para o problema em estudo, composto tanto pela malha associada à placa quanto pelas principais equações discretas, que neste caso foram obtidas através do método dos elementos finitos.

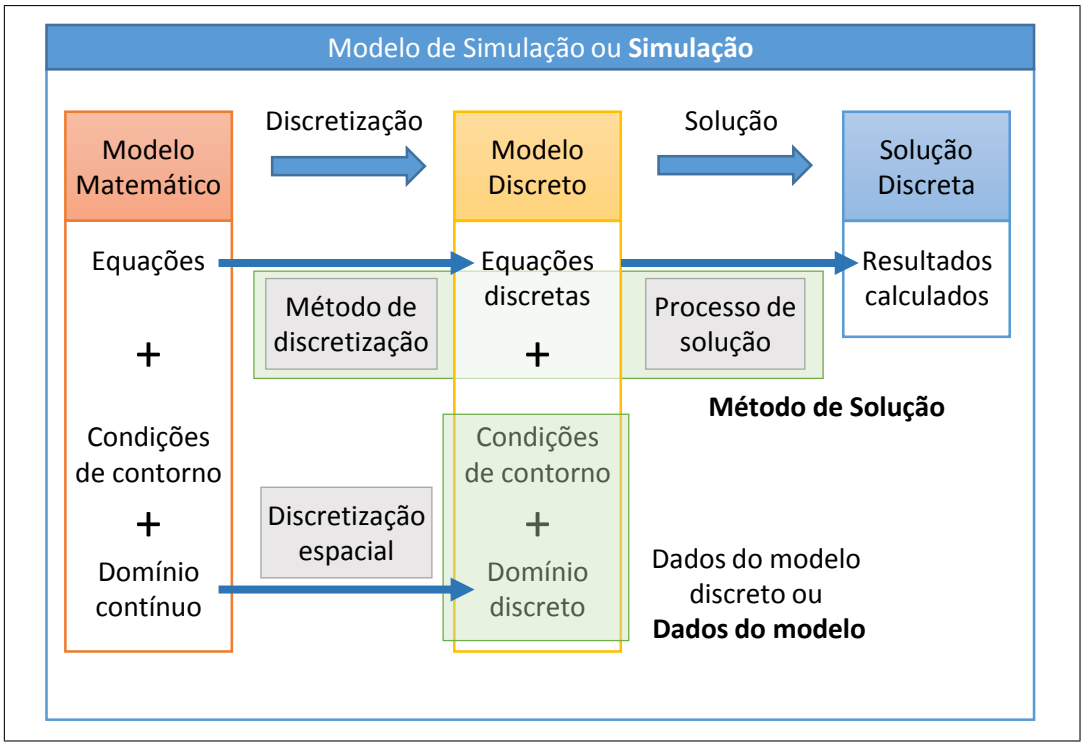


Figura 2.2: Modelo de simulação.

Com o modelo discreto criado, é necessário resolver o novo conjunto de equações para obtenção da solução discreta. Normalmente, isto envolve a resolução de um sistema linear de equações já que em situações onde o modelo é não-linear, são usados métodos de linearização, tais como o método de Newton-Raphson. Na figura 2.2, esta etapa é denominada de processo de solução e está intimamente ligada ao método de discretização utilizado. Ao conjunto composto pelo método de discretização e pelo processo de solução é dado o nome de método de solução. A figura 2.1(e) ilustra a solução discreta obtida.

O conjunto formado pelo modelo matemático, pelo modelo discreto, pela solução discreta e pelos processos de transição entre estes, é denominado modelo de simulação ou simplesmente simulação. Ao conjunto formado pelo domínio discreto, condições de contorno e outros dados adicionais associados ao domínio discreto (não apresentados na figura 2.2), dá-se o nome de dados do modelo discreto, ou simplesmente dados do modelo, uma vez que estas são as principais informações que devem ser fornecidas para execução de uma simulação sobre um novo conjunto de dados.

Finalmente, com base na solução discreta, o processo de visualização permite que o grande volume de dados gerados pela simulação seja transformado em imagens que facilitem sua interpretação. Normalmente este papel é delegado a sistemas de pós-processamento independentes do simulador. A partir da descrição da geometria do problema e dos valores calculados pela simulação, são apresentados ao usuário modelos gráfico-interativos, 1D, 2D ou 3D.

Existem diversas técnicas para visualização de campos escalares, vetoriais e tensoriais (Gattass *et al*, 1991). O estudo destas técnicas é o alvo das linhas de pesquisa em visualização científica. Algumas das possibilidades existentes incluem imagens com gradações de cores, traçados de isolinhas e isosuperfícies, traçados de linhas de fluxo, etc. A figura 2.1(f) apresenta uma imagem representando o campo de temperaturas calculado na simulação em estudo.

2.1 Métodos de discretização

Há diversos métodos de discretização propostos na literatura para a solução de equações diferenciais parciais. Os três métodos tradicionais mais utilizados são os métodos de diferenças finitas (FDM), elementos finitos (FEM) e volumes finitos (FVM). Uma breve comparação entre estes métodos pode ser encontrada em Peiró e Sherwin (2005).

Destes, o mais antigo é o método de diferenças finitas. Neste método, o domínio espacial do problema é discretizado através de uma grade regular. Em cada ponto da grade, as derivadas parciais são substituídas por expansões locais em série de Taylor. A figura 2.3 apresenta os três principais esquemas de diferenças finitas existentes: diferenças para frente (*forward differences*), diferenças para trás (*backward differences*) e diferenças centrais (*central differences*).

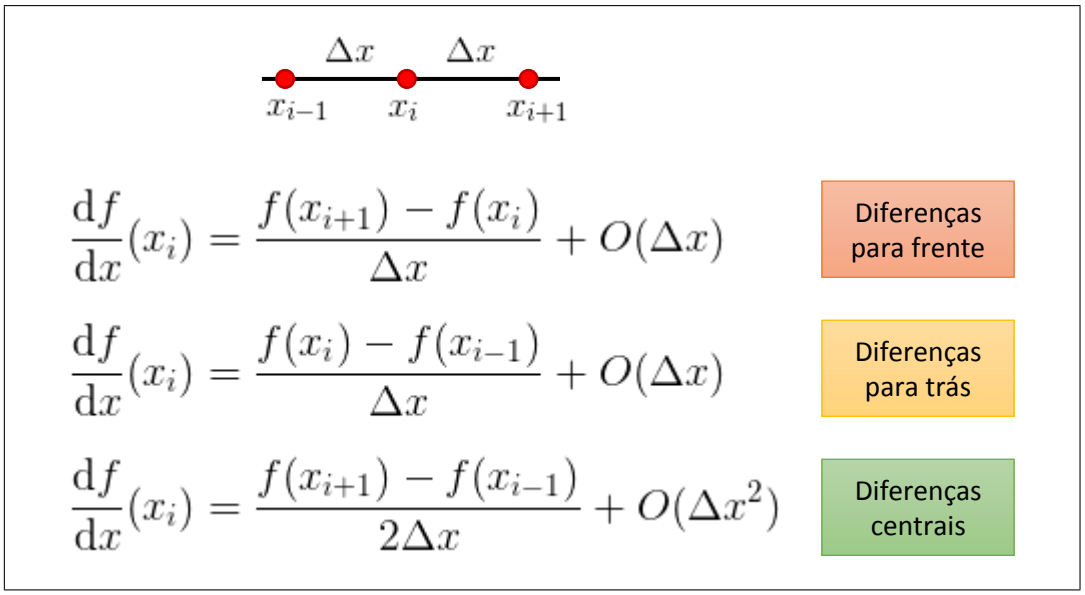


Figura 2.3: Diferenças finitas.

Por trabalhar com grades regulares, este método apresenta limitações quando a simulação envolve geometrias complexas. Embora existam variantes do método que permitem sua aplicação a grades irregulares, seu uso não é largamente difundido.

Enquanto o método de diferenças finitas baseia-se na forma diferencial das equações, os demais métodos baseiam-se em sua forma integral fraca. O método de elementos finitos é o principal método de discretização utilizado em problemas de mecânica estrutural, enquanto o método de volumes finitos é muito utilizado em aplicações de dinâmica dos fluidos.

No método de elementos finitos, o domínio espacial do problema deve ser subdividido em pequenas regiões (partições do domínio) denominadas de elementos, onde o comportamento das grandezas em estudo possa ser aproximado por um polinômio de grau baixo. Estes polinômios interpolantes, conhecidos como funções de forma ou *shape functions*, são descritos em função dos valores das grandezas nos vértices (nós) do elemento (Gattass *et al*, 1991).

As equações discretas que definem como os valores nos nós são calculados podem ser obtidas através de técnicas de cálculo variacional ou através de métodos de minimização baseados em resíduos ponderados, como o método de Galerkin. As integrais utilizadas na definição destas equações podem ser avaliadas localmente em cada elemento, sendo o conjunto global de equações discretas composto por somas das equações locais, em um processo denominado de *assembly*. Esta é uma das características fundamentais do método.

Maiores detalhes sobre o método de elementos finitos podem ser obtidas em Felippa (2004); Zienkiewicz *et al* (2005); Lewis *et al* (2004).

O método de volumes finitos é voltado para o tratamento de problemas conservativos. Sua base consiste em decompor o domínio em volumes de controle e efetuar um balanço da propriedade conservada em cada volume. As integrais resultantes são então aproximadas gerando o modelo discreto. Uma introdução ao método pode ser encontrada em Schäfer (2006).

Recentemente, novos métodos de discretização, denominados de *meshfree* ou *meshless*, vêm sendo bastante estudados. Nestes métodos, a discretização espacial é feita através de uma nuvem de pontos que cobre o domínio da simulação, mas sem conexões pré-estabelecidas entre os nós. Estes métodos são formados por dois passos principais: a definição de funções de aproximação e suas derivadas através de métodos de mínimos quadrados, funções de *kernel* ou funções de base radial, e a discretização das equações contínuas com base nestas funções. Uma introdução aos principais métodos existentes pode ser encontrada em Li *et al* (2005).

Para a discretização temporal das equações diferenciais parciais, é comum a utilização do método de diferenças finitas, mesmo que a discretização espacial seja feita através de outro método. A escolha do esquema de diferenças finitas utilizado (para frente, para trás ou centrais) dá origem a métodos implícitos ou explícitos.

Nos métodos explícitos, o estado do sistema no tempo $t + 1$ é calculado diretamente a partir do estado no tempo t. Estes métodos são computacionalmente

eficientes mas dependem do uso de passos pequenos no tempo para serem estáveis. Já nos métodos implícitos, é necessário resolver um sistema de equações para obter o novo estado do sistema. Sua vantagem consiste em que estes métodos podem ser estáveis mesmo para passos de tempo grandes. Mais detalhes sobre a discretização temporal podem ser obtidos em Zienkiewicz *et al* (2005); Lewis *et al* (2004).

2.2 Simulações multifísicas

Simulações multifísicas são simulações que envolvem o cálculo de diversas grandezas físicas. Seu modelo matemático geralmente é formado por um conjunto de equações diferenciais parciais acopladas, ou seja, o cálculo de uma grandeza depende do valor das demais.

Análises multifísicas podem ser exemplificadas pela interação entre um fluido e uma estrutura. O fluxo do fluido exerce pressão sobre a estrutura, causando sua deformação, o que por sua vez perturba o fluxo original (Lethbridge, 2005). Efeitos térmicos acoplados a outros fenômenos físicos também são exemplos bastante comuns, tais como o efeito de dilatações e alterações nas propriedades físicas dos materiais em cálculos de tensões, ou a geração de calor pelo fluxo de corrente elétrica em um microprocessador, com consequente necessidade de dissipação do mesmo.

Tradicionalmente, existem duas formas distintas para a resolução de problemas multifísicos. Na primeira estratégia, denominada de fracamente acoplada, acoplamento sequencial ou separação de operadores (*operator splitting*), cada fenômeno físico é resolvido de maneira independente. Durante a solução de cada física, suas dependências em relação às demais são mantidas constantes. De posse de uma solução, a próxima física é executada tendo como entrada o resultado da anterior. Após a execução sequencial de todas as físicas do problema, o processo é repetido até que a convergência seja alcançada.

A figura 2.4 ilustra este processo para um sistema composto por três físicas acopladas que calculam, respectivamente, as grandezas u, v e w. O laço mais externo corresponde à iteração temporal. O segundo é o laço de acoplamento entre as físicas. A cada iteração do laço de acoplamento, as físicas 1, 2 e 3 são executadas sequencialmente. A execução da primeira física no passo i tem como base o estado da simulação dado pelos valores de u, v e w no passo anterior $i-1$. Após sua execução, o valor de u é atualizado, assim a execução da física 2 baseia-se no estado dado pelo valor de u no passo i e pelos valores de v e w no passo $i-1$. O mesmo ocorre para a física 3 que opera sobre o estado dado pelos valores de u e v no passo i e pelo valor de w do passo anterior $i-1$. Este laço é executado até que a diferença entre os valores de u, v e w entre dois passos consecutivos seja menor que uma

tolerância. Cabe observar que o cálculo de cada física individual pode incluir laços próprios para tratamento de não-linearidades internas a cada física.

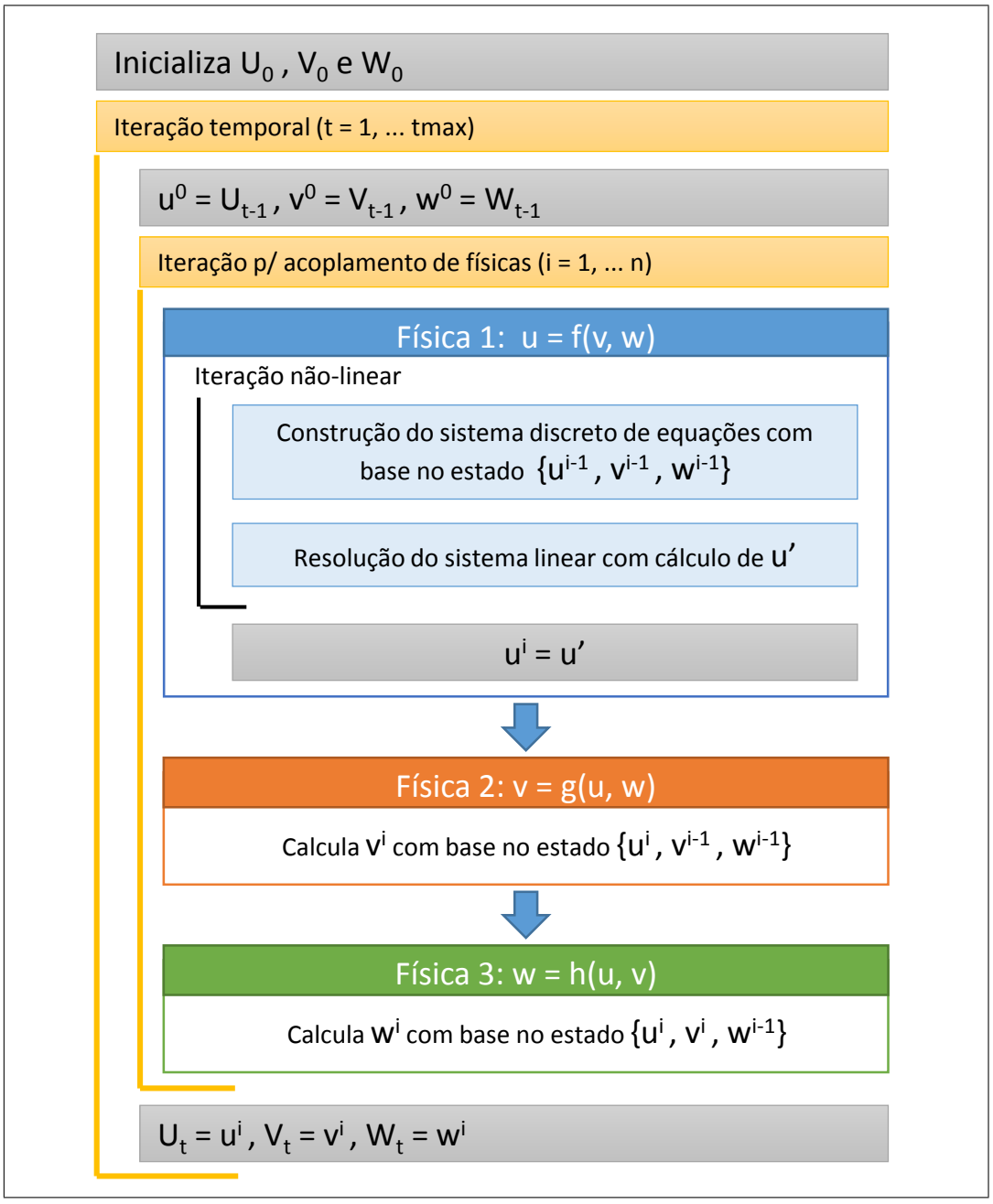


Figura 2.4: Resolução de um sistema multifísico com acoplamento fraco.

Na segunda estratégia, denominada de acoplamento direto ou fortemente acoplada, o conjunto completo de equações discretizadas é resolvido de maneira simultânea através da criação de um único sistema global de equações, materializado em uma grande matriz contendo todos os graus de liberdade do sistema acoplado.

O método fracamente acoplado funciona muito bem quando não há um acoplamento bidirecional forte entre as físicas. Nestes casos, a convergência pode ser alcançada de maneira rápida com poucas iterações do laço de acoplamento

(Novascone *et al*, 2013). Por outro lado, quando este acoplamento bidirecional é forte, a convergência pode ser muito lenta, ou mesmo inalcançável, inviabilizando o método. Nestes casos a solução deve ser feita através do método de acoplamento forte.

Quando possível, a utilização do método fracamente acoplado incentiva à modularização e ao reuso de código, uma vez que permite a reutilização de modelos de simulação individuais previamente existentes, que podem inclusive utilizar métodos de solução distintos. Além disso, a solução de vários sistemas de equações menores (um para cada física) é geralmente mais eficiente do que a solução de um grande sistema de equações[1].

Quando os fenômenos físicos acoplados são estudados em escalas diferentes, no tempo e/ou no espaço, a simulação é denominada de multiescala. Nestas situações, pode ser necessária a utilização de discretizações diferentes na solução de cada um dos problemas, com constantes trocas de valores entre estas. Um exemplo aplicado à modelagem de reservatórios consiste na simulação de parâmetros constitutivos dos materiais, ocorrendo em escala de grãos, acoplada com a simulação do fluxo de fluidos em um reservatório fraturado, que ocorre em escala do reservatório. Os artigos em Horstemeyer (2006) e Fish (2006) fazem uma extensa revisão sobre o assunto e sobre os métodos disponíveis.

[1]Se considerarmos, por exemplo, que o sistema linear de equações será resolvido pelo método de decomposição LU, a solução será $O(N^3)$.

3
Trabalhos relacionados

Existe vasta literatura a respeito de bibliotecas para suporte a análises pelo método de elementos finitos. A série de artigos por Zimmermann *et al* (1992), Dubois-Pèlerin *et al* (1992) e Dubois-Pèlerin e Zimmermann (1992) foi uma das pioneiras em explorar sua relação com a análise orientada a objetos, seguida por diversas outras iniciativas de *frameworks* para auxílio na construção de simuladores usando o método. Uma descrição resumida dos principais trabalhos pode ser encontrada em Beghini *et al* (2014). A mesma diversidade, no entanto, não é encontrada em relação a *frameworks* de suporte para análises multifísicas.

Vários trabalhos descrevendo simuladores e/ou novas infraestruturas para construção de simulações multifísicas apresentam um foco no conjunto de equações que regem o problema, apresentando, no entanto, poucos detalhes sobre a arquitetura utilizada. Em Wheeler *et al* (2000), por exemplo, os diversos modelos matemáticos acoplados disponíveis no *framework* IPARS para simulação de fluxo multifásico em meios porosos são apresentados em detalhes, enquanto aspectos de sua arquitetura computacional são pouco explorados.

A maioria dos *frameworks* multifísicos estudados baseia-se no uso de um único método de simulação, com predominância para o método de elementos finitos. Os projetos Elmer, MOOSE, ambos detalhados nas seções a seguir, e IPARS são exemplos de *frameworks* baseados apenas no método de elementos finitos. A biblioteca OpenFOAM (Weller *et al*, 1998), um *framework* voltado, entre outros, para diversos tipos de simulações de dinâmica dos fluidos, utiliza exclusivamente o método de volumes finitos. Já o *framework* Rocstar, também detalhado nas seções a seguir, permite a utilização de ambos.

Dentre os *frameworks* estudados, o projeto Rocstar é o único a suportar a integração de simuladores que não foram desenvolvidos especificamente para o *framework*. Entretanto, este requer que o código fonte do simulador externo seja adaptado, o que exclui a possibilidade de integração de simuladores para os quais o mesmo não está disponível (simuladores comerciais, por exemplo).

Neste capítulo são apresentadas as principais características dos trabalhos diretamente relacionados com o desenvolvimento do *framework* GeMA, fornecendo uma base para as comparações discutidas no capítulo 7.

3.1 Elmer

O sistema Elmer é uma aplicação multifísica de uso geral com suporte, dentre outros, a físicas para transferência de calor, fluxo de fluidos, elasticidade, acústica e eletromagnetismo. Desenvolvido pelo CSC, Centro de Tecnologia da Informação para a Ciência, uma instituição do governo finlandês, seu desenvolvimento começou em 1995 e segue ativo com grande comunidade de usuários (Lyly *et al*, 1999; Råback e Malinen, 2014).

Apesar de ser manifestamente um sistema, e não um *framework*, sua arquitetura de componentes incentiva os usuários a criarem novas físicas através de *plugins* escritos em Fortran 90, denominados de "resolvedores". Estes, ao serem adicionados ao sistema, passam a fazer parte do ambiente de forma integrada.

Através de um arquivo de configuração, normalmente gerado por uma interface gráfica, o usuário seleciona as equações a serem executadas dentre o conjunto disponível e os diversos parâmetros utilizados na simulação. Esta baseia-se exclusivamente no método de elementos finitos e segue um esquema de acoplamento sequencial muito similar ao apresentado na figura 2.4 (Ruokolainen *et al*, 2014). O sistema é responsável pelos laços temporal e de acoplamento entre as físicas. Os resolvedores são responsáveis pelas demais iterações, incluindo laços para percorrer elementos e condições de contorno.

Embora físicas fortemente acopladas possam ser criadas como um único resolvedor englobando todo o cálculo acoplado, o ambiente favorece o uso de acoplamento fraco.

A criação de um novo resolvedor é relativamente complexa, embora existam diversas funcionalidades de suporte disponíveis para o *plugin*. Funções fornecidas pelo usuário para o cálculo de propriedades também podem ser fornecidas através de *plugins* específicos.

Para que o ambiente suporte a criação de novos resolvedores, a descrição da simulação é feita através de um conceito de atributos e valores genéricos interpretados pelos resolvedores, conceito este que é passado para a interface gráfica.

Processos para adaptação e eventual refinamento das malhas são tratados como se estes fossem um resolvedor. Não há um conceito de orquestração configurável. Cada um dos resolvedores selecionados na simulação são executados em ordem pelo laço de acoplamento. A única influência que o usuário pode ter consiste em definir a ordem em que os resolvedores serão executados.

O sistema Elmer oferece suporte à paralelização das simulações tanto em ambientes com múltiplos núcleos de processamento quanto em ambientes com múltiplos processadores. Durante a fase de construção do modelo, o domínio da

simulação deve ser decomposto nas partições que serão executadas em paralelo. Segundo Ruokolainen *et al* (2014), toda a comunicação necessária é feita pelo ambiente, de forma praticamente transparente para os resolvedores, com uso de trocas de mensagem através do padrão MPI (*Message Passing Interface*).

O sistema está disponível nas plataformas Linux e Windows, sendo publicado com licença do tipo GPL, o que implica que o mesmo não pode ser utilizado para o desenvolvimento de códigos comerciais. A tabela 3.1 sumariza suas principais características.

Elmer	
Domínio de aplicação	Propósito geral com suporte a diversos tipos de físicas.
Métodos de simulação e acoplamento suportados	Elementos finitos. Voltado para o acoplamento sequencial das físicas, embora seja possível criar resolvedores com acoplamento forte.
Suporte à criação de novas físicas	Sim, através de sub-rotinas disponíveis aos resolvedores físicos.
Prototipação rápida de novas físicas	Não.
Orquestração configurável	Não.
Acoplamento com outros simuladores	Não.
Suporte à paralelização	Sim, através de troca de mensagens via uso do padrão MPI.
Disponibilidade e licença	Disponível através da página do projeto (https://www.csc.fi/web/elmer) para os ambientes Windows e Linux. Licença GPL.

Tabela 3.1: Resumo das principais características do sistema Elmer.

3.2 Rocstar

O *framework* Rocstar tem como objetivo facilitar a construção de sistemas voltados para simulações de motores de foguetes movidos a combustível sólido, incluindo módulos físicos para o tratamento de processos de dinâmica de fluidos, combustão, esforços mecânicos e caracterização de materiais (Jiao *et al*, 2006; Brandyberry *et al*, 2012). Desenvolvido originalmente pelo centro para simulação de foguetes avançados da Universidade de Illinois, é atualmente mantido pela empresa Illinois Rocstar.

Voltado para simulações pelos métodos de elementos finitos e volumes finitos, o *framework* Rocstar é composto por três módulos principais que têm como

objetivo, respectivamente, integrar e orquestrar os módulos físicos, bem como prover diversas funcionalidades de suporte a uma simulação multifísica.

Um dos objetivos deste *framework* consiste em permitir que os módulos de física sejam desenvolvidos de maneira independente por equipes distintas e que estas possam escolher individualmente o modelo e a tecnologia mais apropriada para cada módulo e para cada equipe de desenvolvimento, incluindo a definição das estruturas de dados e da linguagem de programação utilizadas.

Para que seja mínima a necessidade de alterações em cada um dos simuladores individuais durante sua integração com o ambiente de simulações multifísicas controlado pelo *framework*, o módulo de integração disponibiliza uma API (*Application Programming Interface*) que cumpre um papel similar ao de uma IDL (*Interface Definition Language*) para descrever como os dados de cada simulação estão armazenados. Esta descrição não se limita apenas ao tipo dos dados e se os mesmos estão associados a nós ou elementos da discretização espacial, incluindo também questões de como os mesmos estão organizados fisicamente na memória. Com versões em C/C++ e em Fortran 90, estas interfaces são utilizadas pelos demais módulos do *framework* para acesso aos dados de cada módulo físico em uma maneira independente da linguagem utilizada.

O módulo de integração permite também que cada módulo físico registre funções implementadas pelo módulo para que estas possam ser chamadas por outros módulos, também de maneira independente de linguagem e compilador. Ainda que com algumas restrições quanto aos tipos de dados que podem ser utilizados nestas funções, o módulo de integração efetua as conversões necessárias para compatibilizar os diversos protocolos de passagem de parâmetros e de chamadas de funções. Cada módulo físico pode ser implementado como uma biblioteca dinâmica, sendo também uma função do módulo de integração efetuar a carga destes módulos.

Em Brandyberry *et al* (2012), os autores consideram que a existência do módulo de integração para permitir a troca de dados entre simulações pré-existentes é uma das principais razões de sucesso do ambiente.

O módulo de suporte inclui, entre outras, funções para leitura e escrita de dados, transferência de dados entre malhas, propagação de superfícies e adaptação de malhas.

O módulo de orquestração tem como objetivo coordenar as múltiplas físicas em uma simulação acoplada, sendo o responsável pelo laço principal da aplicação e pela execução das ações contidas em um esquema de acoplamento. Ações são objetos em C++ que representam interações entre módulos físicos e especificam seus dados de entrada e saída, podendo incluir tanto a execução de funcionalidades

fornecidas pelos módulos de física quanto funcionalidades fornecidas pelo módulo de suporte.

Através da construção de um grafo de ações, onde cada aresta indica a transferência de dados de uma ação para a outra, o *framework* determina automaticamente a ordem em que as mesmas devem ser executadas.

A proposta do *framework* é ser massivamente paralelo. Para isso, cada módulo físico deve ser construído individualmente como um simulador paralelo utilizando o padrão MPI para troca de mensagens. O *framework* utiliza a biblioteca AMPI (Huang *et al*, 2003), uma implementação do padrão MPI que suporta o conceito de balanceamento dinâmico de carga tendo como base a linguagem Charm++ (Kalé e Krishnan, 1993).

O *framework* Rocstar foi disponibilizado ao público apenas em 2015, através de licença aberta que permite seu uso em aplicações comerciais. A tabela 3.2 sumariza suas principais características.

Rocstar	
Domínio de aplicação	Específico, criado para simulações de motores de foguetes movidos a combustível sólido.
Métodos de simulação e acoplamento suportados	Elementos finitos e volumes finitos. Voltado para o acoplamento sequencial das físicas, embora seja possível criar simuladores com acoplamento forte.
Suporte à criação de novas físicas	Não.
Prototipação rápida de novas físicas	Não.
Orquestração configurável	O módulo de orquestração define o modo de acoplamento entre físicas utilizado e coordena a execução dos processos necessários para a execução da simulação através do uso de uma API em C++.
Acoplamento com outros simuladores	Suporta a integração de simuladores pré-existentes através de alterações ao seu código fonte para permitir que o *framework* tenha acesso aos dados do simulador.
Suporte à paralelização	Sim, através de troca de mensagens via uso do padrão AMPI.
Disponibilidade e licença	Disponível através da página do projeto (http://www.illinoisrocstar.com/products/rocstar) para ambiente Linux. Licença "University of Illinois/NCSA Open Source License".

Tabela 3.2: Resumo das principais características do *framework* Rocstar.

3.3 MOOSE

O *framework* MOOSE (*Multiphysics Object-Oriented Simulation Environment*) é um ambiente genérico para construção de simulações multifísicas (Gaston *et al*, 2009). Construído no laboratório "Idaho National Labs", uma de suas aplicações iniciais tem como objetivo suportar o desenvolvimento do sistema BISON (Novascone *et al*, 2013), um simulador para avaliar o comportamento de combustível nuclear no ambiente de um reator.

Voltado primordialmente para a solução de problemas massivamente paralelos, não-lineares e fortemente acoplados, o *framework* utiliza um método bastante específico de solução por elementos finitos onde o sistema não-linear é resolvido exclusivamente pelo método JFNK ("Jacobian Free Newton Krylov"), com discretização temporal feita de forma implícita.

De acordo com Gaston *et al* (2009) e Novascone *et al* (2013), nesta técnica o método de Newton é utilizado para a solução do sistema acoplado de equações não-lineares, porém sem a necessidade de construção completa da matriz Jacobiana do sistema já que o método requer apenas o resultado da multiplicação desta matriz por um vetor, que pode ser aproximada através de diferenças finitas. Uma descrição do método e de suas aplicações pode ser encontrada em Knoll e Keyes (2004).

Neste *framework*, novas físicas são geradas através da criação de *kernels*, responsáveis por descrever a forma fraca da equação a ser simulada. Cada *kernel* representa um termo da equação e deve retornar o cálculo do resíduo a ser integrado em cada ponto de Gauss de um elemento. Opcionalmente, pode ser retornada também uma aproximação para o Jacobiano no ponto de integração, utilizado para pré-condicionar a solução.

Através do uso extensivo de C++ e da disponibilização de diversos valores pré-calculados nos pontos de integração para os *kernels*, estes geralmente podem ser implementados através de uma simples expressão que representa diretamente a forma fraca da equação, o que simplifica em muito a criação de novas físicas e favorece seu uso para a prototipação rápida de soluções.

Embora o *framework* tenha sido construído para a execução de simuladores fortemente acoplados, a execução de simulações fracamente acopladas é possível através do uso de classes especiais de controle do laço principal que consideram cada física da simulação fracamente acoplada como uma simulação independente.

Em sua versão disponível ao público através de licença LGPL, o *framework* MOOSE conta, dentre outras, com físicas para cálculos de condução de calor,

tensões mecânicas, fluxo multifásico através de meios porosos, reações químicas, e dinâmica de fluidos[1]. A tabela 3.3 sumariza suas principais características.

MOOSE	
Domínio de aplicação	Propósito geral com suporte a diversos tipos de físicas.
Métodos de simulação e acoplamento suportados	Elementos finitos, voltado para simulações fortemente acopladas. Solução exclusivamente através do método JFNK.
Suporte à criação de novas físicas	Sim, através do uso de classes e serviços providos pela biblioteca.
Prototipação rápida de novas físicas	Sim, através da definição de expressões que representam diretamente a forma fraca das equações a serem simuladas.
Orquestração configurável	Não.
Acoplamento com outros simuladores	Não.
Suporte à paralelização	Sim, através do uso dos serviços providos pela biblioteca libMesh.
Disponibilidade e licença	Disponível através da página do projeto (http://www.mooseframework.org/) para ambiente Linux. Licença LGPL.

Tabela 3.3: Resumo das principais características do *framework* MOOSE.

[1]Físicas mais específicas sujeitas ao controle de exportação de tecnologia pelo governo americano não são disponibilizadas na versão pública do *framework*.

4
O framework GeMA

O *framework* GeMA tem como objetivo prover um ambiente de suporte para a criação de simulações multifísicas onde o engenheiro responsável pela simulação tenha liberdade para escolher o modelo adotado. Através de uma arquitetura centrada no conceito de orquestração configurável, extensível e modular pelo uso de interfaces abstratas e *plugins*, o *framework* fornece um ambiente flexível para a definição do processo de solução, com suporte ao uso de diversos métodos de discretização e acoplamento.

Para o *framework*, uma simulação é composta por três elementos principais (figura 4.1): *dados do modelo*, *método de solução* e *monitor de resultados*. Os dois primeiros definem, respectivamente, o que será simulado e como a simulação será feita. O terceiro define tanto quais e como os resultados da simulação serão salvos quanto que variáveis devem ser monitoradas enquanto a simulação é executada. Essas definições, e as demais a seguir, tem como objetivo estabelecer uma terminologia que defina, de forma não ambígua, quais são os elementos envolvidos na simulação. Embora a nomenclatura utilizada busque seguir o senso comum da área, para facilitar seu entendimento, alguns termos podem ter significado um pouco diferente dos utilizados na linguagem natural.

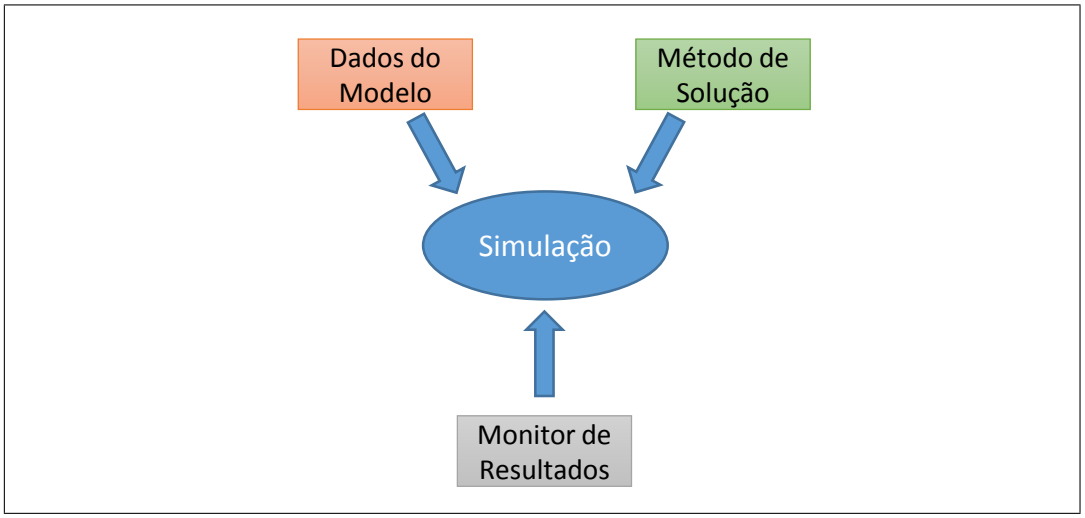


Figura 4.1: Componentes de uma simulação para o *framework* GeMA.

A execução de uma simulação é feita em dois passos. O primeiro consiste em carregar os dados do modelo, a definição do método de solução e a lista das

variáveis a serem monitoradas. Normalmente estes parâmetros são lidos de arquivos conforme descrito na seção 4.7. A seguir, o *script* de orquestração, parte central do método de solução, é interpretado. Durante sua execução, são ativados processos que cooperam para gerar o resultado da simulação, enquanto o monitor de resultados se encarrega de salvar os dados calculados e sinalizar a evolução das variáveis monitoradas.

O conjunto de dados do modelo define as características básicas da simulação, tais como seu domínio espacial e as condições de contorno aplicadas, representando *o que* será simulado. Suas principais entidades são apresentadas na figura 4.2 e detalhadas a seguir.

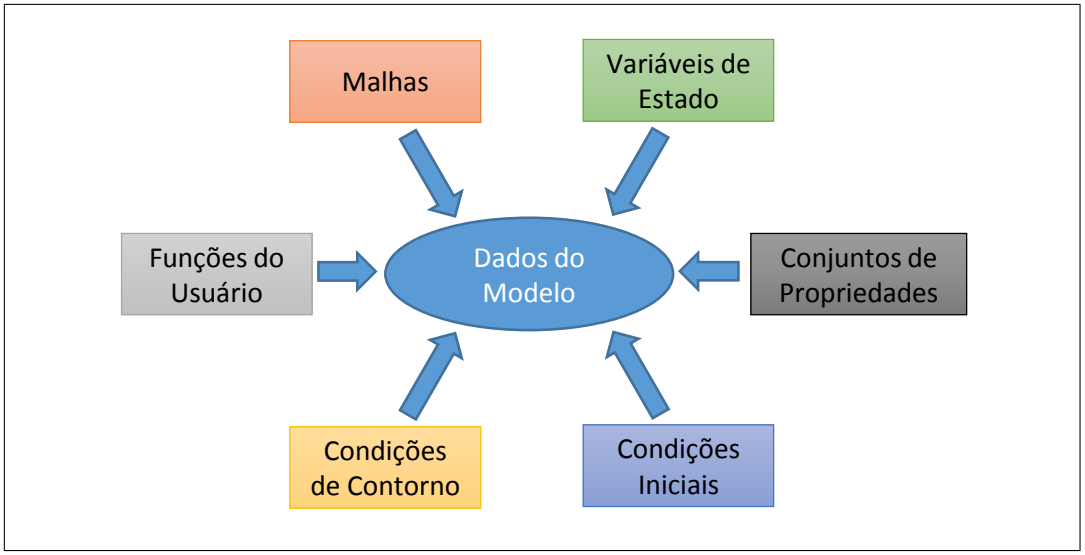


Figura 4.2: Principais entidades associadas ao conjunto de dados do modelo.

Malhas: Definem a discretização do domínio espacial sobre a qual a simulação será efetuada. Nós e células da malha podem ser associados a diversos conjuntos de dados definidos pelo usuário. Múltiplas discretizações, representando domínios distintos ou o mesmo domínio em várias escalas, podem ser associadas ao modelo. Conforme discutido na seção 4.2, neste trabalho o termo "malha" é generalizado para incluir todos os tipos de discretização suportados.

Variáveis de estado: Representam o conjunto de grandezas físicas, escalares, vetoriais ou tensoriais, que serão calculadas pela simulação, estando associadas aos graus de liberdade do sistema. As variáveis de estado são definidas no domínio espacial e, consequentemente, uma mesma variável de estado pode estar associada com múltiplas malhas.

Conjuntos de propriedades: Permitem a definição de tabelas com listas de propriedades, geralmente associadas às características físicas dos materiais,

mas que também podem ser utilizadas para determinação de quaisquer outras características que precisem ser associadas a um sub-domínio da discretização espacial.

Condições iniciais: Representam os valores das variáveis de estado no início da simulação.

Condições de contorno: Representam as informações do contorno do domínio espacial necessárias para a completa definição do modelo físico, podendo ser aplicadas sobre nós, células ou bordas da malha, estas últimas representadas por arestas para modelos 2D e faces para modelos 3D.

Funções do usuário: Permitem que valores associados às condições de contorno ou aos nós e células da malha possam ser avaliados durante a execução da simulação através de funções do usuário, expressas através de *scripts* escritos na linguagem Lua ou através de funções escritas em C++.

O método de solução define *como* os resultados da simulação serão calculados. Suas principais entidades são apresentadas na figura 4.3 e detalhadas a seguir.

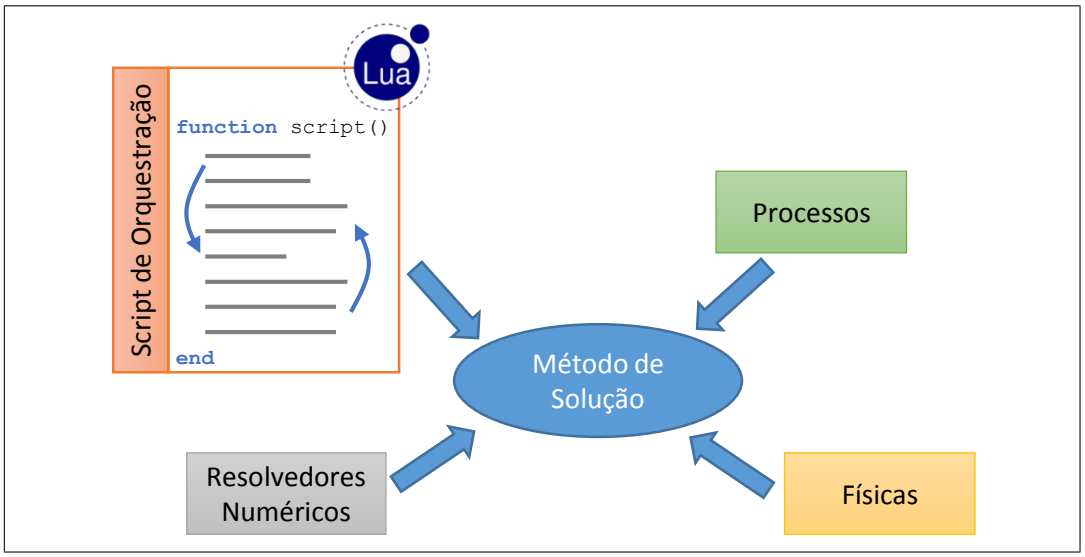


Figura 4.3: Principais entidades associadas ao método de solução.

***Script* de orquestração:** É o objeto central do método de solução e um dos principais responsáveis pela flexibilidade do *framework*. Escrito na linguagem Lua, define, através de chamadas de funções e comandos de fluxo próprios da linguagem (laços, condições, definição de funções, etc), a sequência de processos a serem executados ao longo da simulação, fornecendo assim seu *loop* principal. Devido à expressividade da linguagem, o orquestrador pode executar operações tão complexas quanto necessárias.

Processos: São a unidade básica utilizada para a descrição da solução, podendo ser escritos em C++ ou em Lua. Em geral são primitivas de alto nível que descrevem um ação completa, tais como a execução de uma análise através do método de elementos finitos, a transferência de dados de uma malha para outra, o refinamento adaptativo de uma malha, o salvamento de um conjunto de dados, etc.

Quando um processo é executado pelo orquestrador, pode receber como parâmetros diversas entidades do modelo. Em particular, processos de análise fazem uso de físicas e resolvedores numéricos.

Físicas: Representam os objetos responsáveis por auxiliar os processos de análise a montar o sistema de equações a ser resolvido. No âmbito de uma análise por elementos finitos, por exemplo, as físicas cooperam para produzir a matriz local de um elemento que será combinada na matriz de rigidez global.

Resolvedores numéricos: São as entidades responsáveis pela solução dos sistemas de equações. Diferentes resolvedores podem usar métodos numéricos distintos, tais como os métodos de decomposição LU ou gradientes conjugados, e geralmente são implementados com auxílio de bibliotecas externas que implementam os algoritmos de cálculo.

O monitor de resultados define quais, como e quando os resultados gerados pela simulação serão salvos, tendo como base um *script* de configuração fornecido na parametrização da simulação e processos para salvamento de dados em diversos formatos. Além disso, o monitor de resultados permite que a aplicação hospedeira possa apresentar ao usuário uma interface para acompanhar a simulação e, eventualmente, cancelar sua execução. A figura 4.4 ilustra este comportamento.

No restante deste capítulo, a seção 4.1 detalha a arquitetura global do *framework* com ênfase na estrutura de interfaces abstratas e *plugins*. A seção 4.2 apresenta os conceitos e classes para suporte a malhas e dados associados com estas. Maiores detalhes sobre o *script* de orquestração e processos em geral, com ênfase no suporte ao método de elementos finitos, são apresentados na seção 4.3, enquanto o papel das físicas é explorado na seção 4.4 e as interfaces para resolvedores numéricos são detalhadas na seção 4.5, completando assim a descrição das principais entidades necessárias para a execução de uma simulação. A seção 4.6 apresenta detalhes sobre o monitor de resultados e, finalmente, a seção 4.7 aborda e exemplifica os arquivos utilizados para a definição do modelo de simulação.

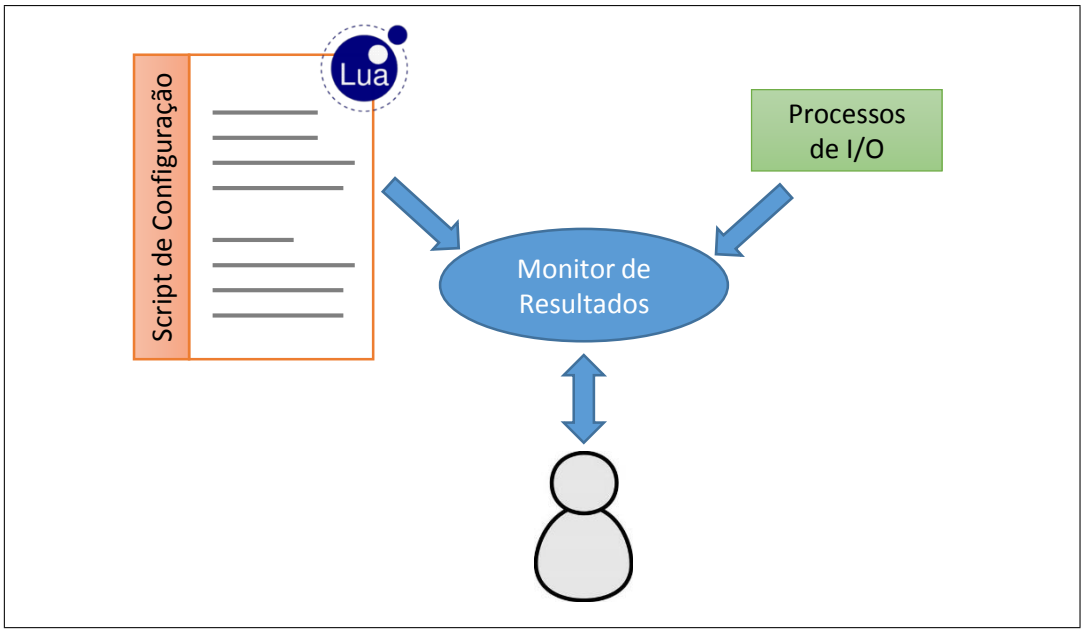


Figura 4.4: Principais entidades associadas ao monitor de resultados.

4.1 Arquitetura

A arquitetura do *framework* GeMA tem como objetivo garantir sua extensibilidade e modularidade, permitindo que seus usuários possam estendê-lo com novos métodos de discretização e/ou novas físicas, sem que seja necessário efetuar alterações em seu núcleo principal. A primeira parte da solução adotada consiste no uso de uma arquitetura baseada em *plugins*. A segunda, no uso da linguagem Lua.

Plugins são objetos externos ao programa principal que exportam funções que este reconhece e utiliza quando necessário (Birsan, 2005). Para isso, é necessário que a aplicação possua pontos de extensão bem definidos e que esteja preparada para descobrir a existência e para carregar estes objetos externos, geralmente representados por bibliotecas dinâmicas, em tempo de execução.

Cada um destes pontos de extensão deve estar associado a uma ou mais interfaces abstratas, precisamente definidas, uma vez que a comunicação entre a aplicação e o *plugin* se dá exclusivamente através destas. Estas interfaces funcionam como contratos. Para a aplicação, não importa como um *plugin* estrutura seus dados ou executa suas operações, apenas que, quando acionado, este execute as ações requisitadas na forma "contratada".

Conforme Cervantes (2006), *plugins* e componentes de *software* são entidades bastante semelhantes. Ambos são objetos independentes que se comunicam com outras entidades através de interfaces bem definidas. Arquiteturas orientadas a componentes, porém, geralmente não consideram que estes sejam opcionais e

carregados em tempo de execução. Componentes são usados para facilitar a construção da aplicação, extensível ou não, e sua manutenção futura.

A utilização de *plugins* é uma tecnologia consolidada, largamente encontrada em diversos tipos de projetos de *software*, tais como navegadores e aplicativos de desenho. Entre as plataformas de simulação multifísica, o sistema Elmer utiliza *plugins* para implementação de seus modelos físicos (seção 3.1). Dentre as principais vantagens do uso de *plugins*, pode-se salientar:

- Permitem a adição de componentes ao sistema por terceiros de forma simples. O código existente não precisa ser modificado ou mesmo recompilado.
- Durante a execução de uma simulação, apenas os componentes necessários são carregados para a memória.
- A utilização de *plugins* força a existência de uma clara separação de conceitos e de interdependências. Na medida em que cada *plugin* é um componente a parte, que depende apenas de um conjunto de bibliotecas padrão que fornecem os serviços básicos do *framework*, a criação de dependências indesejadas entre os *plugins* é dificultada. Isto promove o desacoplamento entre os módulos do sistema, facilitando sua manutenção e evolução.

A principal desvantagem do uso de um sistema de *plugins* está na complexidade adicional necessária para dar suporte à busca de novos componentes e à carga de bibliotecas dinâmicas de maneira portável[1]. Felizmente esta complexidade, uma vez implementada e empacotada em um serviço do *framework*, torna-se transparente para o desenvolvimento de extensões ao mesmo.

No ambiente GeMA, seis dos conceitos centrais ao *framework*, apresentados anteriormente, são representados através de interfaces abstratas e implementados através do uso de *plugins*. São eles:

- Malhas,
- Conjuntos de propriedades,
- Processos,
- Físicas,
- Resolvedores numéricos e
- Processos de I/O.

O uso de interfaces para representar estes conceitos é um dos pilares do *framework* GeMA. A ideia principal consiste em que o *framework* especifica o

[1]Uma característica importante do *framework* GeMA é ser portável entre as plataformas Windows e Linux

comportamento esperado e a API destes objetos, delegando às suas implementações concretas os detalhes de como estes comportamentos são obtidos. No ambiente GeMA, a definição das interfaces pertence à biblioteca principal do *framework*, enquanto as implementações concretas pertencem aos *plugins*.

Cada *plugin* estende o ambiente de simulação fornecendo objetos de um destes tipos, ou seja, implementa uma ou mais interfaces abstratas relacionadas a um destes conceitos centrais conforme ilustrado na figura 4.5.

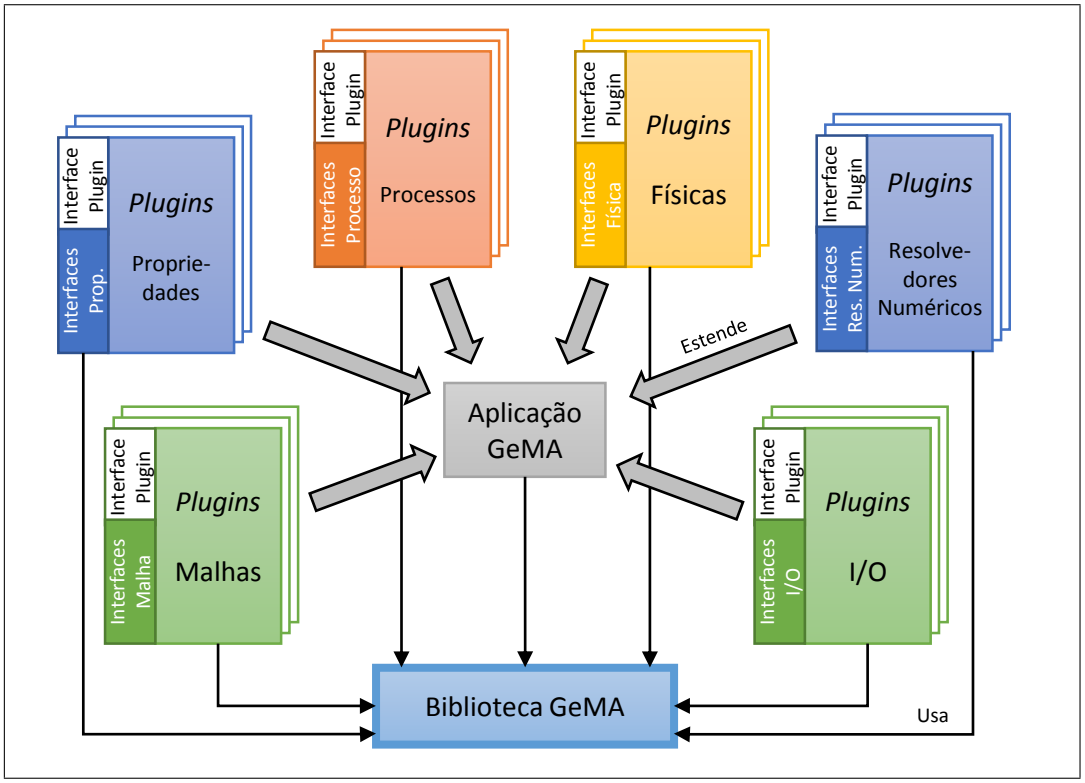


Figura 4.5: Arquitetura de *plugins* para o *framework* GeMA.

O *framework* GeMA não pode ser utilizado sem a presença de *plugins* fornecendo uma implementação concreta de malhas, processos, físicas, etc. Para facilitar sua utilização, o ambiente GeMA inclui implementações de referência para todos os tipos de *plugin* listados anteriormente e uma aplicação base para execução de simulações.

Cada interface abstrata pode ser implementada por vários *plugins*, sendo possível especificar, nos dados do modelo e no método de solução, quais deverão ser utilizados em uma simulação. Esta característica pode ser explorada na integração com outros simuladores a nível de componentes. Pode-se ter, por exemplo, uma implementação padrão para suporte a malhas que armazena internamente os dados da mesma e outra que utiliza uma estrutura de dados externa fornecida por outro simulador/biblioteca. Neste segundo cenário, a interface abstrata de malhas funciona como um adaptador que integra o ambiente GeMA ao componente externo.

Na definição de interfaces, buscando maximizar a facilidade de integração com outras bibliotecas que porventura já possuam estruturas de dados próprias, o *framework* GeMA procura utilizar, sempre que possível, tipos padrão da linguagem C++ nas assinaturas de métodos, evitando a utilização de listas e tipos estruturados. Esta regra, entretanto, é utilizada apenas como um guia geral, sendo "desrespeitada" quando seu uso implicar em uma *API* desnecessariamente complexa.

A figura 4.6 ilustra a maneira como *plugins* são implementados no *framework* GeMA, bem como os processos de descoberta dos *plugins* existentes, sua carga e instanciação de objetos fornecidos pelo *plugin*.[2]

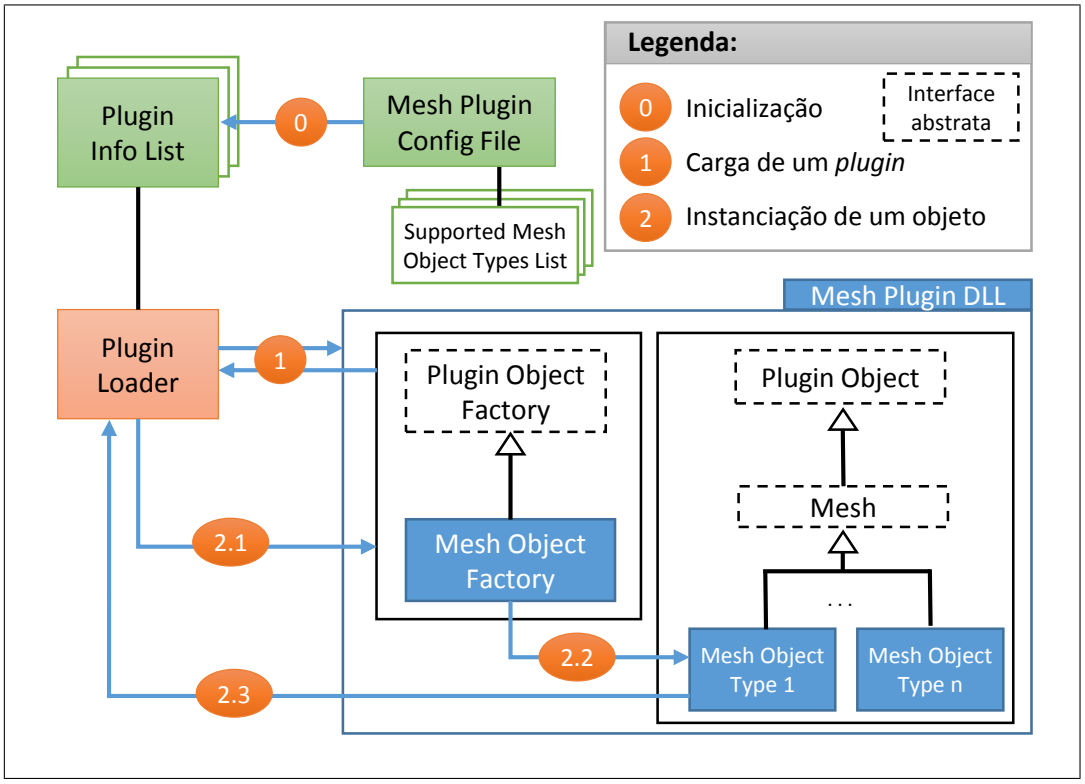


Figura 4.6: Instanciando objetos (malhas) implementados por um *plugin*.

Quando o *framework* é inicializado, alguns diretórios padrão são pesquisados à procura dos *plugins* disponíveis. São considerados como um *plugin* pares de arquivos formados por uma biblioteca dinâmica e um arquivo de configuração, ambos com mesmo nome. O arquivo de configuração define as características básicas do *plugin*. Durante a inicialização, apenas o arquivo de descrição é lido e interpretado. Desta forma, o sistema monta uma lista com informações sobre os *plugins* disponíveis sem a necessidade de carga das bibliotecas dinâmicas (fluxo '0' na figura 4.6).

[2] As entidades descritas na figura 4.6 possuem nomes em inglês para facilitar sua correspondência com as classes implementadas no *framework* GeMA. Isto se repete em outras figuras onde nomes de classes são utilizados.

Durante a leitura dos dados do modelo e do método de solução, ao ser identificada a necessidade de instanciação de um objeto implementado por um *plugin* (ver seção 4.7), a lista de *plugins* disponíveis é consultada para verificar se este já foi carregado para a memória. Em caso negativo, a biblioteca dinâmica é carregada, devendo retornar uma fábrica para construção dos objetos implementados pelo *plugin*, que é então inserida na lista (fluxo '1' na figura 4.6).

Esta fábrica implementa a interface abstrata "**PluginObjectFactory**", responsável por instanciar objetos concretos que herdam da interface "**PluginObject**" e da interface associada ao tipo de *plugin*, "**Mesh**" no exemplo da figura. Como cada *plugin* pode retornar objetos de mais de um tipo, a informação do tipo desejado é passada pelo carregador de *plugins* para a fábrica de objetos, que usa esta informação para instanciar o objeto correto (fluxo '2' na figura 4.6).

A linguagem Lua possui quatro papeis distintos no âmbito da arquitetura do *framework*, sendo responsável pela possibilidade de prototipação rápida dos métodos de solução e pela flexibilidade na orquestração das simulações. São eles:

1. Permitir a extensão dos modelos, tanto através de funções fornecidas pelo usuário, associadas às condições de contorno e/ou a atributos e propriedades da malha (seção 4.2), quanto através da possibilidade de criação de processos e físicas em Lua, essencial para permitir a prototipação rápida de soluções (seção 4.4).

2. Permitir a orquestração de processos, definindo o *loop* principal da aplicação (seção 4.3).

3. Permitir uma descrição flexível do modelo de simulação através do uso de uma sintaxe adaptada ao domínio do ambiente GeMA (seção 4.7). Por ser uma linguagem completa, o uso de Lua permite que a definição do modelo inclua características dinâmicas, como por exemplo a definição de funções para construção de malhas regulares a partir dos limites do domínio.

4. Facilitar a integração com simuladores existentes, atuando como uma linguagem para ligação entre componentes.

4.2 Discretização espacial

Métodos numéricos de simulação têm como base a discretização espacial do modelo a ser simulado. Esta discretização divide o domínio espacial da simulação em partições menores sobre as quais as equações que definem o modelo discreto são resolvidas, geralmente de maneira aproximada. Segundo Felippa (2004), o processo

de discretização espacial pode ser definido como a conversão do modelo contínuo em um modelo discreto composto por um número finito de graus de liberdade.

Diferentes métodos de discretização geram domínios discretos com características distintas. Em simulações por diferenças finitas, por exemplo, este é composto por uma malha regular e não é necessário armazenar informações sobre coordenadas dos nós ou vizinhança entre células, uma vez que estas informações podem ser obtidas implicitamente a partir do índice dos nós. Já em simulações por elementos finitos, cada célula da malha resultante do processo de discretização possui uma geometria própria, e portanto é necessário armazenar as coordenadas de cada nó, bem como a lista de nós que a compõe. Outros métodos, conhecidos como *mesh free*, armazenam dados associados a conjuntos de pontos, sem qualquer relação arbitrária de vizinhança entre os mesmos, mas necessitam de estruturas de dados espaciais auxiliares para otimizar a busca dos vizinhos mais próximos de cada ponto.

Desta forma, de modo a permitir o suporte a vários tipos de modelo de simulações, é necessário também o suporte a vários tipos de domínios discretos. De uma maneira bastante genérica, o *framework* GeMA denomina de "malha" o resultado do processo de discretização espacial, mesmo que este seja composto apenas por uma nuvem de pontos.

Para o *framework*, uma malha é composta por um conjunto de nós, cada um contendo sua posição no espaço e, possivelmente, associado a um conjunto de dados. Esta é uma definição extremamente "aberta", não implicando em nenhum tipo de topologia e nem mesmo que a malha precise existir fisicamente (em associação com uma estrutura de dados).

Para grande parte dos métodos de simulação, tais como os métodos de diferenças finitas, elementos finitos e volumes finitos, a relação de vizinhança entre os nós da malha é parte fundamental do método. Este conceito genérico de malha explicitamente posterga a descrição de relacionamento entre os nós para especializações da malha, uma vez que cada método de simulação possui requisitos possivelmente distintos.

As diversas especializações necessárias se dão através de uma hierarquia de interfaces abstratas, conforme ilustrado nas figuras 4.7 e 4.8. Na implementação atual do *framework* são três os níveis básicos de suporte:

1. **Malha de nós:** Corresponde à base da hierarquia de interfaces de suporte a malhas, indicada pela classe **Mesh** na figura 4.7 e representada pela figura 4.8(a). Define operações sobre nós, incluindo sua associação com conjuntos de dados. Especializações desta interface base são usadas para a descrição de todos os outros tipos de malhas.

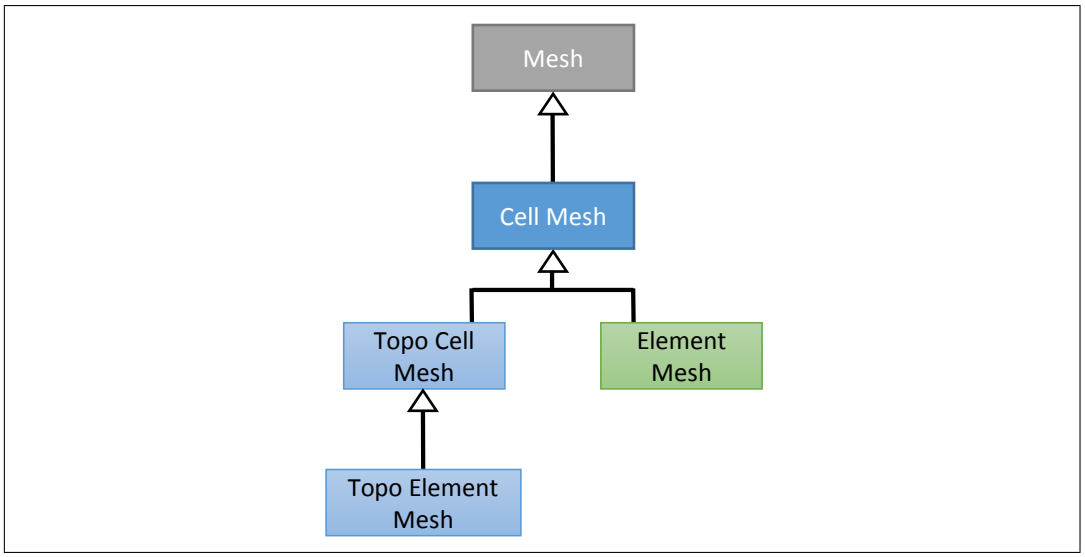


Figura 4.7: Interfaces para representação de malhas.

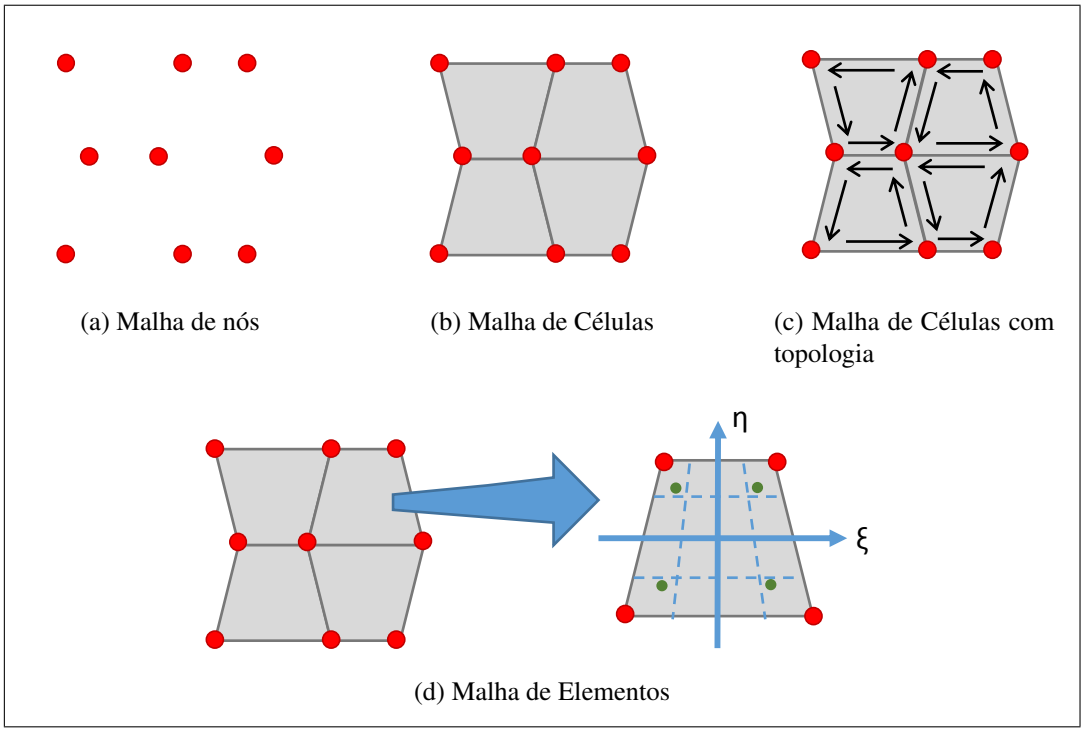


Figura 4.8: Tipos de malhas. (a) Malha de nós; (b) Malha de células; (c) Malha de células com topologia; (d) Malha de elementos.

2. **Malha de células:** Estende a malha de pontos adicionando a esta o conceito de células. Cada célula define uma partição do domínio global compreendido pela malha, representando em 1D um segmento, em 2D uma área e em 3D um volume. Em geral, células de uma malha são disjuntas e sua união representa o domínio como um todo.

   Cada célula é formada por um conjunto ordenado de nós que coletivamente definem a geometria de sua fronteira. O esquema que descreve quantos nós existem em cada célula e sua organização definem o tipo da célula.

Malhas de células são representadas pela classe **CellMesh** na figura 4.7 e pela figura 4.8(b).

3. **Malha de elementos:** Estende o conceito anterior transformando cada célula em um elemento, sendo o tipo base para malhas utilizadas em análises por elementos finitos. Cada elemento é composto por uma célula associada a uma função de forma (*shape-function*) que define como valores são interpolados dentro do elemento. Uma função de forma associada com um tipo de célula define um tipo do elemento. Malhas de elementos são representadas pela classe **ElementMesh** na figura 4.7.

   Elementos também são associados a regras de integração numérica, representados pelos pontos em verde na figura 4.8(d).

Esta hierarquia básica pode ser estendida para incorporar outros conceitos de acordo com a necessidade. Um exemplo de extensão consiste em acrescentar informações adicionais para armazenar a topologia da malha, conforme ilustrado na figura 4.8(c) onde a topologia é armazenada através de uma estrutura de *half-edge*.

4.2.1 Entidades associadas a malhas

Nós e células das malhas podem ser associados a dados conforme será descrito na próxima seção. Além disso, malhas compostas por células e elementos estão também associadas a outras entidades importantes conforme ilustrado na figura 4.9 e detalhadas a seguir.

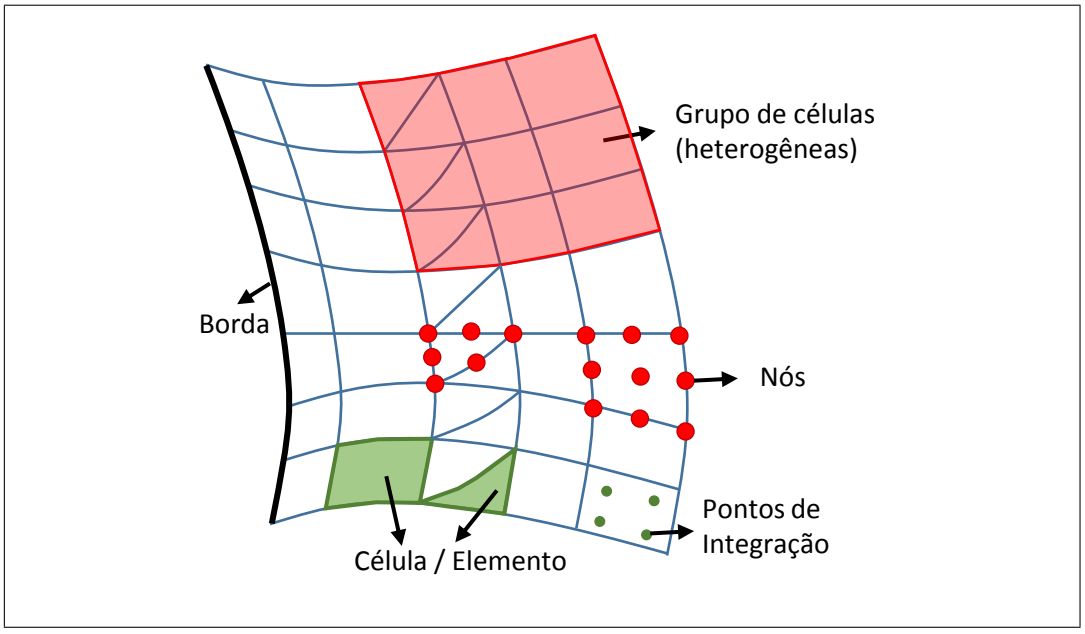


Figura 4.9: Entidades associadas a uma malha.

Nós: Definem os vértices que em conjunto formam as células / elementos da malha. Sua dimensão é uma característica da própria malha. Para o *framework* GeMA em geral, não há limitação quanto a dimensão dos pontos (1D, 2D, 3D ou mesmo dimensões maiores), embora processos e físicas possam incluir restrições quanto à dimensão dos nós suportados por estas.

Células: São formadas por conjuntos ordenados de nós que definem uma partição do domínio do modelo. Dependendo do tipo de malha, estas células são associadas com funções de forma e regras de integração sendo denominadas de elementos. O número de nós e a organização dos mesmos definem o tipo da célula / elemento.

No exemplo da figura 4.9, a malha é heterogênea e composta por quadriláteros e triângulos quadráticos, com quadriláteros definidos por 9 nós, 8 em sua borda e um central, e triângulos definidos por 6 nós.

Grupos de células: Células da malha podem ser organizadas em grupos e uma malha pode conter diversos grupos. Estes não precisam ser disjuntos e podem conter células heterogêneas. No modelo de análise de bacias a ser apresentado no capítulo 6, por exemplo, grupos de células são utilizados para separar os elementos pertencentes a cada camada do modelo.

Bordas: São formadas por conjuntos de arestas em 2D ou faces em 3D, sendo utilizadas na definição de condições de contorno do modelo. Cada aresta ou face é definida pelo identificador da célula que a contém e pelo índice da aresta / face no elemento.

Regras de integração: Definem a localização e o peso dos pontos de integração associados a cada tipo de elemento da malha. Cada malha pode estar associada a múltiplas regras de integração.

Regras de integração são associadas à malha e não diretamente aos elementos desta para minimizar o número de informações armazenadas por elemento. Como cada malha está associada a múltiplas regras de integração, físicas distintas podem utilizar conjuntos de regras distintos. Desta forma, por exemplo, uma física de cálculo de temperatura pode usar menos pontos de integração, para a mesma malha, do que uma física de cálculo de tensões.

Se eventualmente for necessário associar regras distintas para elementos de um mesmo tipo pertencentes à malha, é possível adicionar ao *framework* um conceito de "sub-tipo" e associar regras à estes sub-tipos.

4.2.2 Dados associados a malhas

Nós, células e elementos da malha podem ser associados a conjuntos de dados definidos pelo usuário durante a descrição do modelo de simulação (seção 4.7). Estes dados podem ser separados em três categorias distintas:

Variáveis de estado: Representam os graus de liberdade do modelo, correspondendo às grandezas calculadas pela simulação. São entidades globais e portanto podem estar associadas a várias malhas. Por representarem graus de liberdade do modelo, variáveis de estado estão associadas exclusivamente a nós da malha.

Em análises por elementos finitos, os elementos da malha podem ser categorizados como isoparamétricos, superparamétricos ou subparamétricos (figura 4.10). Elementos isoparamétricos são aqueles que utilizam a mesma função de forma para interpolar a geometria do elemento e a variável de campo sendo calculada (Zienkiewicz *et al*, 2005). Se a geometria utilizar uma função de forma mais "refinada" que a variável de campo, o elemento é denominado de superparamétrico, caso contrário, de subparamétrico.

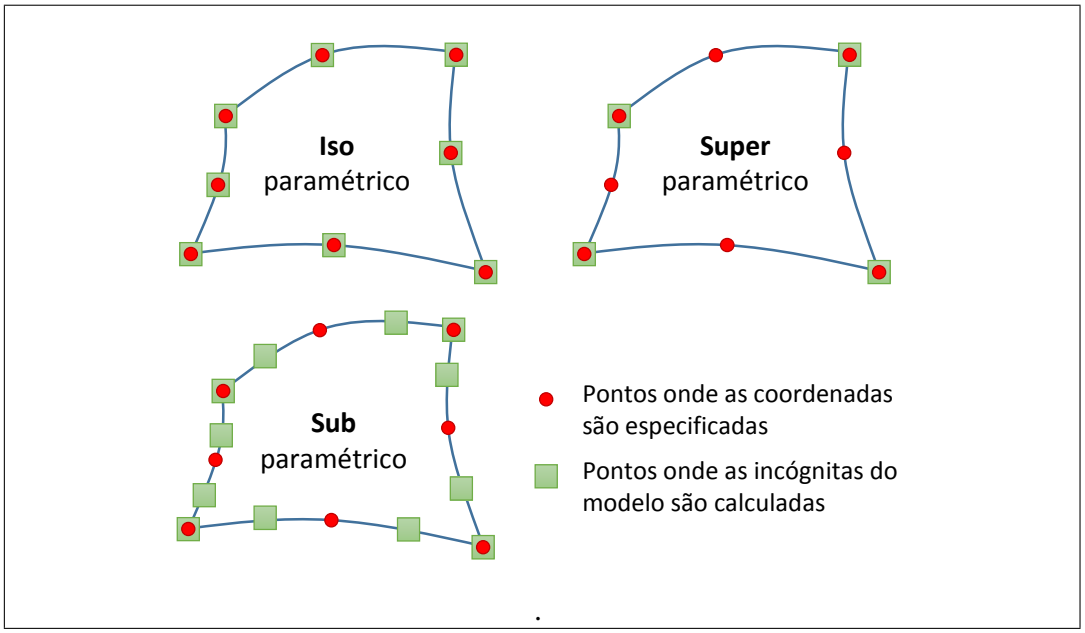


Figura 4.10: Elementos isoparamétricos, superparamétricos e subparamétricos. Adaptado de Zienkiewicz *et al* (2005)

No ambiente GeMA, a associação direta entre nós e variáveis de estado permite o uso de elementos isoparamétricos (mais comuns) e superparamétricos, não suportando o uso de elementos subparamétricos.

Atributos: Representam valores numéricos arbitrários que podem ser associados a nós, células ou aos pontos de integração (pontos de Gauss) de um

elemento. Podem ser usados para armazenar dados do modelo ou para guardar resultados derivados, calculados pelo próprio modelo de simulação.

Conjuntos de propriedades: Representam tabelas de propriedades físicas de materiais. Cada coluna é uma propriedade e cada linha contém valores para todas as colunas, desta forma definindo as múltiplas características do material. Na malha, cada célula armazena o índice de uma linha desta tabela, associando a partição do espaço que a célula representa com seu conjunto de características físicas.

Malhas podem estar associadas a mais de um conjunto de propriedades (uma célula pode estar associada a um conjunto de propriedades térmicas e a outro de propriedades geomecânicas, por exemplo). Desta forma, cada célula armazena um índice para cada conjunto de propriedades associado à malha. De maneira similar às variáveis de estado, conjuntos de propriedades são objetos globais da simulação e podem estar associados a diversas malhas.

Naturalmente, conjuntos de propriedades podem ser utilizados para armazenar também outras tabelas de valores (parâmetros de fabricação, por exemplo) e não apenas propriedades dos materiais.

Propriedades e atributos associados a células são conceitos similares que divergem na forma como são implementados. Se não houvesse preocupação com eficiência no armazenamento de dados, propriedades poderiam ser substituídas por atributos. Atributos são valores armazenados diretamente pelas células. Se uma malha possui 5 atributos e 10 células, 50 valores serão armazenados mesmo que as 10 células compartilhem os mesmos valores de atributos. Propriedades, por sua vez, fazem parte de um conjunto correlacionado de valores. Quando uma célula guarda uma referência para uma linha da tabela de propriedades, automaticamente se associa com todas as propriedades deste material. No mesmo cenário anterior, agora serão armazenados apenas 15 valores (1 valor para cada uma das 5 propriedades e um índice de linha por célula). Em cenários, bastante comuns, onde cada material possui diversas propriedades e diversas células da malha compartilham o mesmo material, o uso de conjuntos de propriedades promove um grande ganho de utilização de memória.

Conjuntos de propriedades possuem uma interface abstrata e são implementados por uma categoria própria de *plugins*. Isto permite a integração com bibliotecas de materiais providas por sistemas pré-existentes.

Apesar de serem entidades distintas, variáveis de estado, atributos e propriedades compartilham uma mesma estrutura básica. Cada um destes dados

pode ser considerado como uma coluna em uma tabela, cujas linhas estão associadas a nós, células, pontos de integração ou tipos de materiais. A figura 4.11 ilustra as três categorias de dados. É interessante notar que em 4.11(a), as variáveis de estado `u` e `T` e o atributo `q` possuem 7 valores já que a malha possui 7 nós. Em 4.11(b), cada uma das propriedades possui apenas 2 valores, uma vez que as células da malha estão associadas a apenas dois tipos de materiais. Em 4.11(c), o atributo `Ro` possui 6 valores já que a malha possui 6 elementos. Finalmente, em 4.11(d) a tabela possui 18 valores correspondentes a cada um dos 3 pontos de Gauss dos 6 elementos.

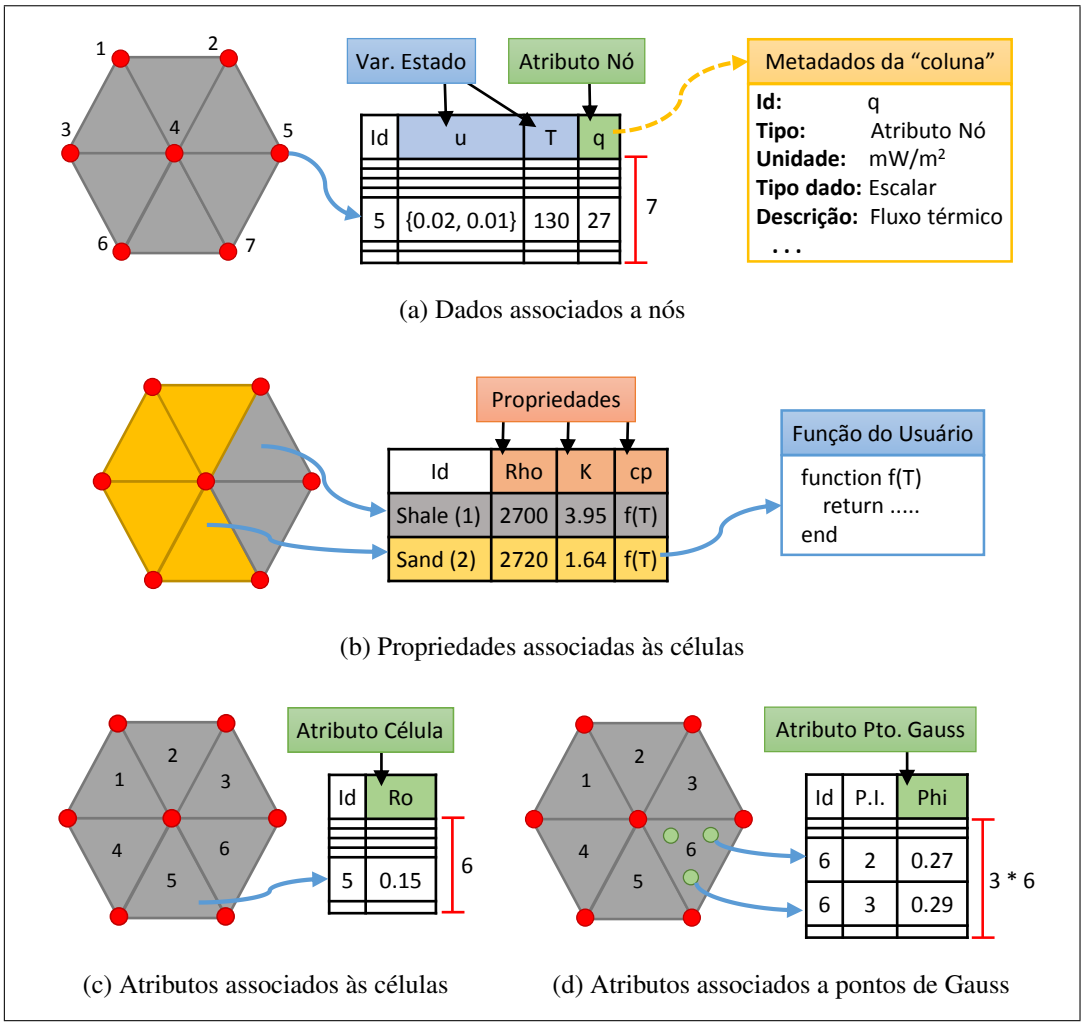


Figura 4.11: Dados associados às entidades da malha. (a) Dados associados a nós; (b) Propriedades associadas às células; (c) Atributos associados às células; (d) Atributos associados a pontos de Gauss.

Independentemente de seu tipo, variáveis de estado, atributos e propriedades possuem um conjunto de metadados associados que definem seu conteúdo, conforme ilustrado na figura 4.11(a) para o atributo `q`. Estes mesmos metadados são utilizados também para descrever as coordenadas dos nós e valores associados às condições de contorno. As seguintes informações são armazenadas:

Identificador: Fornece uma chave utilizada para acesso aos dados.

Descrição: Descrição dos valores armazenados.

Tipo: Define o tipo de informação armazenada (coordenadas, variável de estado, atributo associado a um nó, atributo associado a uma célula, atributo associado a pontos de integração, propriedade ou condição de contorno).

Tipo de dado: Define se o dado armazenado é um valor escalar, um vetor ou uma matriz. Na figura 4.11(a), por exemplo, a variável de estado `u` armazena dados vetoriais.

Dimensão: Define a dimensão associada com dados não escalares (número de linhas para vetores e número de linhas e colunas para matrizes).

Unidade: Define a unidade em que os dados armazenados estão expressos.

Funções: Valor *booleano* indicando se o dado armazenado pode ou não ser expresso por uma função do usuário. Na figura 4.11(b), por exemplo, a propriedade `cp` é definida através de uma função `f(T)` cuja implementação é dada através de uma função em Lua.

Esparso: Valor *booleano* indicando se o dado armazenado possui valores diferentes de um valor padrão para todas as linhas da tabela associada ou apenas para um pequeno subconjunto destas. Nestas situações, a indicação de que o dado é esparso pode ser utilizada para otimizar o armazenamento em memória dos dados. Em particular, situações onde todos os valores são iguais ao padrão podem ser implementadas sem a necessidade de alocação de memória adicional (apenas o dado padrão precisa ser armazenado), algo particularmente útil para propriedades definidas por uma única função do usuário.

Valor padrão: Valor padrão utilizado para dados esparsos, também utilizado como valor inicial dos dados se estes não sofrerem uma inicialização explícita.

Histórico: Define se o conjunto de dados deve armazenar um histórico de valores, bem como a quantidade de estados salvos. Necessário em situações onde o método de cálculo depende não só do valor atual do dado, mas também de valores anteriores do mesmo.

Regra de integração: Define qual conjunto de regras de integração associado à malha deverá ser utilizado para definir o número de pontos de integração por tipo de elemento. Utilizado apenas para atributos associados a pontos de integração.

Formato: Define a formatação (tamanho e número de casas decimais) utilizada quando os valores armazenados são impressos ou exportados em arquivos com formatação textual.

Atributos e propriedades (bem como condições de contorno), além de armazenarem valores escalares, vetoriais ou matriciais, podem armazenar também funções fornecidas pelo usuário[3].

Funções do usuário podem ser implementadas através de funções em Lua, definidas no modelo de simulação, ou através de métodos em C++, fornecidos por *plugins* criados pelo usuário. Ambas podem receber parâmetros avaliados automaticamente pelo *framework*. O conjunto de parâmetros passados para cada função é definido no modelo de simulação através de uma lista contendo o nome das variáveis de estado, atributos e/ou propriedades desejadas. Esta lista pode conter ainda nomes pré-definidos com significados particulares. Os parâmetros disponíveis dependem do tipo de função.

Funções que retornam dados associados a nós podem receber como parâmetros:

- Valores de variáveis de estado associadas ao nó;
- Valores de outros atributos associados ao nó;
- As coordenadas cartesianas do nó sobre o qual a função está atuando;
- O tempo atual de simulação;

Funções que retornam dados associados às células ou bordas de células (para funções associadas com condições de contorno) podem receber como parâmetros:

- Valores de propriedades associadas à célula;
- Valores de atributos associados à célula;
- Valores de atributos armazenados em pontos de integração associados à célula;
- Valores de variáveis de estado;
- Valores de atributos associados a nós;
- As coordenadas cartesianas do ponto de integração atual;
- As coordenadas naturais do ponto de integração atual;
- O tempo atual de simulação;

[3] Variáveis de estado conceitualmente não podem ser associadas a funções, uma vez que seu valor advém da própria simulação.

Quando uma função avaliada sobre uma célula solicita receber como parâmetro valores definidos sobre nós (variáveis de estado ou atributos associados com nós da malha), a avaliação da função deve ocorrer em um ponto específico da célula, geralmente um ponto de integração. Nesta situação, o *framework* automaticamente interpola os valores nodais da célula no ponto de integração e passa este valor interpolado como parâmetro para a função do usuário. No exemplo da figura 4.11(b), a capacidade térmica `cp` é definida como uma função da temperatura `T`. Quando o valor de `cp` é consultado sobre um ponto de integração de um dos elementos da malha, o valor da variável de estado `T` é interpolado a partir dos valores encontrados nos nós do elemento e este valor é passado como parâmetro para a função `f(T)`. Todo este processo é feito automaticamente pelo *framework* de maneira transparente. Em particular, este processo pode ser recursivo pois parâmetros passados para uma função do usuário podem ser também outras funções do usuário.

A figura 4.12 ilustra o armazenamento de uma variável de estado `T` para a qual o armazenamento de estados anteriores foi requisitado. Cada estado salvo está associado a um momento da simulação e guarda o valor de `T` naquele momento.

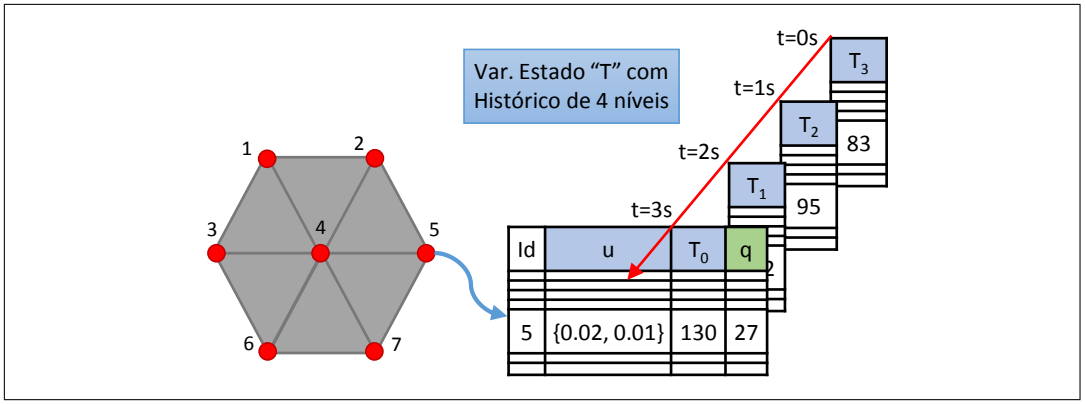


Figura 4.12: Dados com histórico associado.

Há dois tipos de histórico que podem ser associados a um dado. No primeiro deles, o número de estados salvos é ilimitado. A cada vez que o *script* de orquestração ou algum processo solicita o salvamento do estado atual, uma cópia dos dados atuais é criada. Na segunda forma, o histórico é fixo em um número pré-determinado de estados, definido quando da criação do conjunto de dados. Esta forma é adequada para situações onde os métodos de cálculo dependem tanto dos valores atuais de uma variável, quanto de um número fixo de valores anteriores. Neste modo, quando um pedido de salvamento é feito, os valores atuais são repassados para o estado anterior, estes para o anterior ao mesmo e assim subsequentemente até que o último estado salvo seja descartado[4].

[4]Na prática a implementação atual não efetua várias cópias dos dados, de um estado para outro. Ela apenas gerencia qual conjunto de dados corresponde a qual estado salvo.

O salvamento de estados e o tipo de histórico armazenado podem ser habilitados individualmente para cada variável de estado ou atributo associado à malha. Propriedades, por serem globais e compartilhadas entre malhas, não possuem histórico. Quando uma variável de estado ou um atributo com histórico é especificado como um parâmetro para uma função do usuário, é possível definir qual estado será passado como parâmetro.

O modelo de dados descrito permite a associação de dados genéricos, definidos pelo usuário, e com semântica variável às diversas entidades das malhas do modelo. Somando-se a estas características o fato de que estes dados podem ser definidos através de funções, podem ter diversas versões armazenadas ao longo da simulação e ainda podem estar expressos em unidades definidas pelo usuário, faz-se necessária a criação de um mecanismo que:

1. Permita o acesso aos dados de maneira simples e única;
2. Permita que a consulta aos valores armazenados ocorra de maneira exatamente igual, quer os mesmos sejam constantes ou funções do usuário;
3. Independa da localização dos dados acessados;
4. Independa da versão (histórico) sendo acessada;
5. Possua uma mesma forma de acesso para dados escalares, vetoriais ou matriciais e
6. Permita a conversão automática entre a unidade em que os dados estão armazenados e a unidade de trabalho desejada pelo usuário do *framework*.

Para atender a esta necessidade, o *framework* GeMA introduz objetos denominados de *accessors*, um componente central da arquitetura proposta. Ao serem criados pela malha, estes objetos armazenam uma referência para o dado a ser acessado, bem como a unidade de trabalho e a versão desejadas. Estes objetos podem ser armazenados e continuam válidos mesmo se a malha subjacente for alterada.

Conceitualmente, um *accessor* contém apenas dois métodos, `value()` e `setValue()`, responsáveis por consultar e atualizar valores de dados (na prática sua interface é mais complexa, conforme será explorado na seção 4.2.3).

Quando o método `value()` é chamado, recebendo como parâmetro o índice do nó ou elemento a ser consultado, o valor apropriado é obtido da estrutura de dados onde os valores estão armazenados. Se este for uma função, a mesma é avaliada automaticamente, de forma transparente para o chamador. O valor obtido é então convertido da unidade de armazenamento para a unidade de trabalho e retornado pela função `value()`.

É importante observar que *accessors* fazem parte da interface abstrata usada na definição das malhas, sendo também interfaces. Sua implementação concreta depende da maneira como os dados são armazenados.

Na implementação padrão que acompanha o *framework*, a conversão de unidades é feita apenas se as unidades fonte e destino existirem e forem distintas. O custo do tratamento de unidades se a conversão não for necessária limita-se a uma avaliação condicional. Se a conversão for necessária, a mesma é feita através de uma transformação linear com coeficientes pré-armazenados. Esta possibilidade de conversões de unidade com baixo custo permite que os usuários, ao construírem o modelo, utilizem as unidades mais adequadas aos dados existentes. As físicas, por sua parte, determinam em que unidade desejam receber os dados usados nos cálculos e recebem os valores já convertidos, se necessário, de maneira simples e transparente.

Conforme detalhado na seção 4.1, os diversos tipos de malhas suportados são declarados através de interfaces abstratas, cuja implementação ocorre de fato em *plugins*. O suporte ao armazenamento de dados associados a nós, células e elementos é parte integrante destas interfaces abstratas e deve existir em qualquer *plugin* que deseje estender o ambiente GeMA com uma nova implementação de malhas.

Embora cada *plugin* seja livre para implementar este suporte da maneira que lhe convier, o *framework* provê rotinas padrão que incluem o suporte a todas as características apresentadas, incluindo a criação de *accessors* padronizados e o tratamento de funções do usuário. Seu uso viabiliza o desenvolvimento de novos *plugins* de maneira simples, sendo a base do suporte a dados oferecida pelo *plugin* de malhas que acompanha o *framework*.

4.2.3 Estrutura de classes

A figura 4.13 apresenta uma visão simplificada das principais classes associadas às malhas e demais entidades relacionadas. Classes envoltas em borda dupla são interfaces cuja implementação concreta ocorre em *plugins*. As demais classes também são interfaces, porém especializadas e implementadas dentro do próprio *framework*.

A espinha dorsal do suporte a malhas é dada pelas classes **Mesh**, **CellMesh** e **ElementMesh**, conforme detalhado no início desta seção. A principal funcionalidade da classe **Mesh** é permitir o acesso às coordenadas dos nós da malha e aos dados associados a estes nós. A tabela A.1 (encontrada no apêndice A) apresenta os principais métodos de sua interface.

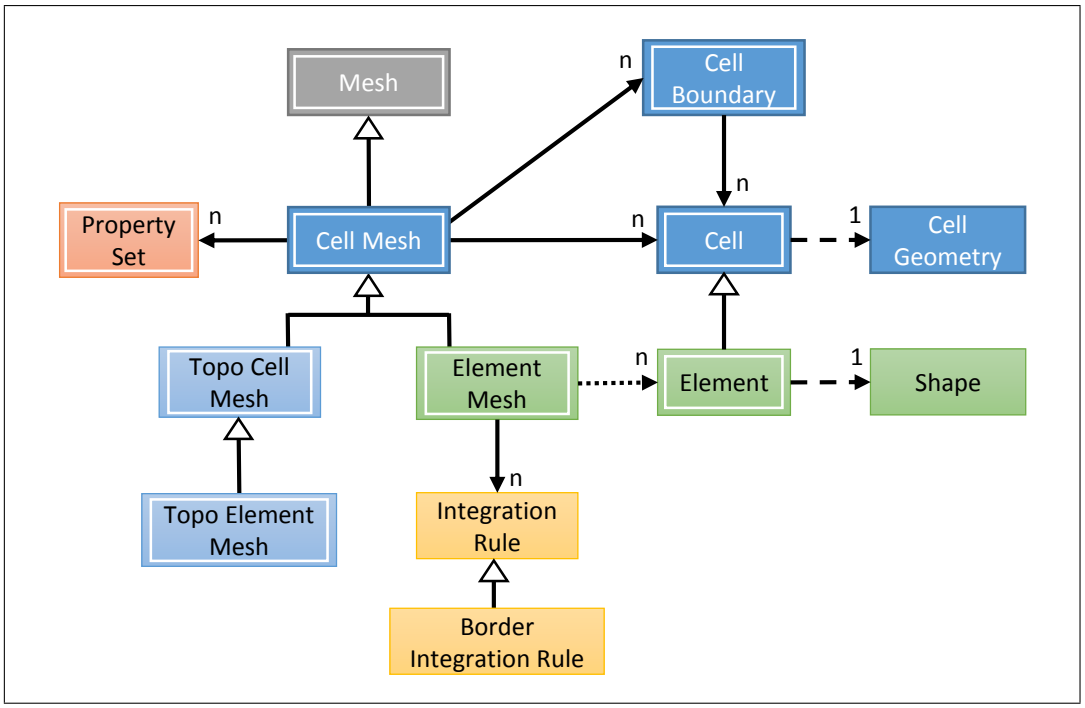


Figura 4.13: Principais classes relacionadas com malhas.

É interessante lembrar que implementações concretas da classe **Mesh** não são obrigadas a armazenar dados sobre as coordenadas de um nó. Se a malha subtendida pelo conjunto de nós for regular (como no método de diferenças finitas, por exemplo), a implementação subjacente pode calcular as coordenadas dos nós a partir de seu índice através de um *accessor* customizado.

A classe **CellMesh** armazena um conjunto de células representadas pela classe **Cell**. Seu objetivo é prover acesso às células da malha, seus dados associados e demais entidades relacionadas (tabela A.2). Conjuntos de propriedades e bordas da malha são representados na figura 4.13, respectivamente, pelas classes **PropertySet** e **CellBoundary**.

A interface da classe **Cell** (tabela A.3) tem como objetivo permitir que implementações concretas armazenem, por célula, o mínimo de informações possível, isto é, a lista de nós da célula e a lista de propriedades associadas (cujo tamanho é igual ao número de conjuntos de propriedades associados à malha).

Na implementação padrão do *plugin* de malhas, a classe concreta que representa uma célula é na verdade um *template* em C++ que tem como parâmetros o número de nós da célula e seu tipo. Desta forma, estas informações são incluídas no tipo da classe e não precisam ser armazenadas por célula da malha. Seguindo o princípio adotado de minimalidade, células não armazenam um ponteiro para a malha. Isto significa que para obter as coordenadas dos nós de uma célula, por exemplo, além da célula, é necessário um *accessor* para consulta às coordenadas, o que, na prática, não se mostra uma limitação.

Informações sobre a geometria de uma célula, tais como seu número de faces, arestas de uma face, etc, são fornecidas pela classe **CellGeometry** e não pela própria célula. Esta classe também é responsável por operações tais como verificar se um ponto pertence a uma célula, cálculo de seu centroide, etc. Esta separação existe pois estas rotinas são gerais (dependem apenas do tipo de célula) e podem ser implementadas no *framework*.

A classe **ElementMesh** estende a classe **CellMesh** de forma que esta, ao invés de armazenar células, armazena elementos representados pela classe **Element**. Além disso, armazena ainda os diversos conjuntos de regras de integração utilizados pelos elementos da malha. As tabelas A.4 e A.5 apresentam os principais métodos de sua interface.

Cada elemento está associado a uma função de forma, representada pela classe **Shape**. De maneira similar à classe **CellGeometry**, subclasses estendendo a interface **Shape** são implementadas pelo próprio *framework* e não pelos *plugins* de malha. As interfaces das classes **Shape** e **IntegrationRule** são apresentadas na seção 4.4.

Todas as interfaces de malha possuem em comum o uso de *accessors* para consulta e alteração de dados, incluindo o acesso às coordenadas dos nós da malha. *Accessors* são interfaces abstratas e a tabela A.6 apresenta os principais métodos da classe **ValueAccessor**. A figura 4.14 apresenta as relações entre malhas, *accessors* e o armazenamento de dados.

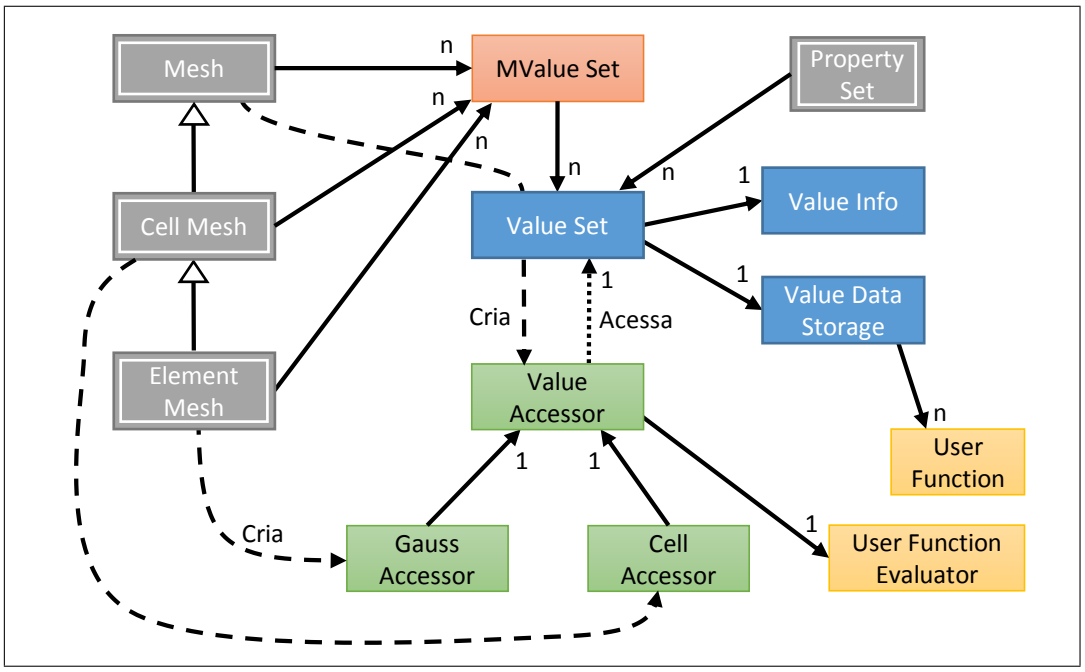


Figura 4.14: Implementação padrão do conceito de *accessors*.

Para os métodos de um **ValueAccessor**, responsável por tratar dados associados aos nós, cada nó individual é identificado por seu índice na malha,

equivalente ao índice ocupado pelos dados do nó em seu armazenamento em memória.

O tratamento dos dados associados a uma célula da malha (atributos e propriedades) é efetuado através de um **CellAccessor**. Cada **CellAccessor** possui um **ValueAccessor** responsável pelo efetivo acesso aos dados. Ao invés de receberem um índice, os métodos de um **CellAcessor** recebem como parâmetro uma célula, que é internamente transformada no índice requerido pelo **ValueAccessor**. Tratamento similar é feito para dados associados aos pontos de integração de um elemento através da classe **GaussAccessor**. Neste caso, a chave para acesso aos dados é composta pelo elemento e pelo índice do ponto de integração.

A classe **ValueSet** é a responsável por armazenar uma tabela de dados, esparsa ou não, suportando o armazenamento de dados escalares ou multidimensionais, bem como funções do usuário (objetos do tipo **UserFunction**). Sua implementação padrão utiliza formas distintas de tratamento para otimizar os casos mais comuns que podem ser armazenados em um vetor, utilizando estruturas mais complexas apenas quando for necessário o suporte a funções e/ou a dados esparsos. A classe **ValueInfo** é a responsável por armazenar os metadados que definem a estrutura necessária. A classe **UserFunctionEvaluator** fornece os serviços necessários para que um *accessor* efetue chamadas às funções do usuário. A classe **MValueSet** é responsável pelo tratamento de históricos, gerenciando os múltiplos **ValueSets** usados para armazenar cada estado salvo.

A criação de um **ValueAccessor** é feita pelo **ValueSet** que contém os dados a serem acessados, em favor da malha que solicitou esta operação. Quando uma malha de células cria um **CellAccessor**, primeiramente um **ValueAccesssor** é criado e este é, então, passado como parâmetro para o **CellAccessor**.

4.2.4 Múltiplas malhas

Modelos de simulação podem conter múltiplas discretizações espaciais. Dependendo do cenário do problema, estas podem representar porções distintas do domínio espacial, com interseção apenas parcial, em escalas distintas e com tipos de células diferentes. A transferência de dados entre estes pares de malhas pode ser feita através de processos implementados nos *plugins* padrão do ambiente GeMA.

O processo de transferência de dados entre malhas utiliza um mapeamento inverso composto por três passos executados para cada nó da malha de destino. A figura 4.15 ilustra este processo. São eles:

1. Busca, na malha fonte, da célula que contém a coordenada associada ao nó da malha de destino cujo valor deseja-se obter.

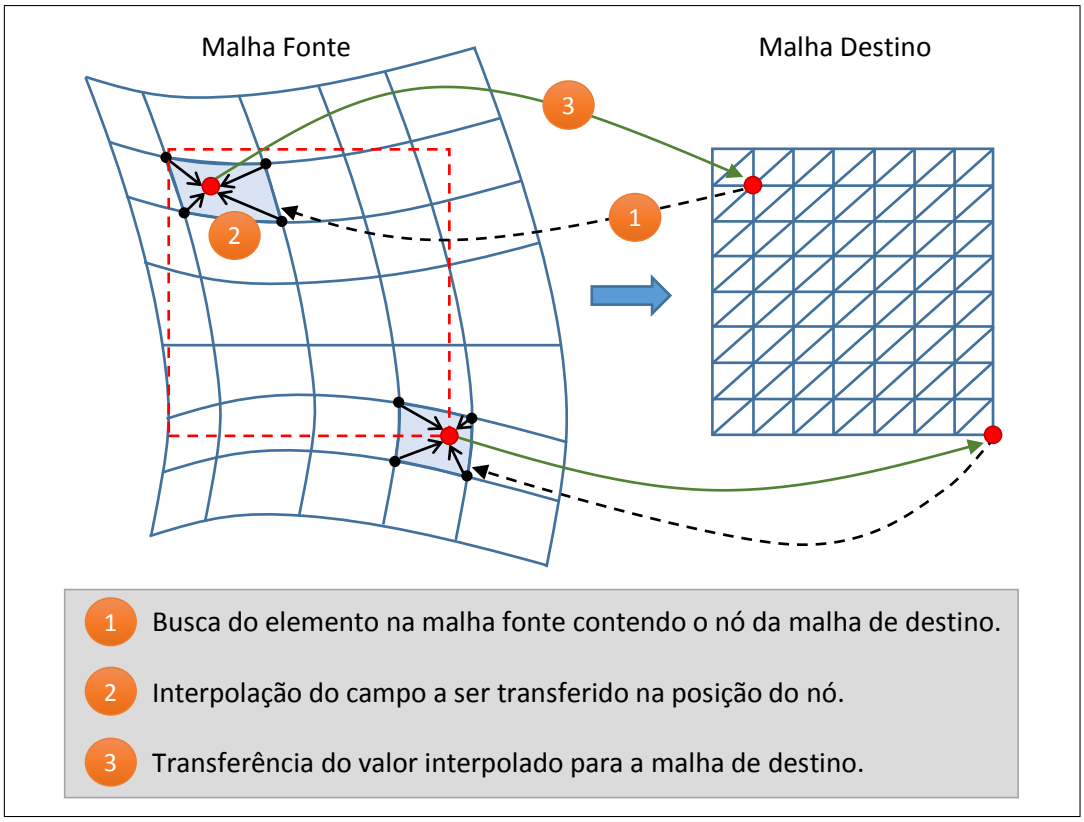


Figura 4.15: Transferindo dados entre domínios distintos.

2. Interpolação dos valores dos campos a serem transferidos (variáveis de estado ou atributos associados aos nós da malha), na coordenada definida pelo nó da malha destino, com base nos valores dos nós da célula encontrada no passo anterior.

3. Transferência dos valores interpolados da malha fonte para a malha de destino, com conversão de unidades se necessário.

Para que este processo seja eficiente, é necessário que o passo de mapeamento da célula que contém uma coordenada espacial seja feito de forma eficiente através do uso de índices espaciais que evitem uma busca em todas as células da malha. A implementação atual do *framework* utiliza um esquema proposto por Scrimieri *et al* (2014), baseado na partição do espaço através de uma grade regular dinâmica, para indexar nós e elementos da malha fonte, otimizando as operações básicas de busca do elemento que contém um ponto e do vértice mais próximo de um ponto. Atualmente, este mapeamento é feito apenas para elementos com função de forma linear.

Quando o nó a ser mapeado não pertence a nenhum elemento da malha fonte, o que pode ocorrer se os domínios forem disjuntos, ou mesmo em situações onde as malhas cobrem conceitualmente o mesmo domínio espacial (mas devido a imprecisões numéricas há descasamentos), seu valor final depende de parâmetros

definidos pelo usuário, podendo assumir um valor padrão, o valor do ponto da malha fonte mais próximo (se a distância entre o nó e este ponto estiver dentro de uma tolerância), um valor extrapolado com base na célula mais próxima, ou mesmo manter-se inalterado.

Outro parâmetro do algoritmo é o método de interpolação utilizado. Geralmente, para malhas de elemento finitos, a interpolação é feita com base na função de forma do elemento. Por outro lado, se a malha for uma malha de células, sem função de forma associada, a interpolação pode ser feita através de técnicas como o inverso da distância ou inverso do quadrado da distância.

O mesmo processo descrito acima, com alguns ajustes, pode ser usado para a transferência de atributos de uma célula ou mesmo atributos armazenados nos pontos de integração de um elemento.

4.3 Orquestração

De uma perspectiva bastante geral, pode-se considerar que a execução de uma simulação é composta pela execução de uma série de processos que trocam dados entre si, cooperando para a obtenção do resultado final desejado. Esta execução coordenada é denominada de orquestração.

A figura 4.16 ilustra a decomposição da simulação em processos para três cenários distintos, de complexidade crescente, onde cada caixa representa um processo. O cenário (a) ilustra uma simulação simples de cálculo em regime permanente da distribuição de temperatura em uma placa. Neste caso, a execução da simulação é composta basicamente pela execução de uma análise por elementos finitos seguida pelo salvamento dos resultados obtidos.

O cenário (b) refere-se a uma simulação para analisar o efeito da alteração das condições de contorno na temperatura de uma bacia sedimentar. Seu primeiro passo consiste em calcular as distribuições de temperatura iniciais da bacia antes das alterações. A seguir, há um laço para cálculo da temperatura e salvamento dos resultados para cada instante de tempo da simulação. Já o terceiro cenário (c) representa uma versão simplificada do processo de análise de bacias 2D apresentado no capítulo 6, onde cada processo envolvido é detalhado. No entanto, é interessante notar o laço utilizado para o tratamento não-linear da condutividade térmica, que é uma função da porosidade das camadas, da própria temperatura e da saturação de óleo e gás na camada.

Estes exemplos apresentam uma pequena amostra da diversidade dos tipos de processos e fluxos de controle que podem ser encontrados. Por ser um *framework* genérico, que se propõe a suportar diversos tipos de simulação e métodos de

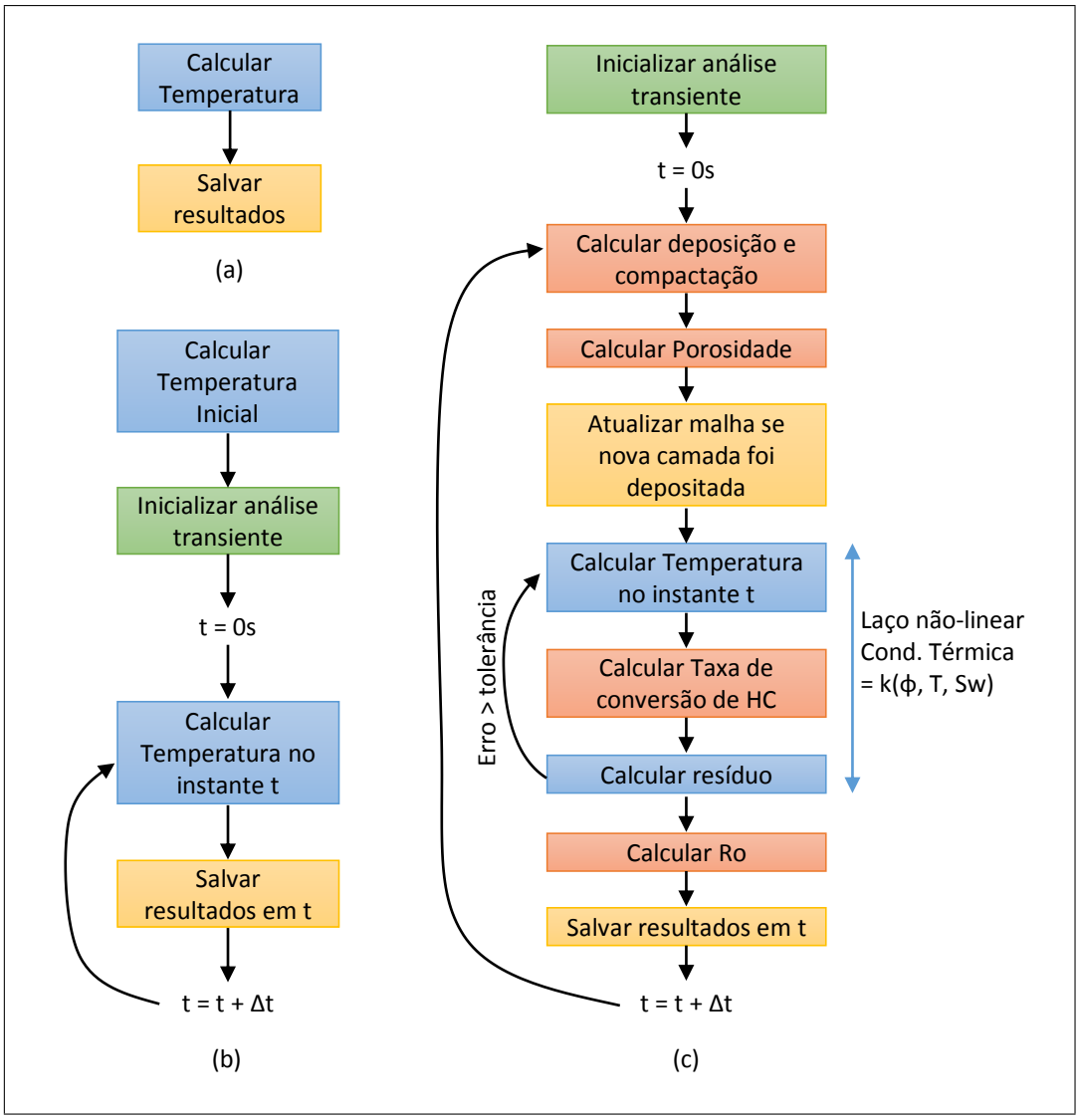


Figura 4.16: Exemplos de processos compondo uma simulação. (a) Simulação para cálculo de temperatura em regime permanente; (b) Simulação para cálculo de temperatura em regime transiente com estado inicial calculado por uma simulação em regime permanente; (c) Versão simplificada de uma simulação de análise de bacias apresentada no capítulo 6.

acoplamento entre físicas, é necessário que o *framework* GeMA permita ao usuário definir sua própria forma de orquestração.

Conforme resumido no início deste capítulo e ilustrado na figura 4.3, a orquestração no ambiente GeMA é definida através de um *script*, escrito na linguagem Lua, que permite ao seu autor definir, através de chamadas de funções e comandos de fluxo próprios da linguagem (laços, condições, definição de funções, etc), a sequência de processos a serem executados ao longo da simulação.

Esta generalidade permite ao usuário descrever os diversos tipos de iterações necessárias em uma simulação multifísica, tais como iterações no tempo, iterações para acoplamento entre físicas e iterações para resolver sistemas não-lineares (ver figura 2.4), bem como sua integração com outros tipos de processos, tais como o

refinamento e a transferência de dados entre malhas. Os *scripts* 1 e 2 ilustram a orquestração correspondente às figuras 4.16 (a) e (b). O *script* referente ao caso (c) será apresentado no capítulo 6.

***Script* 1** *Script* de orquestração para figura 4.16(a)

```
function ProcessScript()
  -- Executa uma análise de elementos finitos, recebendo como
  -- parâmetros uma física e um resolvedor numérico
  fem.solve({'HeatPhysics'}, 'solver')

  -- Salva a variável de estado 'T', associada com a malha 'mesh',
  -- no arquivo resultados.nf de tipo Neutral file ('nf')
  utils.saveMeshFile('mesh', './resultados.nf', 'nf', 'T')
end
```

***Script* 2** *Script* de orquestração para figura 4.16(b)

```
function ProcessScript()
  -- Calcula a temperatura em regime permanente
  -- obtendo o estado inicial em t = 0
  fem.solve({'HeatPhysics t<0'}, 'solver')

  -- Cria o arquivo que irá receber os resultados
  -- salvos: 'T', 'Tana' e 'Err' e adiciona a
  -- solução em t = 0 ao mesmo
  local file = utils.prepareMeshFile('mesh', './resultados.nf', 'nf',
                                     {'T', 'Tana', 'Err'})
  utils.addResultToMeshFile(file, 0.0)

  -- Passo de tempo de 0.01 milhões de anos para
  -- uma simulação de 10 milhões de anos
  local dt         = 0.01
  local secsPerMYr = 60*60*24*365.25*1e6
  local nsteps     = 10/dt
  setCurrentTimeUnit('Myr')

  -- Inicializa análise transiente
  local solver = fem.initTransientSolver({'HeatPhysics t>0'},
                                         'solver')
  -- Loop temporal
  for i=1, nsteps do
    -- Calcula temperatura após um passo de tempo = dt
    fem.transientStep(solver, dt * secsPerMYr)

    -- Atualiza o tempo atual da simulação
    setCurrentTime(i*dt)

    -- Adiciona resultados ao arquivo
    utils.addResultToMeshFile(file, i*dt)
  end
  utils.closeMeshFile(file)
end
```

A figura 4.17 apresenta algumas categorias de processos. Atualmente, o ambiente GeMA implementa processos específicos para:

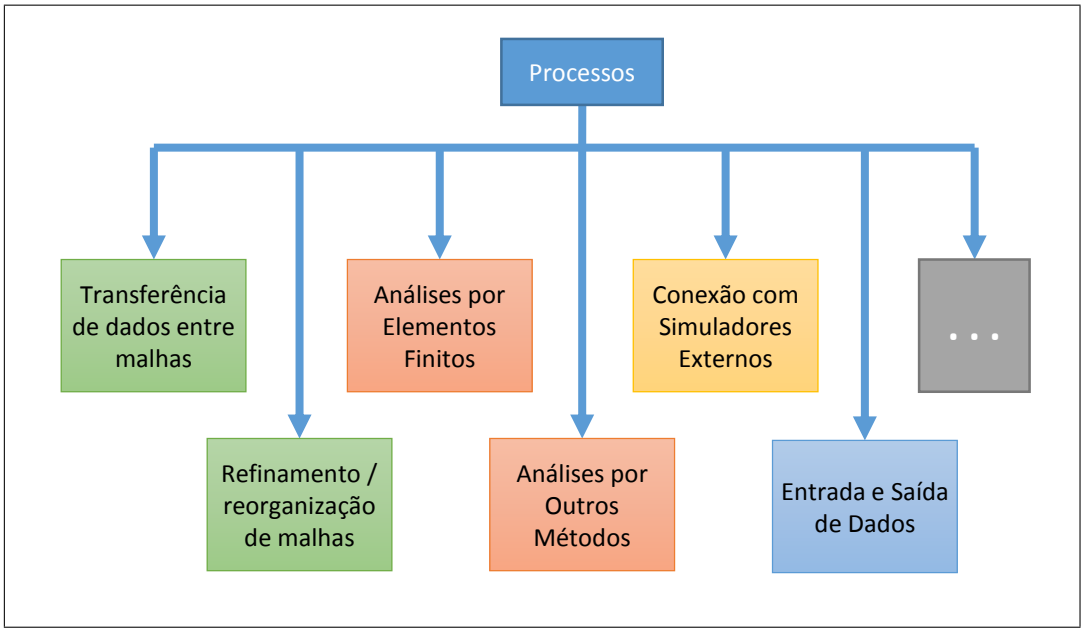


Figura 4.17: Categorias de processos.

- Execução de análises por elementos finitos (detalhados na seção 4.3.1);
- Transferência de dados entre malhas (detalhados na seção 4.2.4);
- Leitura e escrita de arquivos com dados de malha (detalhados na seção 4.6);
- Escrita de informações em arquivos de *log*;

Processos criados pelo usuário podem ser escritos tanto em C++ quanto em Lua. Os primeiros são implementados por meio de *plugins* que publicam os processos oferecidos através de funções que podem ser chamadas pelo *script* de orquestração. Já os segundos são representados por funções escritas pelo usuário diretamente em Lua com acesso a dados e funcionalidades providos pela biblioteca base em C++.

A integração com simuladores externos através da troca de arquivos pode ser feita diretamente através de processos escritos em Lua, sem a necessidade de novos *plugins* específicos. Estes processos de integração podem ser construídos através do uso dos processos existentes para leitura de dados, das funcionalidades padrão da linguagem para execução de programas externos e, se necessário, da transferência de dados entre malhas.

Cabe salientar que apesar de ser interpretado, o custo da execução do *script* de orquestração é geralmente irrelevante, uma vez que praticamente todo o tempo de simulação é gasto na execução dos processos e não na coordenação entre estes.

A seção 4.3.1 detalha o processo de suporte às análises por elementos finitos. A adoção de outros métodos de discretização, tais como os métodos de volumes finitos e diferenças finitas, requer a construção de outros processos de suporte, nos mesmos moldes descritos a seguir.

4.3.1 Processo para análise por elementos finitos

Conforme descrito no capítulo 2, em simulações por elementos finitos a matriz global que define o sistema de equações a ser resolvido é composta por contribuições locais de cada elemento da malha. Estas matrizes locais, por sua vez, são obtidas através da forma discreta da equação a ser resolvida, aplicada a cada elemento.

Os passos necessários a uma análise por elementos finitos são basicamente os mesmos independentemente da equação a ser resolvida. O que distingue, por exemplo, uma análise de tensões de uma análise de temperatura são as equações resolvidas para a obtenção das matrizes locais de cada elemento.

No *framework* GeMA, este conjunto geral de passos compõe o processo de análise por elementos finitos. Já as rotinas para cálculo das matrizes locais são denominadas de físicas, uma vez que estão intimamente ligadas ao problema físico sendo resolvido.

De uma maneira geral, físicas e processos de análise possuem uma relação simbiótica. Um processo de análise depende das físicas, responsáveis por fornecer as equações a serem resolvidas. Estas, por sua vez, devem implementar uma interface definida pelo processo de análise, cabendo salientar que métodos de análise distintos podem definir interfaces distintas. Esta seção concentra-se no processo de análise por elementos finitos. A interface para físicas voltadas à análises por elementos finitos é detalhada na seção 4.4.

A figura 4.18 ilustra as principais entidades envolvidas em uma análise por elementos finitos. São elas:

- A malha sobre a qual a análise será calculada;
- Os graus de liberdade a serem calculados, representados pelas variáveis de estado associadas;
- O conjunto de físicas que fornece as equações a serem resolvidas;
- O resolvedor numérico usado para solução do sistema de equações gerado (ver seção 4.5);
- As condições de contorno necessárias para garantir que as equações analisadas possuam solução única;
- As condições iniciais do problema em análises transientes.

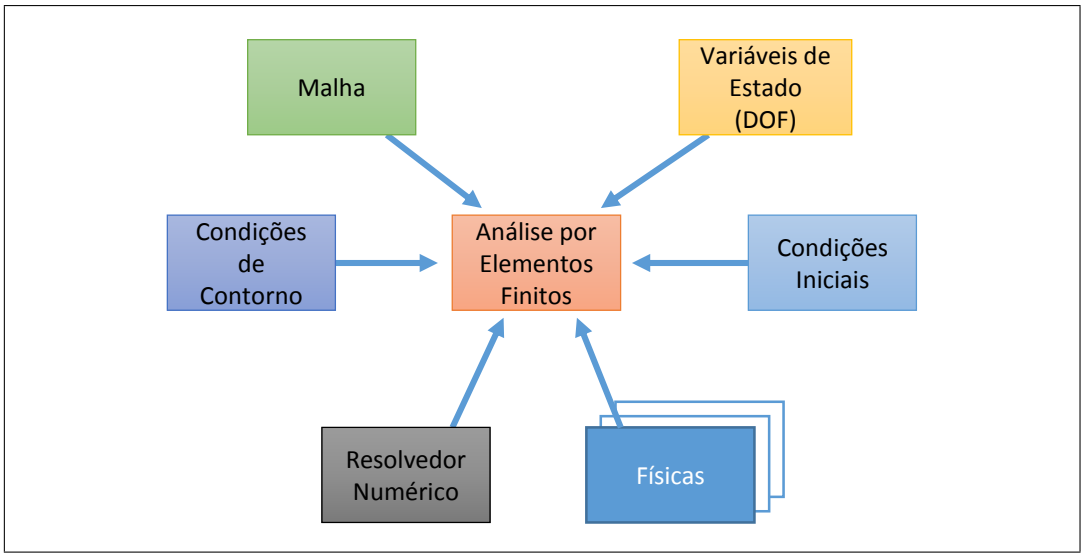


Figura 4.18: Processo de análise por elementos finitos.

Revisitando o *script* 1 podemos ver que o processo de análise por elementos finitos, identificado na linha 4 pela chamada à função `fem.solve()`, recebe como parâmetros uma lista de físicas e um resolvedor numérico, todos representados por seus nomes. A malha, as condições de contorno e as variáveis de estado estão associadas às físicas e portanto não precisam ser passadas como parâmetros. Condições iniciais, quando necessárias, são buscadas dos valores associados à malha antes da execução do processo.

A solução de problemas multifísicos pode seguir diversas estratégias dependendo do tipo de acoplamento entre as físicas do problema (ver capítulo 2). Para problemas fracamente acoplados, uma possível estratégia de solução consiste em iterar por cada física isoladamente até que a convergência seja obtida. Já para problemas fortemente acoplados, pode ser necessário que as físicas sejam compostas em um único sistema de equações resolvidas em conjunto.

No primeiro caso, cada chamada ao processo de solução por elementos finitos recebe uma única física como parâmetro. Já no segundo, pode ser interessante separar o cálculo das matrizes locais em diversas físicas que cooperam entre si para criar a matriz local de cada elemento.

Considerando, por exemplo, o acoplamento entre o cálculo de tensões e o cálculo de temperatura, sua solução pode ser dada através de uma única física GeMA[5], que monta uma matriz local acoplando deslocamentos e temperaturas. Outra solução, porém, pode ser composta por três físicas: a primeira contribui apenas com termos relacionados ao cálculo de deslocamentos, a segunda com

[5]Neste trabalho, o termo "física" é utilizado tanto para se referenciar a um fenômeno físico quanto à entidade usada pelo *framework* para prover equações para um processo. Em geral o contexto deixa claro a acepção correta. Quando for necessário efetuar uma distinção entre ambos, o termo "física GeMA" será usado para referir-se ao objeto do *framework*.

termos relacionados à temperatura e a terceira com o acoplamento entre ambos. A figura 4.19 ilustra estas duas possibilidades.

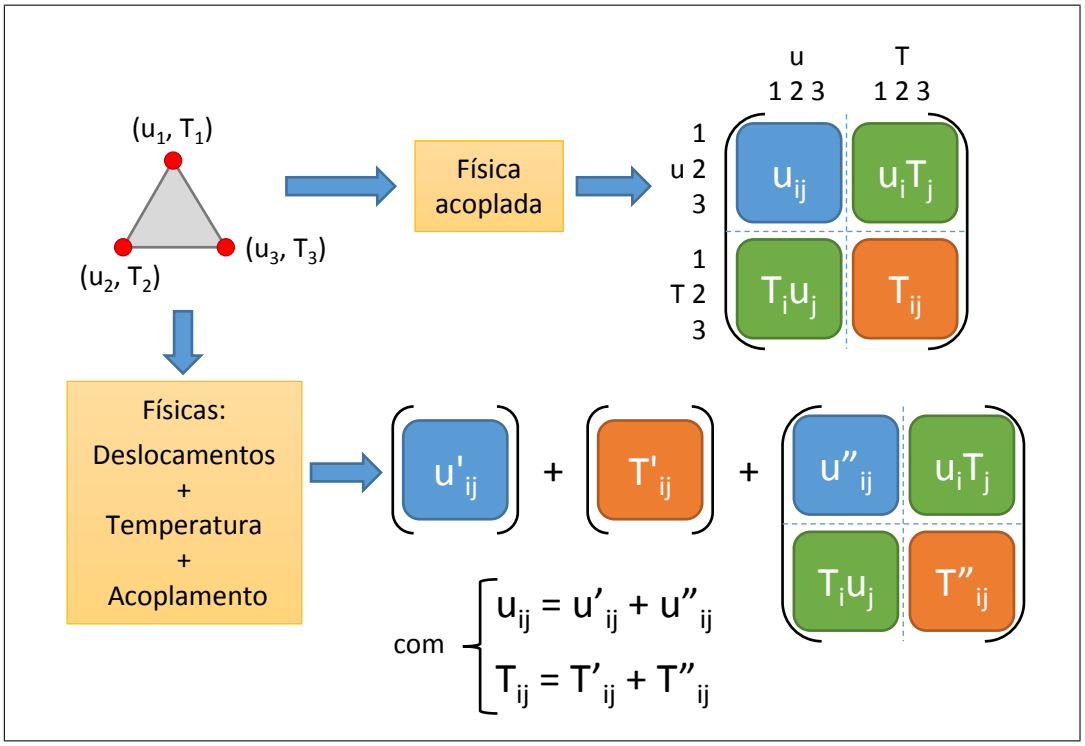


Figura 4.19: Decomposição de uma física acoplada.

Claramente, a segunda opção privilegia o reuso de físicas, uma vez que as mesmas físicas individuais de tensão e temperatura podem ser utilizadas tanto para análises com uma única física quanto para análises acopladas. Mesmo em análises compostas por um único modelo físico, pode ser interessante quebrar as equações em várias físicas GeMA visando o reuso de alguns destes 'componentes' quando mais de uma estratégia de solução for possível. Esta quebra também pode simplificar as implementações, facilitando a manutenção futura do código.

Na solução de um problema linear em regime permanente, o sistema de equações gerado pelo método de elementos finitos é do tipo $K.x = f$, onde K é conhecida como a matriz de rigidez do sistema[6], x são os graus de liberdade a serem calculados e f é o vetor de forças. Os passos principais utilizados pelo processo de análise por elementos finitos do *framework* GeMA para a composição deste sistema e para sua solução são ilustrados na figura 4.20 e detalhados a seguir.

1. Inicializar o **Assembler** responsável por gerenciar os graus de liberdade do sistema. Este objeto é responsável por traduzir graus de liberdade locais a um elemento em seus índices globais, adicionando a matriz e o vetor locais K_e e

[6]Neste trabalho, por falta de uma alternativa melhor e seguindo o exemplo de Felippa (2004), a matriz K será sempre denominada de matriz de rigidez, mesmo para aplicações não relacionadas à mecânica estrutural, de onde o termo se origina.

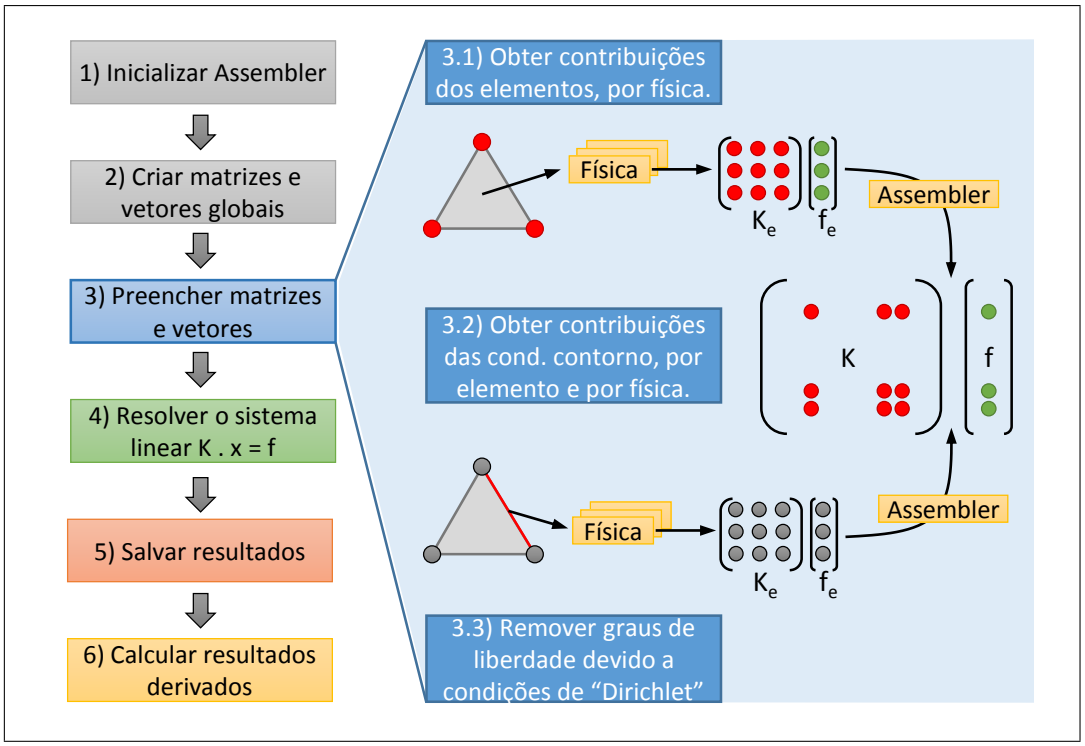


Figura 4.20: Passos principais de uma análise por elementos finitos.

f_e, retornados pelas físicas para cada elemento da malha, na matriz e no vetor globais K e f.

O *assembler* adotado é capaz de trabalhar com elementos heterogêneos. Para cada tipo de elemento presente na malha, cada um de seus nós pode estar associado a conjuntos distintos de graus de liberdade. Este tipo de *assembler* é chamado por Felippa (2004) de "MET-VFC (Multi Element Type - Variable Freedom Configuration)", e permite, por exemplo, que uma física que acopla o cálculo de temperaturas e deslocamentos para elementos quadráticos, calcule os deslocamentos em todos os nós do elemento, mas calcule temperaturas apenas em seus vértices (figura 4.21). No ambiente GeMA, cada física informa ao *assembler*, para cada tipo de elemento suportado, quais os graus de liberdade associados a cada nó do elemento.

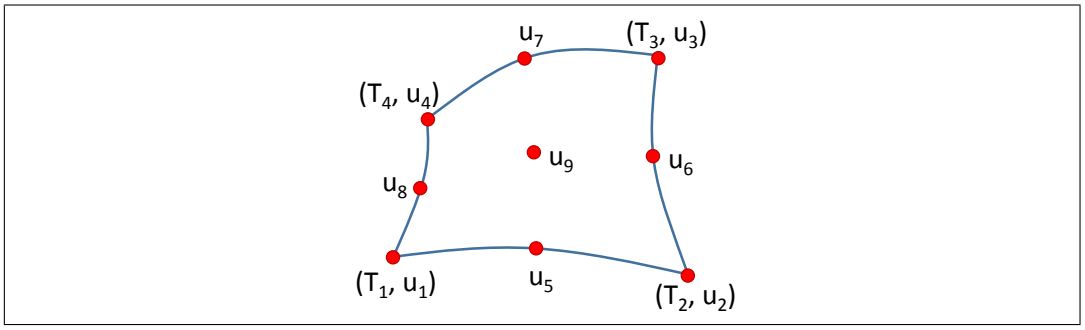


Figura 4.21: Graus de liberdade por nó de um elemento.

2. Criar as matrizes e vetores globais, K e f. O formato da matriz K (completa ou esparsa e, se esparsa, sua organização) depende do resolvedor numérico utilizado (ver seção 4.5).

3. Preencher a matriz K e o vetor f com as contribuições de cada elemento da malha e de cada condição de contorno. Em um primeiro passo, cada elemento da malha é percorrido. Para cada elemento, cada física associada ao processo é consultada para que esta retorne sua contribuição para a matriz local K_e e para o vetor local f_e. Com auxílio do *assembler*, estes elementos locais são somados à matriz de rigidez global K e ao vetor de forças global f.

   No segundo passo, para cada física, suas condições de contorno são percorridas. Para cada condição de contorno, os elementos envolvidos são visitados e a física é consultada para obtenção das contribuições locais do elemento para esta condição de contorno. Estas contribuições são, então, combinadas na matriz global pelo *assembler*.

   No terceiro e último passo, condições de contorno de "Dirichlet", fixando valores de alguns graus de liberdade do modelo, são aplicados à matriz global com ajuste das equações.

4. Resolver o sistema linear $K.x = f$ utilizando o resolvedor numérico fornecido.

5. Salvar os resultados calculados em x nas variáveis de estado correspondentes aos graus de liberdade.

6. Calcular outros resultados, derivados dos graus de liberdade, tais como tensões e fluxos térmicos.

Problemas transientes e problemas não lineares possuem estrutura bastante semelhante à descrita acima, envolvendo normalmente o tratamento de matrizes adicionais, que também devem ser preenchidas pelas físicas, e laços envolvendo os passos 3 a 6. Análises transientes podem levar, por exemplo, a sistemas de equações do tipo:

$$C.\frac{\partial x}{\partial t} + K.x = f \tag{4-1}$$

Esta equação, quando discretizada no tempo através de um método de diferenças finitas implícito transforma-se em (Lewis *et al*, 2004):

$$(C + \Delta t.K).x^{n+1} = C.x^n + \Delta t.f^{n+1} \tag{4-2}$$

A equação 4-2 permite calcular o valor de x no tempo $n + 1$ a partir de seu valor em n através da solução de um sistema linear. Pode-se observar que os termos do lado esquerdo da igualdade são uma matriz multiplicada pelo vetor x a ser calculado no tempo $n + 1$, e o lado direito da igualdade resume-se a um vetor.

Problemas deste tipo podem ser implementados no ambiente GeMA através de novos processos que estendem a classe base de análise por elementos finitos. Todas as rotinas de inicialização e preenchimento de matrizes e vetores podem ser reaproveitadas, bastando ao novo processo ajustar os passos globais necessários e solicitar à física que preencha também a matriz C. Outros métodos de discretização temporal (totalmente explícito, ou Crank-Nicolson, por exemplo) levam a situações semelhantes. O mesmo ocorre no tratamento de sistemas não-lineares resolvidos pelo método de Newton–Raphson, onde uma matriz tangente deverá ser calculada.

Para auxiliar estes novos tipos de processos, o *assembler* utilizado é capaz de tratar um conjunto de matrizes definido pelo usuário. Da mesma maneira, a interface de comunicação com as físicas (apresentada na seção 4.4) permite que estas preencham um número arbitrário de matrizes e vetores por elemento.

Nestes cenários, onde cada física precisa preencher mais de uma matriz, pode ser interessante separá-la em mais de uma classe, visando simplificar a implementação e maximizar o reuso. Pode-se tomar como exemplo uma física de temperatura. Para o tratamento de análises em regime permanente, basta que a física preencha a matriz K. Análises transientes, por outro lado, necessitam também da matriz C (a matriz K utilizada em ambos os casos é a mesma). Ao invés de tornar o código de análise em regime permanente mais complexo para tratar a matriz C, pode-se criar uma nova física para preencher apenas esta nova matriz e usá-la em conjunto com a original.

Em contraponto, a opção com duas físicas é menos eficiente do que a opção com uma única física em situações onde o cálculo de ambas possui elementos em comum. Neste caso (comum), pode-se ter duplicação de cálculo quando cada matriz é calculada por uma física separada. Portanto, é interessante avaliar se o ganho de eficiência de uma implementação única compensa, ou não, a maior complexidade.

4.4 Físicas

Conforme detalhado na seção anterior, processos podem ser utilizados para implementar novos métodos de análise. Estes métodos, em geral, podem ser aplicados a diversos tipos de problemas físicos, e, portanto, é interessante que sua implementação delegue as ações específicas de um problema a outros objetos, denominados no *framework* GeMA de físicas.

Físicas são implementadas através de *plugins* e, por sua própria definição, estão intimamente atreladas a métodos particulares de análise. No caso específico de análises pelo método de elementos finitos, cada física tem como objetivo fornecer matrizes locais a um elemento da malha, para um conjunto específico de equações.

Nos primeiros esforços para construção de códigos para elementos finitos em linguagens orientadas a objetos, descritos em uma série de artigos por Zimmermann *et al* (1992); Dubois-Pèlerin *et al* (1992); Dubois-Pèlerin e Zimmermann (1992), as equações necessárias para criação de matrizes locais são implementadas diretamente por classes derivadas de uma interface denominada *Element*. Estas classes são responsáveis tanto pelo tratamento da geometria do elemento quanto por suas equações, sem uma clara separação entre estes dois conceitos distintos. Este tipo de modelagem pode ser encontrado também em publicações mais recentes, como Patzák e Bittnar (2001).

Para promover a independência entre o modelo de equações e a geometria do elemento, o sistema FEMOOP (Martha e Junior, 2002) e o *framework* TopFEM (Beghini *et al*, 2014) trabalham com os conceitos de *Shape* e *AnalysisModel*. Cada elemento da malha guarda referências para ambos. Objetos *Shape* são responsáveis pelas interpolações da geométria e dos campos associados a um elemento. Objetos do tipo *AnalysisModel* são responsáveis por gerenciar os graus de liberdade do elemento, por fornecer as matrizes locais necessárias e pelo cálculo de resultados derivados.

Físicas no ambiente GeMA são bastante similares a um *AnalysisModel*. A principal diferença encontra-se no fato de que físicas são objetos globais e não estão associados diretamente a um elemento, uma vez que, em um ambiente multifísica, uma mesma malha pode ser compartilhada por várias análises. Este caráter "global" de uma física é similar ao conceito de equações encontrados em Nie *et al* (2010).

É importante fazer uma distinção entre uma física e o *plugin* que a implementa. Físicas são objetos que implementam uma interface bem definida. Um *plugin* de físicas é uma biblioteca dinâmica que exporta objetos do tipo física. Por ser uma biblioteca, esta pode incluir outras classes para implementar conceitos importantes a um determinado domínio. Um exemplo consiste na implementação do conceito de material por um *plugin* de análise não-linear de tensões. O *framework* GeMA implementa o conceito geral de propriedades, mas não o conceito de material, específico de uma física de análise de tensões, delegando sua implementação ao *plugin* apropriado.

4.4.1
Físicas para processos de análise por elementos finitos

A interface para físicas publicada pelo processo de análise por elementos finitos define os métodos necessários para interação entre o processo e a física. Seu objetivo principal é permitir que a física indique quais os graus de liberdade suportados por esta e o cálculo de matrizes e vetores locais por elemento e por condição de contorno. Seus principais métodos estão descritos nas tabelas A.7 e A.8. Há ainda alguns métodos não inclusos nestas tabelas, chamados quando a física é lida do arquivo de configuração do modelo, responsáveis por validar se todas as informações necessárias para os cálculos, tais como atributos e propriedades da malha, estão disponíveis e são compatíveis com os requisitos desta física.

O tratamento de múltiplas matrizes e vetores de dados pela física é feito pelas classes **FemMatrixSet** e **FemVectorSet** que permitem ao processo de cálculo por elementos finitos informar à física quais tipos de matriz devem ser preenchidos.

As implementações das funções que retornam as matrizes e vetores locais de um elemento (`fillElementData()` e demais funções correlatas) contam com o suporte, por parte do *framework*, dos métodos providos pelas classes **IntegrationRule** e **Shape**. Estas interfaces são especializadas por tipo de elemento e contêm, respectivamente, funções para retorno dos pontos de integração de um elemento e funções para cálculo de funções de forma, suas derivadas (em coordenadas naturais e cartesianas) e para cálculo do Jacobiano da transformação. Seus principais métodos estão descritos nas tabelas A.9 e A.10.

Utilizando seus serviços, é possível escrever físicas independentes do tipo de elemento subjacente, de sua dimensão ou da ordem de interpolação. Um exemplo de implementação da função `fillElementData()` é apresentado no *script* 3, e pode ser aplicado indistintamente a quadriláteros e triângulos, lineares ou quadráticos. A mesma função é aplicável também a elementos 3D.

Este exemplo ilustra os cálculos necessários para obtenção da matriz de rigidez e do vetor de forças locais K_e e f_e para uma física de cálculo de temperatura em regime permanente[7]. O modelo de cálculo adotado baseia-se nas equações 4-3 a 4-5 apresentadas por Lewis *et al* (2004), onde λ é o tensor de condutividade térmica do elemento, G é a taxa de geração de calor interna, r seu número de nós, $N = \{N_1, N_2, \cdots, N_r\}$ é o vetor de funções de forma e Ω o domínio do elemento.

[7]Esta função é uma simplificação do código real, já que diversas otimizações podem ser efetuadas para evitar a alocação de matrizes temporárias e para que a obtenção de propriedades seja efetuada fora do laço de integração se estas forem constantes.

***Script* 3** Implementação simplificada da função `fillElementData()` para a classe `HeatPhysics`.

```
1  bool HeatPhysics::fillElementData(const Element* e,
2                                    FemMatrixSet& elemMatrices,
3                                    FemVectorSet& elemVectors)
4  {
5    const Shape* shape = e->shape();
6    int n    = e->numNodes();              // Number of elem nodes
7    int ndim = _nodeAccessor->valueSize(); // Node coord dimension
8
9    // Get a reference for element K and f, filling them with zeros
10   Matrix& elemK = elemMatrices.useMatrix(FemMatrix_K);
11   Vector& elemF = elemVectors.useVector(FemVector_Fe);
12   elemK.zeros();    // Size = (n x n)
13   elemF.zeros();    // Size = (n)
14
15   // Fills the matrix X with node coordinates (size = n x ndim)
16   Matrix X;
17   e->fillNodeMatrix(_nodeAccessor, X);
18
19   // Aux values needed in the calculation:
20   Matrix   B;       // The shape function cartesian partials matrix
21                     // (size = ndim x n)
22   Vector   N;       // The shape functions vector (size = n)
23   CRMatrix cond;    // The conductivity matrix (size = ndim x ndim)
24   double   t = 1.0; // Element thickness for 2D elements
25   double   G;       // Internal element heat generation rate
26   Vector   ip;      // Integration point natural coordinates
27   double   w;       // The Gauss point weight
28   double   detJ;    // The determinant of the Jacobian matrix
29
30   // Gets the element Gauss rule and loops over integration points
31   const IntegrationRule* ir = elementIntegrationRule(e->type());
32   for(int i = 0, nip = ir->numPoints(); i<nip; i++)
33   {
34     // Get the integration point & weight
35     ir->integrationPoint(i, ip, &w);
36
37     // Get needed element properties
38     _kAccessor->matrixValueAt(e, &ip, cond, i);
39     G = _generationRateAccessor->scalarValueAt(e, &ip, i);
40     if(ndim == 2)
41       t = _planeWidthAccessor->scalarValueAt(e, &ip, i);
42
43     // Fill the N (column) vector and Calc B matrix and detJ.
44     shape->shapeCartesianPartials(ip, X, B, &detJ, true);
45     shape->shapeValues(ip, N);
46
47     // Calculate values
48     double c = w * t * detJ;
49     elemK += B.t() * cond * B * c;
50     elemF += N * (c * G);
51   }
52   return true;
53 }
```

$$K_e = \int_\Omega B^T \lambda B \, d\Omega \tag{4-3}$$

$$f_e = \int_\Omega G N^T \, d\Omega \tag{4-4}$$

$$B = \begin{bmatrix} \frac{\partial N_1}{\partial x} & \frac{\partial N_2}{\partial x} & \cdots & \frac{\partial N_r}{\partial x} \\ \frac{\partial N_1}{\partial y} & \frac{\partial N_2}{\partial y} & \cdots & \frac{\partial N_r}{\partial y} \\ \frac{\partial N_1}{\partial z} & \frac{\partial N_2}{\partial z} & \cdots & \frac{\partial N_r}{\partial z} \end{bmatrix} \tag{4-5}$$

Uma descrição completa deste modelo, incluindo termos adicionais referentes às condições de contorno, pode ser encontrada na seção 5.1. É interessante notar como o cálculo do vetor N e da matriz B, contendo as derivadas parciais das funções de forma em relação às coordenadas cartesianas, são efetuados por métodos da classe **Shape**. Estes valores são comuns à maioria dos modelos baseados em elementos finitos.

4.4.2 Prototipação rápida

Em cenários de pesquisa, onde o modelo físico a ser adotado ainda está em desenvolvimento, a possibilidade de implementar e testar novas ideias celeremente pode ser mais importante do que a eficiência destas implementações intermediárias, necessárias para definição do modelo final.

O ambiente Lua, por ser baseado em uma linguagem interpretada completa e de fácil aprendizado, mostra-se um ambiente bastante interessante para prototipação rápida. Seu uso independe da disponibilidade de compiladores, da preparação de um ambiente de desenvolvimento ou de conhecimentos de C++.

É possível criar em Lua tanto novos processos quanto novas físicas. Para permitir a criação de novas físicas de elementos finitos em Lua, o *framework* implementa um *plugin* padrão que disponibiliza uma física *proxy*. Os métodos desta classe simplesmente repassam as chamadas recebidas para um objeto em Lua que deve se encarregar de implementar as funções necessárias.

Para facilitar a prototipação, as validações necessárias quanto à existência e compatibilidade de variáveis de estado, propriedades, atributos e condições de contorno são feitas de maneira declarativa através de uma tabela que informa ao *proxy* os dados necessários para a física. Além disso, *accessors* são disponibilizados

automaticamente pelo *proxy* para uso nas funções de preenchimento das matrizes e vetores locais.

Através do uso de funções criadas dinamicamente, de ambientes globais customizados por função e de meta-métodos (Ierusalimschy *et al*, 2006) que facilitam o acesso aos valores das propriedades e atributos, sem que seja necessário o uso explícito de *accessors*, é possível substituir toda a função `fillElementData()`, apresentada no *script* 3, por, essencialmente, um conjunto de expressões que definem a contribuição de cada ponto de integração para a matriz local. Assim, a função do exemplo passa a ser definida pelas expressões "`B.t() * cond * B * t`" e "`N * t * G`".

Um exemplo completo de uma física implementada em Lua e uma comparação dos tempos de execução obtidos são apresentados na seção 6.3.

4.5 Resolvedores numéricos

Conforme discutido anteriormente, o *framework* GeMA suporta diversos métodos de simulação. A maior parte dos métodos numéricos, porém, tem em comum o fato de transformarem o problema a ser resolvido em um sistema (linear) de equações.

Resolvedores numéricos são as entidades responsáveis pela solução destes sistemas de equação, sendo implementados através de *plugins* cuja principal tarefa é resolver um sistema do tipo $Ax = b$, onde A é uma matriz de tamanho $n \times n$, com n igual ao número de graus de liberdade do sistema. Dentro da arquitetura proposta, resolvedores distintos podem utilizar métodos numéricos distintos para a solução deste sistema de equações.

Geralmente, as matrizes envolvidas são grandes, porém esparsas. Na solução de problemas com dezenas de milhares de graus de liberdade, é fundamental que o fato das matrizes serem esparsas seja aproveitado para que apenas a parte não nula da matriz seja armazenada em memória.

Existem diversos padrões para o armazenamento eficiente de matrizes esparsas, e as bibliotecas para solução de sistemas de equações normalmente adotam, cada uma, um formato distinto. Para que o *framework* GeMA possa interfacear com o maior número possível de bibliotecas externas, sem estar atrelado a um formato próprio, ainda que padronizado, de armazenamento de matrizes, estas são tratadas como uma interface abstrata a ser implementada pelo *plugin* do resolvedor numérico. Assim, cada resolvedor pode adotar o formato mais adequado ao método (ou biblioteca) de solução adotado.

A interface adotada para matrizes, esparsas ou não, é bastante simples, consistindo apenas de funções para consultar e alterar valores em uma posição da

matriz e para multiplicar uma matriz por um vetor. Seus principais métodos são apresentados na tabela A.11. A criação de uma nova matriz é função do resolvedor numérico.

O resolvedor numérico incluído na implementação padrão do *framework* utiliza a biblioteca "Armadillo" (Sanderson, 2010), tanto para resolver sistemas lineares de equação quanto para suporte a matrizes esparsas[8]. Um parâmetro passado ao resolvedor permite a escolha de se trabalhar (ou não) com matrizes esparsas.

A biblioteca "Armadillo", por sua vez, utiliza a biblioteca "SuperLU" (Demmel *et al*, 1999; Li *et al*, 1999) para resolver sistemas baseados em matrizes esparsas, e a biblioteca "lapack" (Anderson *et al*, 1999) para matrizes completas.

4.6 Resultados

Após a execução de uma simulação, os dados gerados precisam ser salvos para visualização em programas de pós-processamento. Em simulações transientes, dados de interesse são gerados continuamente ao longo da execução da simulação.

Resultados podem ser salvos de maneira explícita através de processos chamados durante o *script* de orquestração. Esta maneira de trabalho é extremamente flexível, porém tem como desvantagem a necessidade de edição do *script* se for necessária uma alteração dos dados salvos. Em cenários de produção, onde os usuários finais do processo de simulação apenas configuram os dados de entrada de uma simulação preparada por especialistas, não é aconselhável permitir que o *script* de orquestração seja alterado.

A solução proposta consiste em permitir que o usuário descreva quais, como e quando os dados serão salvos, de maneira similar a como o modelo é descrito (seção 4.7). Esta é a primeira responsabilidade do monitor de resultados.

A segunda solução consiste em permitir ao usuário acompanhar resultados parciais da simulação para que este possa interromper sua execução quando os resultados indicarem que a simulação não está ocorrendo de acordo com o esperado.

4.6.1 Descrevendo os dados a serem salvos

A descrição dos dados a serem salvos baseia-se na criação de regras de salvamento de dados. Múltiplas regras podem ser definidas e o monitor de resultados gerencia sua aplicação. Cada regra é composta por três pontos principais:

[8] A biblioteca "Armadillo" também é utilizada para suporte a operações de álgebra linear com vetores e matrizes auxiliares.

1. **Quais** dados serão salvos. As grandezas a serem salvas são definidas através de uma lista contendo as variáveis de estado e/ou atributos (associados a nós, células ou pontos de integração) a serem armazenados. As unidades em que os dados serão salvos podem ser alteradas, se for o caso.

   Nada mais sendo especificado, serão incluídos todos os nós e/ou elementos da malha. Opcionalmente, este conjunto pode ser filtrado por um grupo de elementos ou mesmo por um conjunto de posições espaciais. O segundo caso permite o acompanhamento no tempo de um ponto específico.

2. **Como** e **onde** os dados serão salvos, através da especificação do nome e do formato dos arquivos gerados. Conforme ilustrado pela figura 4.4, os formatos possíveis para salvamento de dados são definidos pelo conjunto de *plugins* disponíveis.

   Cabe salientar que alguns tipos de arquivos não suportam todas as opções de salvamento de dados disponíveis. Cada *plugin* define suas capacidades, e o monitor de resultados verifica se o mesmo pode salvar o conjunto de dados selecionado. Atualmente, estão disponíveis os seguintes formatos:

   – Arquivo neutro: Formato padronizado adotado pelo Instituto Tecgraf e utilizado por diversos aplicativos, tais como o pós-processador "Pos-3D". Sua principal deficiência consiste em não permitir a variação da malha no tempo;
   – Arquivo mesh: Formato usado pelo software "Medit", desenvolvido pelo instituto INRIA (Frey, 2001). Suporta a evolução de dados no tempo através do salvamento de vários arquivos, integrados pelo visualizador. Sua principal deficiência está na falta de suporte a dados salvos em pontos de integração e falta de suporte a elementos não lineares;
   – Arquivos texto: Formato interno para salvamento de dados através de tabelas de valores;

3. **Quando** os dados serão salvos. Para simulações transientes, os momentos no tempo em que os dados devem ser salvos são especificados por um intervalo de tempo (que pode ser diferente do passo de simulação) ou por uma lista de instantes.

   O mesmo ocorre para simulações em regime permanente que envolvem uma evolução através de diversos passos. Neste caso, os momentos de salvamento são indicados através do passo de iteração, ao invés do tempo de simulação.

O *script* 4 ilustra a sintaxe utilizada na construção de regras de salvamento.

Como todo o gerenciamento da simulação é efetuado pelo *script* de orquestração, o monitor de resultados precisa ser notificado do andamento desta

***Script* 4** Regra para salvamento de dados.

```
1  local secsPerMyr = 60*60*24*365.25*1e6
2
3  -- Regras de salvamento de dados
4  SaveRule {
5    id          = 'rule1',
6    description = 'Salva dados de temperatura (em C) e fluxo '..
7                  'térmico a cada milhão de anos.',
8
9    -- Dados a serem salvos
10   data = {'T[degC]', 'q'},
11
12   -- Quando os dados serão salvos
13   saveStep = 1 * secsPerMyr,
14
15   -- Arquivo a ser salvo (extensão define o driver a ser usado) e
16   -- parâmetros adicionais a serem passados ao driver
17   file = 'resultFile.nf',
18   formatParameters = { },
19 }
```

para que o salvamento de dados seja efetuado no momento correto. Esta cooperação entre o *script* e o monitor é feita por chamadas de funções que indicam o início e o término de cada passo de tempo.

4.6.2 Monitorando resultados

O monitoramento de resultados baseia-se no uso do padrão "Observador" (Gamma *et al*, 1994). Em momentos importantes da simulação, tais como no início de cada passo de tempo, esta emite eventos que podem ser monitorados pela aplicação. Como o *framework* GeMA utiliza a biblioteca Qt[9] como base, naturalmente observadores são implementados através de sinais e *slots* (Blanchette e Summerfield, 2008).

Para complementar as notificações ao início e término de cada passo no tempo, tanto o *script* de orquestração quanto processos e físicas podem notificar o monitor de resultados do progresso atual, de forma que este possa emitir eventos para a aplicação.

Para que a aplicação receba informações sobre o andamento da simulação, notificações de progresso podem incluir "indicadores". Indicadores são valores calculados pela simulação, associados a um nome padronizado, que quantificam o progresso da simulação. Exemplos de indicadores incluem:

– O progresso percentual da simulação no tempo;

[9] A biblioteca Qt é bastante utilizada para a criação de interfaces com o usuário em aplicações científicas. No âmbito do *framework*, porém, apenas o seu núcleo central é utilizado como uma biblioteca de classes multiplataforma.

– Indicações de convergência da simulação;

– Estimativas de erro;

– Valores calculados pela simulação, tais como a temperatura ou a tensão em um ponto do domínio, a posição da interface sólido/líquido em um problema de moldagem, etc;

Quando o monitor emite um evento de progresso, disponibiliza para a aplicação os valores atuais dos indicadores existentes. A aplicação, por sua vez, pode, então, apresentar estes valores para acompanhamento pelo usuário. Se este desejar cancelar a simulação, poderá solicitar o cancelamento, que será efetuado de maneira síncrona e segura quando o monitor for processar a próxima notificação de progresso.

4.7 Definição de modelos

Esta seção tem como objetivo complementar a descrição do *framework* feita nas seções anteriores, "materializando" os principais conceitos apresentados através de um exemplo de simulação, simples, porém completo. Esta simulação baseia-se na estrutura de uma ponte composta por treliças, apresentado em Felippa (2004) e exemplificado na figura 4.22. É importante notar que este exemplo não engloba todas as entidades possíveis a uma simulação GeMA. Seu objetivo é apresentar o estilo de descrição adotado, e não simplesmente servir como um manual de opções disponíveis. Outras construções serão apresentadas, quando necessário, no decorrer do capítulo 6.

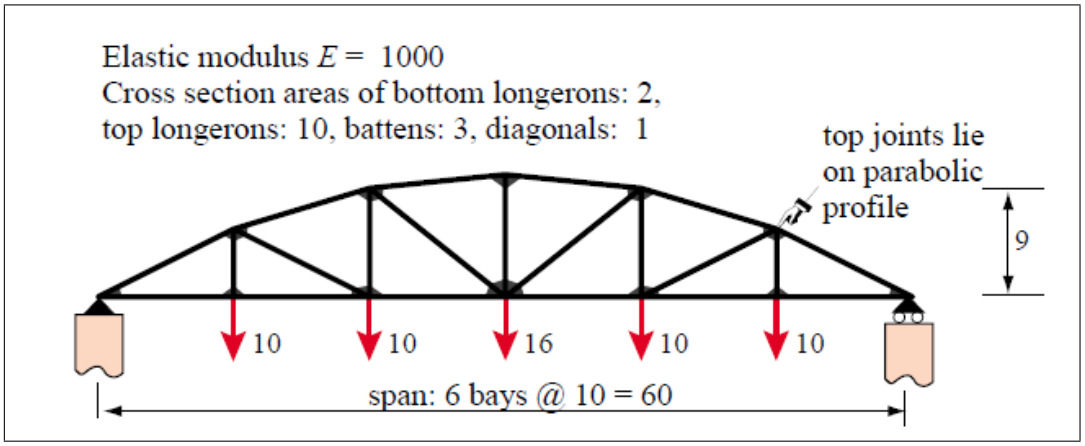


Figura 4.22: Exemplo de estrutura de uma ponte composta por treliças. Fonte: Felippa (2004), seção 21.6.

Uma simulação GeMA é descrita por um arquivo textual contendo a descrição dos dados do modelo, do método de solução e dos dados a serem monitorados. Nada impede que estes elementos sejam criados através da construção manual de objetos

em C++ com uso da *API* do *framework*, mas geralmente sua descrição textual é preferível, mesmo que seu uso esteja embutido em uma interface gráfica.

Como a descrição é feita através de um arquivo Lua, construções da linguagem podem ser utilizadas para separar as entidades em arquivos distintos. Desta forma, normalmente o arquivo principal é responsável por carregar um arquivo com a descrição dos dados do modelo, outro arquivo com o método de solução e, opcionalmente, um terceiro arquivo com as regras de salvamento de dados, se este não for feito de maneira explícita durante a orquestração. O *script* 5 apresenta o arquivo principal para a simulação de exemplo.

***Script* 5** Arquivo principal de simulação - 'bridge.lua'.

```
1 -- Descrição da simulação
2 Simulation {
3   name        = 'Bridge',
4   description = 'Bridge model based on Felippa, example found '..
5                 'in section 21.6, figures 21.13 through 21.16'
6 }
7
8 dofile('$SIMULATIONDIR/$SIMULATIONNAME_model.lua')
9 dofile('$SIMULATIONDIR/$SIMULATIONNAME_solution.lua')
```

Em sua forma mais simples, modelos são especificados utilizando-se a linguagem Lua de uma forma puramente descritiva. Nestas situações, a linguagem é utilizada como um *parser* automático que valida a sintaxe utilizada e transforma as entidades descritas, de textos entrados pelo usuário, em um formato estruturado contendo objetos em Lua que serão posteriormente lidos pela aplicação e transformados em instâncias das classes utilizadas pelo *framework*. No *script* 5, a entidade `Simulation` é responsável por descrever meta-dados globais sobre a simulação.

Usuários mais avançados, porém, podem se beneficiar do fato de que os arquivos de descrição são, na realidade, códigos Lua, executados quando o arquivo é lido, para manipular e criar o modelo de forma dinâmica[10]. Uma malha regular, por exemplo, pode ser criada automaticamente por uma chamada a uma função que recebe como parâmetros as coordenadas de seus limites e o número de subdivisões desejadas.

Os *scripts* 6, 7 e 8 apresentam a listagem do modelo de simulação. Os deslocamentos calculados são representados pelas variáveis de estado `ux` e `uy`. O módulo de elasticidade e a área da seção transversal de cada treliça são representados pelas propriedades `E` e `A`, que são associadas aos elementos pela tabela `mesh_elements`.

[10] `Simulation` é, na verdade, uma função em Lua que recebe como parâmetro uma única tabela, cujo conteúdo será validado pela função e armazenado em uma estrutura global para ser lida pelo *framework*.

***Script* 6** Descrição do modelo - 'bridge_model.lua' (1/3)

```
1  -- Variáveis de estado
2  StateVar {id = 'ux', description = 'Displacement in the X direction',
3            format = '8.4f'}
4  StateVar {id = 'uy', description = 'Displacement in the Y direction',
5            format = '8.4f'}
6
7  --  Conjuntos de propriedades associadas aos elementos do modelo
8  PropertySet {
9    id          = 'MatProp',
10   typeName    = 'GemaStdPropertySet',
11   description = 'Material parameters',
12   properties  = {
13     {id = 'E', description = 'Elasticity modulus'},
14     {id = 'A', description = 'Cross section area'},
15   },
16   values = {
17     {id = 'Bottom',   E = 1000, A =  2.0},
18     {id = 'Top',      E = 1000, A = 10.0},
19     {id = 'Batten',   E = 1000, A =  3.0},
20     {id = 'Diagonal', E = 1000, A =  1.0},
21   }
22 }
23
24 -- Lista de coordenadas dos nós da malha (X, Y)
25 local mesh_nodes = {
26   { 0.0,   0.0}, {10.0,   5.0}, {10.0,   0.0}, {20.0,   8.0},
27   {20.0,   0.0}, {30.0,   9.0}, {30.0,   0.0}, {40.0,   8.0},
28   {40.0,   0.0}, {50.0,   5.0}, {50.0,   0.0}, {60.0,   0.0},
29 }
30
31 -- Lista de nós e propriedades por elemento da malha
32 local mesh_elements = {
33   { 1,   3, MatProp = 'Bottom'  },
34   { 3,   5, MatProp = 'Bottom'  },
35   { 5,   7, MatProp = 'Bottom'  },
36   { 7,   9, MatProp = 'Bottom'  },
37   { 9,  11, MatProp = 'Bottom'  },
38   {11,  12, MatProp = 'Bottom'  },
39   { 1,   2, MatProp = 'Top'     },
40   { 2,   4, MatProp = 'Top'     },
41   { 4,   6, MatProp = 'Top'     },
42   { 6,   8, MatProp = 'Top'     },
43   { 8,  10, MatProp = 'Top'     },
44   {10,  12, MatProp = 'Top'     },
45   { 2,   3, MatProp = 'Batten'  },
46   { 4,   5, MatProp = 'Batten'  },
47   { 6,   7, MatProp = 'Batten'  },
48   { 8,   9, MatProp = 'Batten'  },
49   {10,  11, MatProp = 'Batten'  },
50   { 2,   5, MatProp = 'Diagonal'},
51   { 4,   7, MatProp = 'Diagonal'},
52   { 7,   8, MatProp = 'Diagonal'},
53   { 9,  10, MatProp = 'Diagonal'},
54 }
```

***Script* 7** Descrição do modelo - 'bridge_model.lua' (2/3)

```
1  -- Definição da malha
2  Mesh {
3    -- Atributos gerais
4    id          = 'bridgeMesh',
5    typeName    = 'GemaStdMesh.elem',
6    description = 'Bridge mesh discretization',
7
8    -- Numero de dimensoes do problema
9    coordinateDim = 2,
10
11   -- Listas de variáveis de estado armazenadas por nó
12   -- e propriedades armazenadas por elemento da malha
13   stateVars      = {'ux', 'uy'},
14   cellProperties = {'MatProp'},
15
16   -- Geometria
17   nodeData = mesh_nodes,
18   cellData = {{cellType = 'bar2', cellList = mesh_elements}},
19 }
20
21 --  Condições de contorno
22 BoundaryCondition {
23   id          = 'bc1',
24   description = 'External forces applied to model nodes',
25   type        = 'node forces',
26   mesh        = 'bridgeMesh',
27
28   properties  = {
29     {id = 'f',  description = 'External force applied on the node',
30      dim = 2},
31   },
32
33   nodeValues = {
34     -- node,  fx,     fy
35     { 3,     {0.0, -10.0}},
36     { 5,     {0.0, -10.0}},
37     { 7,     {0.0, -16.0}},
38     { 9,     {0.0, -10.0}},
39     {11,     {0.0, -10.0}},
40   }
41 }
```

Durante a definição da malha, denominada de `bridgeMesh`, sua geometria é dada pelos campos `nodeData` e `cellData`. É interessante notar que este último é uma tabela contendo subtabelas para permitir o tratamento de malhas heterogêneas e grupos de elementos.

As condições de contorno do problema são fornecidas pelos objetos `bc1` e `bc2`. Sua descrição segue um padrão absolutamente genérico, usado para impor forças externas e remover graus de liberdade dos nós da malha, mas que também pode ser usado, por exemplo, para definir as temperaturas e o fluxo térmico nas bordas de um conjunto de células, em outras simulações.

***Script* 8** Descrição do modelo - 'bridge_model.lua' (3/3)

```
1  -- Condições de contorno (continuação)
2  BoundaryCondition {
3    id          = 'bc2',
4    description = 'Prescribed node displacements',
5    type        = 'node displacements',
6    mesh        = 'bridgeMesh',
7
8    properties  = {
9      {id = 'ux',  description = 'Fixed node displacement in the X direction',
10      defVal = -9999},
11     {id = 'uy',  description = 'Fixed node displacement in the Y direction',
12      defVal = -9999},
13   },
14
15   nodeValues = {
16     -- node,  ux,  uy
17     { 1,     0.0, 0.0},  -- Node fixed in both directions
18     {12,     nil, 0.0},  -- Node fixed in y but free to move in x
19   }
20 }
```

Os valores associados ao ponto de aplicação da condição são definidos pelo conjunto de propriedades descritas pelo atributo `properties`, que segue o mesmo padrão geral usado para descrever variáveis de estado, atributos e propriedades da malha, conforme detalhado na seção 4.2.2.

Naturalmente, para que uma física utilize uma condição de contorno, é necessário que os valores fornecidos estejam de acordo com o esperado pela física, ou seja, esta precisa entender a semântica da condição de contorno. Esta ligação é feita através do atributo `type`, cujos valores possíveis são padronizados e indicam a estrutura esperada.

Durante a leitura de um modelo, nem todas as informações descritas são interpretadas pelo *framework* GeMA. Variáveis de estado e condições de contorno são interpretadas completamente pelo *framework*. Por outro lado, malhas e conjuntos de propriedades são entidades implementadas por *plugins*, e, portanto, o *framework* interpreta apenas as informações básicas necessárias para identificação da entidade, do *plugin* a ser carregado e do tipo de objeto a ser instanciado. Demais informações são interpretadas e validadas pelo próprio *plugin*. Na malha apresentada no *script* 7, e em qualquer outra entidade implementada através de *plugins*, os atributos `id`, `description` e `typeName` são lidos pelo *framework*. Todos os demais são interpretados pelo *plugin*. No exemplo, o valor "GemaStdMesh.elem", atribuído ao atributo `typeName`, indica que a malha deve ser instanciada pelo *plugin* "GemaStdMesh" e tem tipo "elem". O significado do tipo é arbitrado pelo *plugin*, e neste caso, indica que o mesmo deve criar uma malha de elementos.

Desta forma, um hipotético *plugin* de malhas que lê sua geometria de um arquivo em formato CAD, por exemplo, pode conter em sua descrição apenas o caminho do arquivo a ser lido, não sendo necessária a existência de tabelas com listas de nós e elementos, utilizadas pela implementação padrão.

O *script* 9 completa a simulação, apresentando a descrição do método de solução, já que este exemplo não faz uso de regras de salvamento. Os objetos `stressPhysics` e `solver` são utilizados para definir que a simulação irá utilizar a física padrão de cálculo de tensões para barras e o resolvedor numérico padrão. A física está associada com a malha de simulação e com as condições de contorno apresentadas previamente. O *script* de orquestração segue o mesmo padrão apresentado na seção 4.3, efetuando chamada ao processo de solução via elementos finitos e depois utilizando processos utilitários para imprimir os resultados da simulação (deslocamentos calculados nos nós da malha e tensões calculadas sobre seus elementos).

***Script* 9** Descrição do modelo - 'bridge_solution.lua'

```
1  -- Física utilizada
2  PhysicalMethod {
3    id       = 'stressPhysics',
4    typeName = 'GemaStdStressPhysics.bar',
5    type     = 'fem',
6
7    mesh               = 'bridgeMesh',
8    boundaryConditions = {'bc1', 'bc2'},
9  }
10
11 -- Resolvedor numérico utilizado
12 NumericalSolver {
13   id          = 'solver',
14   typeName    = 'GemaStdNumSolver',
15   description = 'Standard matrix solver',
16 }
17
18 -- Script de orquestração
19 function ProcessScript()
20   fem.solve({'stressPhysics'}, 'solver')
21
22   utils.print('Calculated results:')
23   utils.printMeshNodeData('bridgeMesh', {'ux', 'uy'})
24   utils.print('')
25   utils.printMeshCellData('bridgeMesh', 's')
26 end
```

Com exceção da função de orquestração, todas as demais entidades pertencentes à simulação são representadas através de tabelas em Lua, contendo, ao menos em seu primeiro nível, pares do tipo (atributo, valor). Em particular, todas as entidades incluem um nome e sua descrição, bem como a indicação clara do *plugin* associado, se for o caso. Estas informações, combinadas com as

características de reflexão da linguagem Lua, permitem que seja possível recuperar diversos metadados sobre uma simulação de forma automática para, por exemplo, alimentar bases de conhecimento que listem os tipos de simulação disponíveis em uma organização. Podem ser utilizadas também para permitir a construção de interfaces gráficas genéricas para edição dos modelos.

5
Modelos de teste

Este capítulo apresenta os modelos matemáticos utilizados nos testes descritos na seção 1.3 e cujos resultados são apresentados no capítulo 6.

As discretizações das equações para cálculos de temperatura e tensão através do método de elementos finitos são bastante conhecidas na literatura. Desta forma, o modelo discreto utilizado é apresentado sem grandes discussões nas seções 5.1 e 5.2. A modelagem de bacias, entretanto, é um assunto menos difundido e, portanto, a seção 5.3 faz uma introdução sobre o tema, além de apresentar os modelos utilizados.

5.1
Temperatura

Em regime permanente, a condução de calor é dada pela equação 5-1, onde T é a temperatura (em K), λ é o tensor de condutividade térmica (em $W/m.K$) e G é a taxa de geração de calor por unidade de volume (em W/m^3).

$$\nabla \cdot (\lambda \nabla T) + G = 0 \tag{5-1}$$

As condições de contorno associadas às bordas Γ_d, Γ_{n1} e Γ_{n2} do domínio Ω (figura 5.1) podem ser de Dirichlet, onde o valor da temperatura na borda é especificado (equação 5-2), ou de Neumann, onde o fluxo térmico na direção normal à superfície é especificado (equação 5-3). Condições adiabáticas podem ser obtidas pela condição de Neumann com $q_0 = 0$ e condições de transferência de calor convectivo podem ser obtidas com $q_0 = h(T - T_a)$, onde h é o coeficiente de convecção (em $W/m^2.K$) e T_a a temperatura ambiente (equação 5-4).

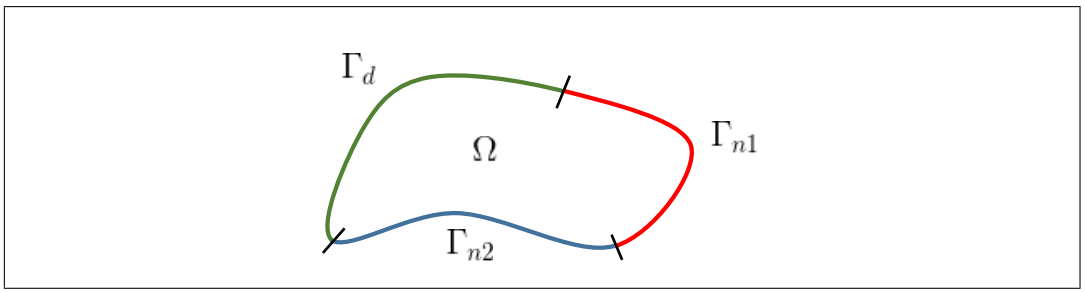


Figura 5.1: Condições de contorno para equação de condução de calor.

$$T = T_0 \quad \text{em} \quad \Gamma_d \tag{5-2}$$

$$-\lambda \frac{\partial T}{\partial n} = q_0 \quad \text{em} \quad \Gamma_{n1} \tag{5-3}$$

$$-\lambda \frac{\partial T}{\partial n} = h(T - T_a) \quad \text{em} \quad \Gamma_{n2} \tag{5-4}$$

Estas equações, quando discretizadas através do método de Galerkin, reduzem-se às equações 5-5 à 5-8, apresentadas por Lewis *et al* (2004), onde n é o número de nós do elemento, K_e e f_e são, respectivamente, a matriz de rigidez e o vetor de forças locais, T_e é o vetor de temperaturas associadas aos nós do elemento e $N = \{N_1, N_2, \cdots, N_n\}$ é o vetor de funções de forma.

$$K_e T_e = f_e \tag{5-5}$$

$$K_e = \int_\Omega B^T \lambda B \, d\Omega + \int_{\Gamma_{n2}} h N^T N \, d\Gamma \tag{5-6}$$

$$f_e = \int_\Omega G N^T \, d\Omega - \int_{\Gamma_{n1}} q_0 N^T \, d\Gamma + \int_{\Gamma_{n2}} h T_a N^T \, d\Gamma \tag{5-7}$$

$$B = \begin{bmatrix} \frac{\partial N_1}{\partial x} & \frac{\partial N_2}{\partial x} & \cdots & \frac{\partial N_n}{\partial x} \\ \frac{\partial N_1}{\partial y} & \frac{\partial N_2}{\partial y} & \cdots & \frac{\partial N_n}{\partial y} \\ \frac{\partial N_1}{\partial z} & \frac{\partial N_2}{\partial z} & \cdots & \frac{\partial N_n}{\partial z} \end{bmatrix} \tag{5-8}$$

Em regime transiente, a equação 5-1 é substituída pela equação 5-9, onde ρ é a densidade do material (em kg/m^3) e c_p é seu calor específico à pressão constante (em $J/kg.K$).

$$\rho c_p \frac{\partial T}{\partial t} = \nabla \cdot (\lambda \nabla T) + G \tag{5-9}$$

Após a discretização espacial da equação 5-9 são obtidas as novas equações 5-10 e 5-11, com K_e e f_e dados, respectivamente, pelas equações 5-6 e 5-7.

$$C_e \frac{\partial T_e}{\partial t} + K_e T_e = f_e \tag{5-10}$$

$$C_e = \int_\Omega \rho c_p N^T N \, d\Omega \tag{5-11}$$

Efetuando a discretização da equação 5-10 no tempo através de diferenças finitas, obtém-se a equação 5-12, onde θ é um parâmetro que permite alterar o esquema de discretização e Δt é o intervalo de tempo entre os instantes t^n e t^{n+1}. Para $\theta = 0$ obtém-se uma discretização explícita, para $\theta = 1$ uma discretização totalmente implícita e para $\theta = 0.5$ o método de Crank-Nicolson (Lewis *et al*, 2004). Nas simulações de teste apresentadas no capítulo 6 foi utilizado o esquema com $\theta = 1$ (equação 5-13).

$$(C + \theta\Delta tK)T^{n+1} = (C - (1-\theta)\Delta tK)T^n + \Delta t(\theta f^{n+1} + (1-\theta)f^n) \quad (5\text{-}12)$$

$$(C + \Delta tK)T^{n+1} = CT^n + \Delta t f^{n+1} \text{ com } \theta = 1 \quad (5\text{-}13)$$

Problemas envolvendo mudança de fase requerem que a energia absorvida ou liberada durante o processo seja considerada. A transição entre os estados líquido e sólido podem ser tratados através do conceito de calor específico efetivo (c_{eff}) dado pela equação 5-14, onde c_s é o calor específico do material em estado sólido, c_l no estado líquido e c_f durante a transição, L é o calor latente de fusão (em J/kg), T_s é a temperatura em que o material é sólido e T_l é a temperatura em que o material é líquido (Lewis *et al*, 2004; Wangen, 2010).

$$c_{eff} = \begin{cases} c_s & \text{se } T \leqslant T_s \\ c_f + \dfrac{L}{T_l - T_s} & \text{se } T_s < T < T_l \\ c_l & \text{se } T \geqslant T_l \end{cases} \quad (5\text{-}14)$$

O problema de condução de calor pode se tornar um problema não linear se a condutividade térmica e o calor específico forem considerados funções da temperatura: $\lambda(T)$ e $c_p(T)$. O problema resultante pode ser resolvido através do método de Newton-Raphson ou através de iterações até que a convergência seja alcançada.

5.2 Tensões

5.2.1 Treliças

Para problemas lineares elásticos, o tratamento de estruturas formadas apenas por treliças pode ser feito através do método de rigidez direta ("*direct stiffness*"). Como nestes casos os deslocamentos das treliças são completamente definidos pelos deslocamentos das articulações, o sistema a ser resolvido é da forma $Ku = f$, onde u é o vetor formado pelos deslocamentos em x, y e z das articulações, f é o vetor

de forças externas aplicadas sobre a estrutura e K é uma matriz que estabelece a relação linear entre forças e deslocamentos.

Para uma treliça com coordenadas em 2D, sua matriz de rigidez local K_e é dada pela equação 5-15, onde E é o módulo de elasticidade do material (em N/m^2), A é a área da seção transversal da treliça (em m^2), L é o comprimento da treliça (em m) e ϕ é o ângulo entre a treliça e o eixo x. Maiores detalhes podem ser obtidos em Felippa (2004).

$$K_e = \frac{EA}{L}\begin{bmatrix} \cos^2\phi & \sin\phi\cos\phi & -\cos^2\phi & -\sin\phi\cos\phi \\ \sin\phi\cos\phi & \sin^2\phi & -\sin\phi\cos\phi & -\sin^2\phi \\ -\cos^2\phi & -\sin\phi\cos\phi & \cos^2\phi & \sin\phi\cos\phi \\ -\sin\phi\cos\phi & -\sin^2\phi & \sin\phi\cos\phi & \sin^2\phi \end{bmatrix} \quad (5\text{-}15)$$

5.2.2
Placas em estado plano de tensões

Para placas em estado plano de tensões, as equações 5-16 à 5-18 relacionam deslocamentos, tensões e deformações para um comportamento linear elástico, onde u é o vetor de deslocamentos (em m), σ é o tensor de tensões (em N/m^2), e é o tensor de deformações (adimensional), E é a matriz de tensão-deformação (em N/m^2) e b é o vetor de forças internas atuando na placa por unidade de volume (em N/m^3).

$$\begin{bmatrix} e_{xx} \\ e_{yy} \\ 2e_{xy} \end{bmatrix} = \begin{bmatrix} \partial/\partial x & 0 \\ 0 & \partial/\partial y \\ \partial/\partial x & \partial/\partial y \end{bmatrix} \begin{bmatrix} u_x \\ u_y \end{bmatrix} \quad (5\text{-}16)$$

$$\begin{bmatrix} \sigma_{xx} \\ \sigma_{yy} \\ \sigma_{xy} \end{bmatrix} = E \begin{bmatrix} e_{xx} \\ e_{yy} \\ e_{xy} \end{bmatrix} \quad (5\text{-}17)$$

$$\begin{bmatrix} \partial/\partial x & 0 & \partial/\partial y \\ 0 & \partial/\partial y & \partial/\partial x \end{bmatrix} \begin{bmatrix} \sigma_{xx} \\ \sigma_{yy} \\ \sigma_{xy} \end{bmatrix} + \begin{bmatrix} b_x \\ b_y \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \quad (5\text{-}18)$$

As condições de contorno associadas às bordas Γ_u e Γ_t do domínio Ω (figura 5.2) representam, respectivamente, deslocamentos prescritos, $u = c$, e trações (t) aplicadas à borda por unidade de área.

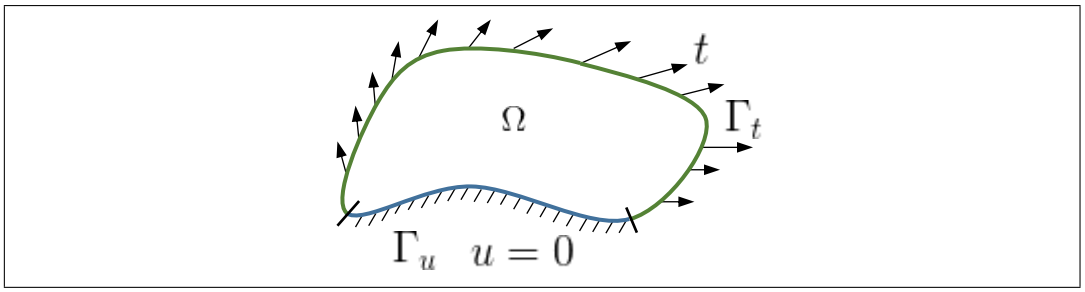


Figura 5.2: Condições de contorno para placas em estado plano de tensões.

Estas equações, quando discretizadas, reduzem-se às equações 5-19 à 5-23, apresentadas por Felippa (2004), onde n é o número de nós do elemento, K_e e f_e são, respectivamente, a matriz de rigidez e o vetor de forças locais, u_e é o vetor de deslocamentos associados aos nós do elemento, h é a espessura da placa e $\{N_1, \cdots, N_n\}$ são as funções de forma do elemento.

$$K_e u_e = f_e \tag{5-19}$$

$$K_e = \int_\Omega h B^T E B \, d\Omega \tag{5-20}$$

$$f_e = \int_\Omega h \bar{N}^T b \, d\Omega + \int_{\Gamma_t} h \bar{N}^T t \, d\Gamma \tag{5-21}$$

$$\bar{N} = \begin{bmatrix} N_1 & 0 & N_2 & 0 & \cdots & N_n & 0 \\ 0 & N_1 & 0 & N_2 & \cdots & 0 & N_n \end{bmatrix} \tag{5-22}$$

$$B = \begin{bmatrix} \frac{\partial N_1}{\partial x} & 0 & \frac{\partial N_2}{\partial x} & 0 & \cdots & \frac{\partial N_n}{\partial x} & 0 \\ 0 & \frac{\partial N_1}{\partial y} & 0 & \frac{\partial N_2}{\partial y} & \cdots & 0 & \frac{\partial N_n}{\partial y} \\ \frac{\partial N_1}{\partial y} & \frac{\partial N_1}{\partial x} & \frac{\partial N_2}{\partial y} & \frac{\partial N_2}{\partial x} & \cdots & \frac{\partial N_n}{\partial y} & \frac{\partial N_n}{\partial x} \end{bmatrix} \tag{5-23}$$

Maiores detalhes para cálculos lineares de tensão podem ser obtidos em Felippa (2004); Zienkiewicz *et al* (2005). O tratamento de modelos não-lineares é dado por Crisfield (1991).

5.2.3 Acoplamento tensão - temperatura

O modelo matemático apresentado na seção anterior considera que as deformações ocorrem devido à aplicação de forças externas, porém estas podem ocorrer também em resposta a variações de temperatura, que podem ser modeladas através de uma "força térmica". Desta forma, a equação 5-19 deve ser alterada para

incluir o efeito térmico, cuja descrição é dada por Logan (2011) e apresentada nas equações 5-24 à 5-26, onde o vetor de forças externas mecânicas f_{eM} é igual ao vetor f_e apresentado na equação 5-21, α é o coeficiente de expansão térmica do material (em $1/K$) e ΔT é a alteração de temperatura imposta ao elemento.

$$K_e u_e = f_{eM} + f_{eT} \tag{5-24}$$

$$f_{eT} = \int_\Omega h B^T E e_T \, d\Omega \tag{5-25}$$

$$e_T = \alpha \Delta T \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} \tag{5-26}$$

Para um efetivo acoplamento entre os cálculos de tensão e temperatura, é desejado que seja construído um único sistema de equações para o elemento envolvendo ambos os fenômenos. Este acoplamento pode ser construído através da substituição do valor de ΔT na equação 5-26 por $(T - T_r)$, onde T é a temperatura em um ponto do elemento e T_r é uma temperatura de referência.

Lembrando que, dentro do elemento, $T = NT_e$, e transpondo os termos dependentes de T_e em f_{eT} do vetor de forças para a matriz de acoplamento, é obtido o sistema acoplado dado pelas equações 5-27 à 5-29, onde K_e^σ e f_{eM}^σ são, respectivamente, a matriz de rigidez e o vetor de forças para o modelo independente de tensões (equações 5-20 e 5-21), e K_e^T e f_e^T são, respectivamente, a matriz de rigidez e o vetor de forças para o modelo de temperatura (equações 5-6 e 5-7).

$$\begin{bmatrix} K_e^\sigma & C \\ 0 & K_e^T \end{bmatrix} \begin{bmatrix} u_e \\ T_e \end{bmatrix} = \begin{bmatrix} f_{eM}^\sigma \\ f_e^T \end{bmatrix} + \begin{bmatrix} F \\ 0 \end{bmatrix} \tag{5-27}$$

$$C = -\int_\Omega h \alpha B^T E \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} N \, d\Omega \tag{5-28}$$

$$F = -\int_\Omega h \alpha T_r B^T E \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} \, d\Omega \tag{5-29}$$

É interessante notar que o termo inferior esquerdo da matriz resultante é zero, uma vez que o cálculo de temperatura não depende do cálculo dos deslocamentos.

5.3 Modelagem de bacias

A modelagem de bacias consiste em um conjunto de conhecimentos e técnicas que têm como objetivo estudar a formação e a evolução temporal de bacias sedimentares, de modo a caracterizar o sistema petrolífero e quantificar potenciais acumulações de hidrocarbonetos, tornando claros os riscos envolvidos nos processos de exploração e explotação.

Um sistema petrolífero é caracterizado por todos os elementos estruturais necessários para a existência de hidrocarbonetos, tais como rochas geradoras, reservatórios e selos, bem como o mecanismo de formação da trapa e a relação temporal de formação de todos estes elementos de modo a permitir a existência de acumulações de hidrocarbonetos.

Através de simulações físicas, a modelagem de bacias estuda os processos de deposição, compactação, história térmica, maturação, geração, expulsão, migração e trapeamento de hidrocarbonetos, o que permite a quantificação do volume de óleo e gás gerados e a determinação dos momentos críticos da geração e acumulação, provendo informações vitais para a determinação do sistema petrolífero.

Desta forma, a definição do sistema petrolífero e os resultados da simulação ajudam a diminuir os riscos inerentes ao processo exploratório de óleo e gás. Os principais fenômenos físicos estudados são:

1. **Deposição e compactação:** A formação de uma bacia sedimentar ocorre pela deposição ao longo do tempo de sedimentos que gradativamente preenchem a bacia. Na medida em que estes sedimentos são depositados, o próprio peso dos sedimentos compacta os sedimentos localizados na camada abaixo.

   Um dos passos iniciais na modelagem de bacias consiste em reconstruir a geometria das camadas sedimentares no passado, levando-se em conta que a espessura encontrada hoje é menor do que quando estas foram depositadas (descompactação), bem como as características individuais da composição litológica de cada camada. Neste processo, também podem ser levados em consideração outros tipos de eventos geológicos, tais como erosão, hiatos, intrusões, halocinese, etc.

2. **História térmica:** A história térmica consiste na definição do perfil de temperatura da bacia ao logo de toda a sua história, ou seja, desde o início de sua formação até o presente.

   Em geral, a temperatura de uma camada aumenta de acordo com sua profundidade (soterramento), logo é dependente da geometria das camadas reconstruídas durante o processo de descompactação. Além disso, a história

térmica é afetada tanto pelas propriedades térmicas de cada camada quanto por condições de contorno, tais como a temperatura da superfície e o fluxo térmico na base da bacia, ambas variando no tempo. Eventos intrusivos também são importantes pois geram alterações significativas no perfil de temperatura em intervalos pontuais da história térmica.

3. **Maturação e geração de hidrocarbonetos:** Hidrocarbonetos são produtos da diagênese da matéria orgânica (querogênio) contida nos sedimentos. Sua formação depende do tipo de matéria orgânica e da história térmica (tempo e temperatura aos quais os sedimentos foram submetidos).

   A maturação térmica de uma rocha é uma medida do grau de transformação a que esta foi submetida. Para validar a história térmica modelada para a bacia, utilizam-se indicadores de maturação, chamados também de indicadores de paleo temperatura. Estes indicadores são propriedades, tais como a reflectância da vitrinita (Ro), que podem ser observadas na bacia através de medidas obtidas em poços, mas que também podem ser previstas através de modelos cinéticos. A comparação entre os valores medidos e calculados é, então, uma excelente ferramenta para validar a história térmica do modelo.

   De posse da história térmica calibrada, pode-se estimar o volume de matéria orgânica contida na rocha geradora que foi convertida em hidrocarbonetos. Para isso, utilizam-se modelos que têm como entrada a história térmica à qual os sedimentos foram submetidos e um conjunto de parâmetros cinéticos que definem como a matéria orgânica responde às alterações de temperatura.

4. **Expulsão, migração e trapeamento de hidrocarbonetos:** A medida em que hidrocarbonetos são gerados, a rocha geradora tende a ficar saturada. Quando um determinado limite de saturação é ultrapassado, o óleo/gás gerado é expulso da rocha geradora, migrando através de caminhos permeáveis até encontrar um reservatório, capeado por um selo que impeça que a migração continue, gerando então uma acumulação de hidrocarbonetos.

O simulador 2D de bacias implementado neste trabalho inclui cálculos de compactação, história térmica (incluindo tratamento para intrusões ígneas), maturação e geração de hidrocarbonetos. Processos de expulsão, migração e trapeamento não são considerados. Maiores informações sobre modelagem de bacias em geral podem ser encontradas em Hantschel e Kauerauf (2009); Wangen (2010).

5.3.1 Compactação

A figura 5.3 apresenta um diagrama ilustrativo do processo de compactação de camadas. Na medida em que uma camada é adicionada ao topo da coluna sedimentar, seu peso compacta as camadas inferiores.

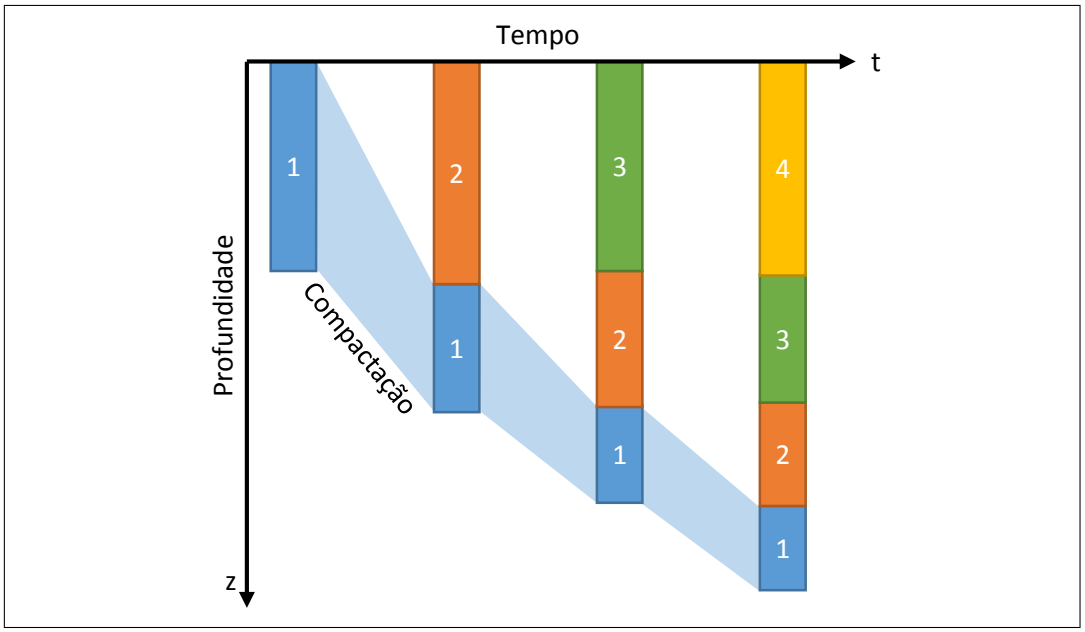


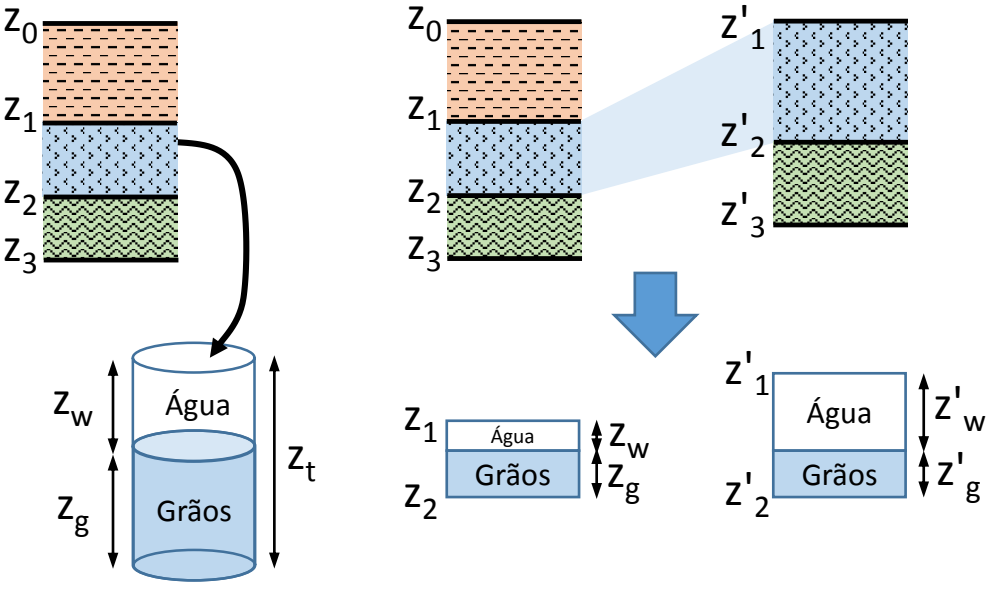
Figura 5.3: Compactação de camadas.

Nos cálculos de compactação assume-se que a mesma ocorre devido a uma diminuição da porosidade da camada, ocasionada unicamente pela expulsão de água e pela reorganização dos grãos, mantendo-se a massa de grãos original. Dado que a massa de grãos da camada não se altera, pode-se usar esta como base para o processo de compactação/descompactação, com variação apenas do volume de água da camada de acordo com sua profundidade na bacia.

A figura 5.4(a) mostra uma camada decomposta em duas partes distintas: uma parte composta apenas por grãos, de volume V_g, e outra composta apenas por água, de volume V_w. O volume total da camada é dado, então, por $V_t = V_g + V_w$, e sua porosidade pela relação entre o volume de água (que preenche o espaço vazio entre os grãos da rocha) e o volume total, ou seja, $\phi = \dfrac{V_w}{V_w + V_g}$.

Considerando-se, sem perda de generalidade, que a seção da camada apresentada na figura é unitária, pode-se trabalhar apenas com as espessuras de água e grãos, z_w e z_g. Assim, o volume de água pode ser calculado pela integral do espaço poroso, conforme a equação 5-30.

$$z_w = \int_{z_1}^{z_2} \phi(z)\, dz \tag{5-30}$$



(a) Decomposição (b) Descompactação

Figura 5.4: Descompactação de uma camada. (a) Decomposição da camada em um volume composto apenas por água e em um volume composto apenas por grãos; (b) Descompactação da camada z_1-z_2.

Como a espessura dos grãos é constante, pode-se calcular a espessura de uma camada, qualquer que seja sua profundidade inicial, compactando-a ou descompactando-a de acordo com a função $\phi(z)$. A figura 5.4(b) ilustra o processo. Quando uma camada com topo e base em z_1 e z_2 é movida para as profundidades z'_1 e z'_2, tem-se que $z_g = z_2 - z_1 - z_w = z'_g = z'_2 - z'_1 - z'_w$ e, portanto, pode-se calcular z'_2 através da equação 5-31.

$$z'_2 = z'_1 + z_2 - z_1 - \int_{z_1}^{z_2} \phi(z)\, dz + \int_{z'_1}^{z'_2} \phi(z)\, dz \tag{5-31}$$

Para que a equação 5-31 possa ser efetivamente usada, é necessário que seja estabelecida uma função para $\phi(z)$. Um dos modelos mais adotados é a função de Athy (1930) que estabelece uma relação exponencial entre a porosidade e a profundidade (equação 5-32), onde ϕ_0 é a porosidade inicial da rocha, c é uma constante de decaimento que depende do tipo de material e z é a profundidade de soterramento (não incluindo a lâmina d'água).

$$\phi(z) = \phi_0 e^{-cz} \tag{5-32}$$

Substituindo a função $\phi(z)$ em 5-31 pelo modelo de Athy e avaliando analiticamente as integrais, obtém-se a equação 5-33, proposta originalmente por Sclater e Christie (1980). Como o valor de z'_2 aparece em ambos os lados da

equação, esta deve ser avaliada numericamente. O uso de um algoritmo de ponto fixo é suficiente e converge em poucas iterações.

$$z_2' = z_1' + z_2 - z_1 - \frac{\phi_0}{c}(e^{-cz_1} - e^{-cz_2}) + \frac{\phi_0}{c}(e^{-cz_1'} - e^{-cz_2'}) \tag{5-33}$$

O simulador implementado neste trabalho utiliza a equação 5-33 para o cálculo de compactação e descompactação de camadas. Este método é um fenômeno unidimensional, simples de calcular, baseado exclusivamente na profundidade de soterramento das camadas.

Este modelo pode ser refinado através de uma abordagem baseada na tensão efetiva a que a rocha é submetida devido ao peso total dos sedimentos sobre esta. A base deste tipo de modelo consiste no cálculo da pressão a que a matriz da rocha e o fluido (água) estão submetidos. Em uma rocha totalmente compactada, a pressão nos poros é hidrostática, ou seja, equivalente à pressão da coluna de água sobre a camada. Este cenário pode ser alterado se a permeabilidade da camada ou das camadas adjacentes impedir que o fluido contido nos poros seja expelido pela compactação. Neste caso, a pressão nos poros aumenta, gerando uma sobrepressão que atua suportando parte do peso dos sedimentos sobre a camada. Isto ocasiona uma região com porosidade maior do que àquela que seria obtida se a compactação da rocha fosse completa para a profundidade. Este processo é modelado pelo conceito de tensão efetiva, definido como a diferença entre a pressão litostática (pressão total exercida pelos grãos e pela água sobre a camada) e a pressão de poro. O cálculo das pressões de poro pode ser feito usando-se a lei de Darcy para modelar a expulsão de água durante a compactação, fugindo ao escopo da presente simulação.

5.3.2 Temperatura

A história térmica da bacia pode ser simulada através do método de elementos finitos com utilização do modelo matemático apresentado na seção 5.1.

As condições de contorno geralmente utilizadas consistem da especificação da temperatura no topo da bacia e do fluxo térmico na base da mesma. Ambas variam no tempo geológico. Considera-se que não há fluxo térmico nas bordas laterais da bacia.

A solidificação de intrusões ígneas pode ser tratada através do método do calor específico efetivo apresentado anteriormente, mas é necessário que durante o período de solidificação o passo de tempo da simulação seja alterado da escala de milhões de anos para a escala de milhares de anos. Hantschel e Kauerauf (2009)

sugerem o uso de passos temporais de 500, 1.000, 2.000, 5.000, 10.000, 20.000, 50.000 e 100.000 anos após uma intrusão.

Durante o cálculo das matrizes locais dos elementos da malha, é fundamental levar em consideração o efeito da porosidade nos parâmetros térmicos utilizados, uma vez que um elemento é efetivamente uma mistura dos grãos da rocha com o fluido contido nos poros da mesma. No tratamento da condutividade térmica e do calor específico, é necessário levar em consideração ainda que estes parâmetros são funções da própria temperatura, conforme ilustrado pela figura 5.5.

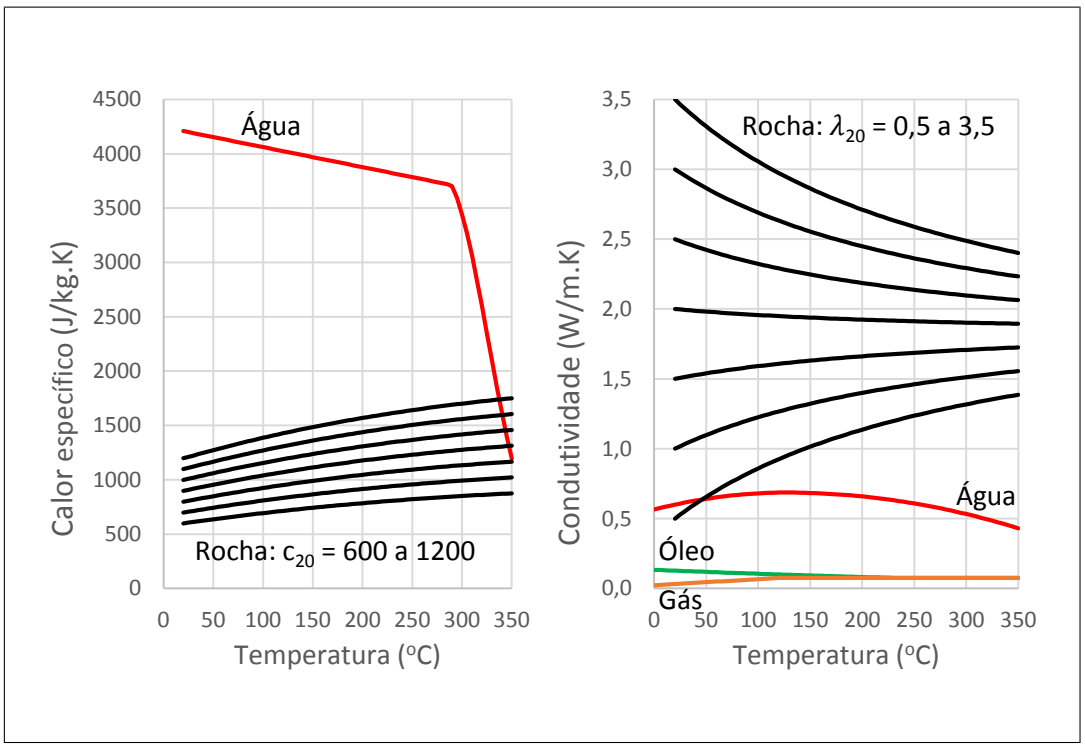


Figura 5.5: Variação do calor específico e da condutividade térmica com a temperatura. Adaptado de Hantschel e Kauerauf (2009).

Na determinação da densidade efetiva do elemento (ρ), utiliza-se uma média aritmética das densidades da rocha (ρ_r) e da água (ρ_w), ponderadas pela porosidade (equação 5-34).

$$\rho = \rho_w\phi + \rho_r(1-\phi) \tag{5-34}$$

Para o cálculo do calor radiogênico efetivo (Q), deve ser levado em consideração apenas o volume de rocha, uma vez que a produção de calor interna ao elemento deve-se ao decaimento radiativo de traços de urânio, tório e potássio presentes na rocha, e inexistente nos fluidos que preenchem os poros da mesma (equação 5-35).

$$Q = Q_r(1-\phi) \tag{5-35}$$

No cálculo do calor específico utiliza-se uma média aritmética das propriedades da rocha e da água, ponderadas pela porosidade, levando-se em consideração as dependências de ambos da temperatura (equação 5-38). Existem diversos modelos empíricos para estas dependências. Nas equações 5-36 e 5-37 são utilizados, respectivamente, os modelos propostos por Somerton (1992) e Waples e Waples (2004). Na equação 5-37, o termo c_{20} refere-se ao calor específico da rocha à 20°C.

$$c_{pw} = \begin{cases} 4245 - 1.841T & \text{se } T < 290°C \\ 3703e^{(\ -0.00481(T-290)\ -\ 0.000234(T-290)^2\ )} & \text{se } T \geqslant 290°C \end{cases} \quad (5\text{-}36)$$

$$c_{pr} = c_{20}(0.953 + 2.29 \times 10^{-3}T - 2.835 \times 10^{-6}T^2 + 1.191 \times 10^{-9}T^3) \quad (5\text{-}37)$$

$$c_p = c_{pw}\phi + c_{pr}(1 - \phi) \quad (5\text{-}38)$$

Finalmente, no cálculo da condutividade térmica utiliza-se uma média geométrica das propriedades da rocha e do fluido, ponderadas pela porosidade, levando-se em consideração as dependências de ambas da temperatura (equações 5-44 e 5-45). A inclusão da condutividade do óleo e do gás no cálculo geralmente ocorre apenas em simuladores que consideram fenômenos de migração, mas pode ser interessante, mesmo sem este cálculo, para cenários de gás de folhelho ("shale gas"), onde o reservatório é a própria rocha geradora.

$$T_K = T + 273.15 \quad (5\text{-}39)$$

$$\lambda_w = \begin{cases} 0.565 + 1.88 \times 10^{-3}T - 7.23 \times 10^{-6}T^2 & \text{se } T < 137°C \\ 0.602 + 1.31 \times 10^{-3}T - 5.14 \times 10^{-6}T^2 & \text{se } T \geqslant 137°C \end{cases} \quad (5\text{-}40)$$

$$\lambda_o = \begin{cases} 0.2389 - 4.593 \times 10^{-4}T_K + 2.676 \times 10^{-7}T_K^2 & \text{se } T < 240°C \\ 0.075 & \text{se } T \geqslant 240°C \end{cases} \quad (5\text{-}41)$$

$$\lambda_g = \begin{cases} -0.0969 + 4.37 \times 10^{-4}T_K & \text{se } T < 120°C \\ 0.075 & \text{se } T \geqslant 120°C \end{cases} \quad (5\text{-}42)$$

$$\lambda_r = 358(1.0227\lambda_{20} - 1.882)(\frac{1}{T_K} - 0.00068) + 1.84 \quad (5\text{-}43)$$

$$\lambda = (\lambda_w)^{\phi}(\lambda_r)^{(1-\phi)} \quad (5\text{-}44)$$

$$\lambda = (\lambda_w)^{(S_w\phi)}(\lambda_o)^{(S_o\phi)}(\lambda_g)^{(S_g\phi)}(\lambda_r)^{(1-\phi)} \quad (5\text{-}45)$$

Existem diversos modelos empíricos para as dependências térmicas da condutividade. Nas equações 5-40 e 5-43 são utilizados, respectivamente, os modelos propostos por Deming e Chapman (1989) e Sekiguchi (1984). Na equação 5-43, o termo λ_{20} refere-se à condutividade térmica da rocha à 20°C. As equações 5-41 e 5-42 são apresentandas por Hantschel e Kauerauf (2009).

5.3.3 Maturação e Geração

Modelos cinéticos tem como objetivo quantificar a taxa de conversão de um composto em seus derivados, podendo ser utilizados para quantificar como a matéria orgânica presente na rocha geradora reage às alterações de temperatura através do tempo, reduzindo a quantidade inicial de querogênio e formando hidrocarbonetos. Também podem ser utilizados para prever a reflectância da vitrinita (Ro), um importante indicador de maturação.

A vitrinita é um tipo de maceral, ou seja, um componente de origem orgânica, derivado de plantas, presente no carvão e na maioria dos folhelhos. Em laboratório, amostras retiradas dos poços podem ser analisadas para medição de Ro, uma medida da porcentagem de luz polarizada refletida pelas amostras de vitrinita.

O que torna o valor de Ro importante como indicador de maturação é o fato de que esta medida é sensível à história térmica da amostra, ou seja, quanto maior for a temperatura à qual a rocha foi submetida, maior será seu valor. As transformações ocorridas na vitrinita com o aumento de temperatura não são reversíveis, logo, se uma amostra for submetida a altas temperaturas, e depois esfria (devido a uma erosão, ou ao resfriamento de rochas intrusivas), sua reflectância não diminui. Desta forma, pode-se dizer que a vitrinita é um indicador sensível à máxima temperatura / maturação alcançada pela amostra (Beardsmore e Cull, 2001).

A premissa básica adotada nas análises cinéticas consiste em supor que a taxa de transformação do material original no material transformado depende apenas da temperatura e da quantidade de material disponível para ser convertido. Em cálculos de maturação e geração, utiliza-se uma relação linear dada pela equação 5-46, que integrada transforma-se na equação 5-47, onde w é a concentração do material a ser convertido, w_0 é a concentração inicial de material e $k(t)$ a taxa de conversão da reação. A equação 5-48 representa a fração de material convertida F.

$$\frac{dw}{dt} = -k(t)w \tag{5-46}$$

$$\frac{w}{w_0} = e^{-\int_0^t k(t)\,dt} \tag{5-47}$$

$$F = 1 - \frac{w}{w_0} \tag{5-48}$$

Geralmente, $k(t)$ é modelado como uma equação de Arrhenius (5-49), onde E é a energia de ativação (em J/mol), representando o limiar de energia necessário para que a reação se inicie, A é o fator de frequência (em 1/s), relacionado com a frequência com que as moléculas serão transformadas, R é a constante universal dos gases e T é a temperatura (em K).

$$k(t) = Ae^{-\frac{E}{RT(t)}} \tag{5-49}$$

Desta forma, combinando-se as equações 5-47 à 5-49, obtém-se uma expressão para a fração convertida de material em função da história térmica $T(t)$ e das características cinéticas do material, dadas por sua energia de ativação e fator de frequência (equação 5-50).

$$F(t) = 1 - e^{-\int_0^t Ae^{-\frac{E}{RT(t)}}\, dt} \tag{5-50}$$

Tomando como premissa que a história térmica de uma bacia pode ser decomposta em N períodos com taxa de aquecimento constante, a integral $I = \int_0^t k(t)\ dt$ pode ser reescrita pelas equações 5-51 e 5-52, onde $\frac{\Delta T}{\Delta t} = cte$ para cada período de tempo.

$$I = \sum_1^N \Delta I_i \tag{5-51}$$

$$\Delta I_i = \int_{t_{i-1}}^{t_i} Ae^{-\frac{E}{RT(t)}}\, dt \tag{5-52}$$

Braun e Burnham (1987) apresentam uma solução analítica para a equação 5-52, dada pelas equações 5-53 à 5-56.

$$\Delta I_i = \begin{cases} (t_i - t_{i-1})Ae^{-\frac{E}{RT_i}} & \text{se } T_i = T_{i-1} \\ (I_i - I_{i-1})H_i & \text{se } T_i \neq T_{i-1} \end{cases} \tag{5-53}$$

$$H_i = \frac{t_i - t_{i-1}}{T_i - T_{i-1}} \tag{5-54}$$

$$I_i = T_i \left(1 - \frac{(\frac{E}{RT_i})^2 + a_1(\frac{E}{RT_i}) + a_2}{(\frac{E}{RT_i})^2 + b_1(\frac{E}{RT_i}) + b_2} \right) A e^{-\frac{E}{RT_i}} \quad (5\text{-}55)$$

$$a_1 = 2,334733 \;\; a_2 = 0,250621 \;\; b_1 = 3,330657 \;\; b_2 = 1,681534 \quad (5\text{-}56)$$

A equação 5-50 é uma aproximação que considera que a reação pode ser modelada com apenas uma energia de ativação. Na prática, a matéria orgânica original é composta por diversos tipos de compostos, cada qual necessitando de uma quantidade de energia diferente para que sua quebra seja iniciada. Desta forma, os modelos adotados consideram a existência de várias reações cinéticas ocorrendo em paralelo, cada qual contando com um fator de frequência e uma energia de ativação[1]. Cada uma destas reações em paralelo possui um peso f, também denominado de fator estequiométrico, que indica a proporção do material original que é convertido por esta energia de ativação.

Desta forma, considerando-se a existência de n reações em paralelo, cada qual com peso f_j, energia de ativação E_j e fator de frequência A_j, a equação 5-48 pode ser reescrita pela equação 5-57 com o valor de $\frac{w_j}{w_{j0}}$ avaliado da mesma maneira que a utilizada para solucionar a equação 5-47, substituindo-se os valores de A e E pelos valores de A_j e E_j.

$$F = \sum_{j=1}^{N} f_j \left(1 - \frac{w_j}{w_{j0}} \right) \quad (5\text{-}57)$$

A figura 5.6 apresenta um histograma identificando os diversos valores de energia de ativação e seus pesos para o conjunto de reações em paralelo utilizadas no cálculo da reflectância da vitrinita pelo modelo *EasyRo* proposto por Sweeney e Burnham (1990). Após o cálculo de F, a fração convertida é então correlacionada com a reflectância da vitrinita através da equação 5-58.

$$Ro(\%) = e^{(-1,6 + 3,7F)} \quad (5\text{-}58)$$

Uma das grandes vantagens deste modelo consiste em sua adequação para uso com taxas de aquecimento de diversas ordens de grandeza, tais como deposições, intrusões e estudos de laboratório, cujas taxas de aquecimento são da ordem de grandeza de $^{\circ}C/M.a$, $^{\circ}C/ano$ e $^{\circ}C/hora$, respectivamente.

O mesmo método de solução pode ser utilizado para considerar cinéticas composicionais, onde o querogênio presente na rocha geradora é transformado

[1]Na prática, a maioria dos modelos utilizam "n" energias de ativação, mas apenas um fator de frequência.

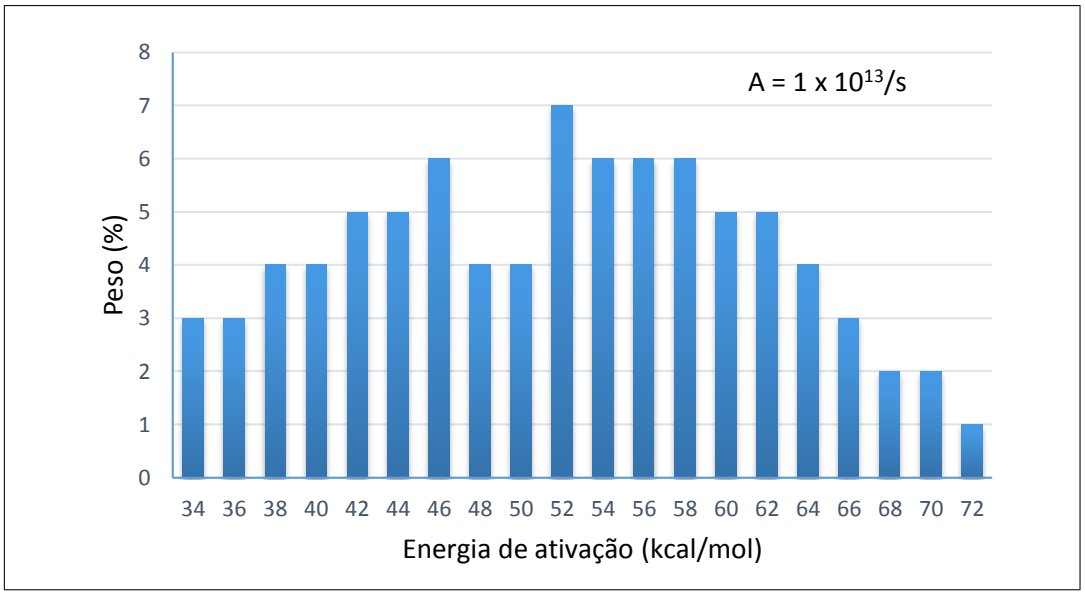


Figura 5.6: Histograma com energias de ativação utilizadas no modelo *Easy Ro*. Adaptado de Sweeney e Burnham (1990).

em vários componentes. A figura 5.7 ilustra o histograma de energias de ativação para um modelo cinético com 14 componentes. Ao aplicar a equação 5-57, cada energia de ativação, de cada componente, pode ser considerada como uma reação em paralelo. O único cuidado adicional necessário consiste em contabilizar a fração convertida por componente.

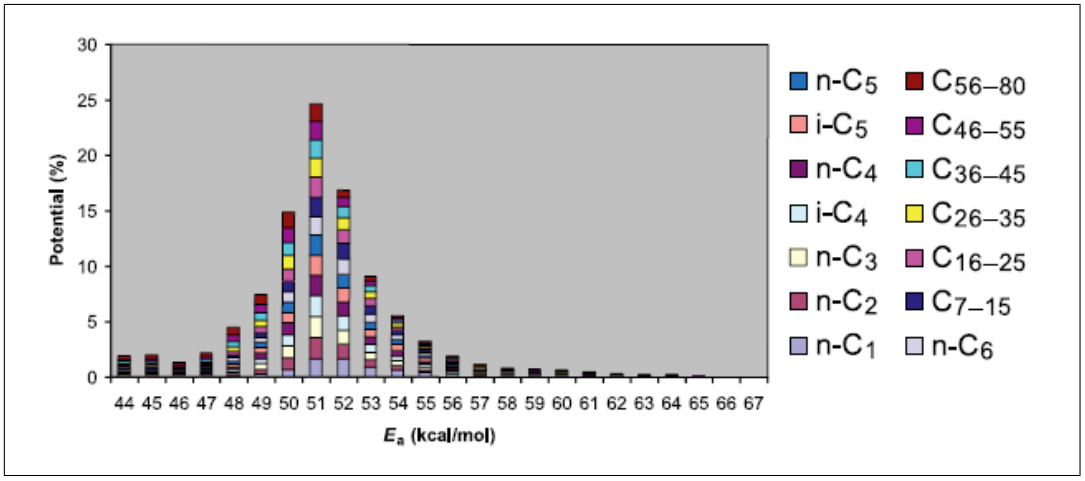


Figura 5.7: Exemplo de cinética composicional. Adaptado de Di Primio e Horsfield (2006).

5.3.4 Acoplamento

A figura 5.8 apresenta uma organização esquemática dos processos necessários para a implementação do simulador de bacias proposto, em duas versões. Na primeira (figura 5.8(a)), o cálculo de condutividade térmica depende

apenas da porosidade e da temperatura. Na segunda (figura 5.8(b)), a condutividade depende também da fração de óleo e de gás convertidas (F_o e F_g), sendo voltado a cenários de gás de folhelho.

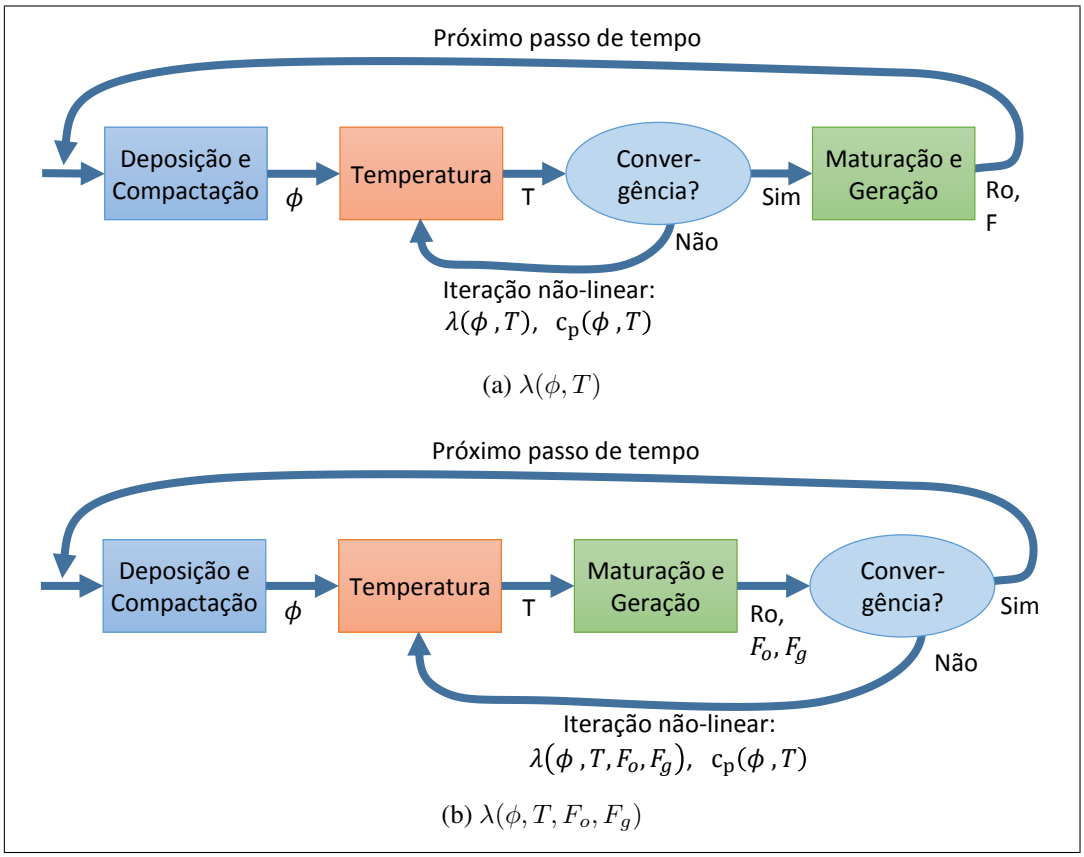


Figura 5.8: Acoplamento entre físicas para modelagem de bacias. (a) Condutividade térmica em função da porosidade e da temperatura; (b) Condutividade térmica em função da porosidade, temperatura, fração de óleo e fração de gás.

Em ambos os cenários, o primeiro passo consiste no processo de deposição e compactação, responsável pela evolução da malha no tempo e pelo cálculo da porosidade (ϕ) de cada camada. A seguir, são executados os cálculos de temperatura seguidos pelo cálculo de maturação e geração.

No primeiro cenário, o cálculo de temperatura é iterado para solução do problema não-linear e, uma vez obtida a convergência, são efetuados os cálculos de Ro para todas as camadas e da fração convertida F para as camadas geradoras.

No segundo cenário, executa-se um passo do cálculo de temperatura, seguido pelo cálculo de geração para determinação da fração de óleo e gás convertidos nas camadas geradoras. Neste cenário a iteração linear engloba os dois processos, já que a condutividade térmica leva em consideração também a presença de óleo e gás nas camadas.

6
Testes

Este capítulo apresenta um conjunto de testes implementados com o intuito de avaliar a corretude e a expressividade do *framework* GeMA, conforme sumarizado na seção 1.3.

Os testes apresentados foram selecionados para, em conjunto, fornecerem um visão global do *framework*, ilustrando suas principais características. Os modelos matemáticos adotados foram apresentados no capítulo 5. Embora alguns testes apresentem validações dos resultados das simulações, comparando-as com resultados analíticos ou com resultados publicados, é importante ressaltar que o objetivo destes testes não é validar os resultados obtidos, mas sim o *framework*, ou seja, validar se o mesmo atende aos requisitos de suporte a simulações multifísicas apresentados na seção 1.1.

6.1
Temperatura

6.1.1
Condução de calor em uma placa

Neste primeiro exemplo, uma placa quadrada de metal com $1\ m$ de largura é submetida a uma temperatura fixa de $500\ ^{\circ}C$ em sua borda superior e a uma temperatura de $100\ ^{\circ}C$ nas demais bordas (figura 2.1 (b)). Em um primeiro passo, deseja-se determinar a temperatura, em equilíbrio térmico, de cada ponto da placa. A seguir, considera-se que a placa inicialmente encontra-se a uma temperatura uniforme de $0\ ^{\circ}C$ quando então as condições de contorno anteriores são aplicadas. Neste novo cenário, deseja-se estudar a evolução da temperatura da placa no tempo. Em ambos os casos, são considerados os seguintes parâmetros para o material: $\lambda = 10\ W/(m.^{\circ}C)$, $\rho = 2\ kg/m^3$ e $c_p = 10\ J/(kg.^{\circ}C)$.

A solução de ambos os problemas pode ser feita através do método de elementos finitos, com uso do modelo discreto apresentado na seção 5.1. A equação 6-1, apresentada por Holman (1989 apud Lewis *et al*, 2004), onde w e h são, respectivamente, a largura e altura da placa, e T_{side} e T_{top} são as temperaturas nas bordas laterais e superior, fornece uma solução analítica para o problema inicial em regime permanente, e pode ser utilizada para validar os resultados obtidos pelo

método numérico.

$$T(x,y) = T_{side} + (T_{top} - T_{side})\frac{2}{\pi}\sum_{n=1}^{\infty}\frac{(-1)^{n+1}+1}{n}\sin\left(\frac{n\pi x}{w}\right)\frac{\sinh\left(\frac{n\pi y}{w}\right)}{\sinh\left(\frac{n\pi h}{w}\right)} \quad (6\text{-}1)$$

A mesma discretização espacial pode ser utilizada para as duas simulações. O que as diferencia é o método de solução adotado para cada uma delas. Os *scripts* 10 a 12 apresentam os dados do modelo, onde o domínio espacial é discretizado em uma malha com 800 triângulos. É interessante notar que, neste modelo, ao invés da malha ser fornecida através de uma lista de coordenadas de vértices e de uma lista de triângulos, estas são geradas dinamicamente por uma função em Lua que cria as referidas tabelas.

***Script* 10** Modelo de simulação para condução de calor em uma placa (1/3)

```
1  -- Parâmetros do modelo
2  local w   = 100  -- Largura da placa em cm
3  local h   = 100  -- Altura da placa em cm
4  local t   = 1    -- Espessura da placa em cm
5  local nx  = 21   -- Número de nós na direção x
6  local ny  = 21   -- Número de nós na direção y
7  local k   =  10  -- Condutividade em W/(m.degC)
8  local rho = 2    -- Densidade em kg/m3
9  local cp  = 10   -- Calor específico em J/(kg.degC)
10 local Tt  = 500  -- Temperatura no topo em degC
11 local Ts  = 100  -- Temperatura na base e nas laterais em degC
12
13 -- Função analítica para cálculo de T(x, y)
14 local function T(x, y)
15   local sum = 0
16   for i=1, 200 do
17     local u   = i*math.pi/w
18     local num = ((-1)^(i+1) + 1) * math.sin(u*x) * math.sinh(u*y)
19     local den = i * math.sinh(u*h)
20     sum = sum + num/den
21   end
22   return Ts + (Tt-Ts) * (2/math.pi) * sum
23 end
24
25 -- Função do usuário para cálculo de erro
26 NodeFunction { id = 'errf',
27                parameters = { {src = 'Tana', unit = 'degC'},
28                               {src = 'T',    unit = 'degC'},
29                             },
30                method = function(Tana, T)
31                  return math.abs(Tana - T)
32                end
33              }
```

***Script* 11** Modelo de simulação para condução de calor em uma placa (2/3)

```
1  -- Funções auxiliares para criar as listas de nós e triângulos de
2  -- um malha contendo nx/ny nós em cada direção, espaçados de dx/dy.
3  local function createNodeList(dx, nx, dy, ny)
4    local nodes = {} -- {x, y, Tana}
5    for y=0, ny-1 do
6      for x=0, nx-1 do
7        nodes[#nodes+1] = {x*dx, y*dy, T(x*dx, y*dy)}
8      end
9    end
10   return nodes
11 end
12
13 local function createElementList(dx, nx, dy, ny)
14   local N = function (r,c) return (r-1)*nx + c end
15   local cells = {}
16   for y=1, ny-1 do
17     for x=1, nx-1 do
18       cells[#cells+1] = {N(y, x),   N(y,   x+1), N(y+1, x)}
19       cells[#cells+1] = {N(y, x+1), N(y+1, x+1), N(y+1, x)}
20     end
21   end
22   return cells
23 end
24
25 -- Funções auxiliares para criar as listas de arestas usadas
26 -- na definição das condições de contorno
27 local function topBorder(nx, ny)
28   local list = {}
29   local nelem = (nx-1)*(ny-1)
30   for i=nelem-nx+2, nelem do
31     list[#list+1] = {i*2, 2} -- {Id da célula, Id da borda}
32   end
33   return list
34 end
35
36 local function otherBorders(nx, ny)
37   local list = {}
38   -- Borda esquerda
39   for i=1, ny-1 do
40     list[#list+1] = {(i-1)*(nx-1)*2+1, 3}
41   end
42   -- Borda inferior
43   for i=1, nx-1 do
44     list[#list+1] = {i*2-1, 1}
45   end
46   -- Borda direita
47   for i=1, ny-1 do
48     list[#list+1] = {i*(nx-1)*2, 1}
49   end
50   return list
51 end
52
53 --  Variáveis de estado
54 StateVar {id = 'T', description = 'Temperature', unit = 'degC'}
```

***Script* 12** Modelo de simulação para condução de calor em uma placa (3/3)

```
-- Conjunto de propriedades associadas aos elementos do modelo
PropertySet {
  id          = 'MatProp',
  typeName    = 'GemaStdPropertySet',
  description = 'Material parameters',
  properties  = {
    {id = 'k',   description = 'Conductivity',  unit = 'W/(m.K)'},
    {id = 'rho', description = 'Density',       unit = 'kg/m3'},
    {id = 'cp',  description = 'Specific heat', unit = 'J/(kg.K)'},
    {id = 't',   description = 'Element thickness', unit = 'cm'},
  },
  values = { {k = k, rho = rho, cp = cp, t = t} }
}

-- Definição da malha
Mesh {
  -- Atributos gerais
  id          = 'plateMesh',
  typeName    = 'GemaStdMesh.elem',
  description = 'Plate mesh discretization',

  -- Número de dimensões do problema e unidade usada p/ coordenadas
  coordinateDim  = 2,
  coordinateUnit = 'cm',

  -- Lista de variáveis de estado e atributos armazenados por nó e
  -- propriedades armazenadas por elemento da malha
  stateVars      = {'T'},
  cellProperties = {'MatProp'},
  nodeAttributes = {
    {id = 'Tana', description = 'Analytical value', unit = 'degC'},
    {id = 'Err',  description = 'Error', unit = 'degC',
                  functions = true, defVal = 'errf', format = '10.4'}
  },

  -- Geometria
  nodeData = createNodeList(w/(nx-1), nx, h/(ny-1), ny),
  cellData = {{cellType = 'tri3', MatProp = 1,
               cellList = createElementList(w/(nx-1),nx,h/(ny-1),ny)}},
  boundaryEdgeData = {
    {id = 'bottom and sides', cellList = otherBorders(nx,ny)},
    {id = 'top',              cellList = topBorder(nx,ny)},
  }
}

-- Condições de contorno
BoundaryCondition {
  id   = 'Border temperature',
  type = 'fixed temperature',
  mesh = 'plateMesh',
  properties  = { {id = 'T', unit = 'degC'} },
  nodeValues  = { {'bottom and sides', Ts}, -- Arestas, temperatura
                  {'top',              Tt} }
}
```

Da mesma maneira, as listas de arestas que definem as bordas do modelo, compostas por pares contendo o identificador de um elemento e o identificador de uma aresta deste elemento, também são criadas por chamadas de funções.

Para facilitar a validação dos resultados em regime permanente, foram adicionados dois atributos ao modelo. O primeiro, `Tana`, armazena a temperatura calculada analiticamente para cada nó da malha, inicializado durante sua construção. O segundo, `Err`, é um atributo associado a uma função do usuário, tendo por objetivo calcular a diferença entre a variável de estado `T`, cujo valor é calculado pela simulação, e o atributo `Tana`. Este cálculo é feito pela função `errf`, que recebe como parâmetros os valores de `T` e `Tana`.

O método de solução utilizado na simulação em regime permanente é mostrado no *script* 13. Seu *script* de orquestração consiste na execução do processo de análise por elementos finitos, parametrizado pela física de cálculo de temperaturas, e pela execução de um processo para salvar a temperatura e o erro calculados. Cabe salientar que o erro é calculado pela função `errf` apenas no momento em que o atributo `Err` é consultado para ser salvo no arquivo.

***Script* 13** Método de solução para condução de calor em uma placa em regime permanente

```
1  -- Resolvedor numérico utilizado
2  NumericalSolver {
3    id          = 'solver',
4    typeName    = 'GemaStdNumSolver',
5    description = 'Standard matrix solver',
6  }
7
8  -- Física de cálculo de temperatura
9  PhysicalMethod {
10   id       = 'HeatPhysics',
11   typeName = 'GemaStdHeatPhysics',
12   type     = 'fem',
13
14   mesh               = 'plateMesh',
15   boundaryConditions = {'Border temperature'},
16 }
17
18 -- Script de orquestração
19 function ProcessScript()
20   -- Executa uma análise de elementos finitos, recebendo como
21   -- parâmetros uma física e um resolvedor numérico
22   fem.solve({'HeatPhysics'}, 'solver')
23
24   -- Salva a variável de estado 'T', e o atributo 'err' associados
25   -- com nós da malha 'plateMesh' no arquivo resultados.nf
26   utils.saveMeshFile('plateMesh', './resultados.nf', 'nf',
27                      {'T', 'Err'})
28 end
```

A temperatura calculada é exibida na figura 6.1(a), enquanto o erro é exibido na figura 6.1(b). Podemos observar que a solução numérica obtida concorda com a solução analítica, sendo que os maiores valores de erro encontram-se justamente nos cantos onde a condição de contorno é descontínua e o próprio resultado analítico não é igual às condições de contorno prescritas.

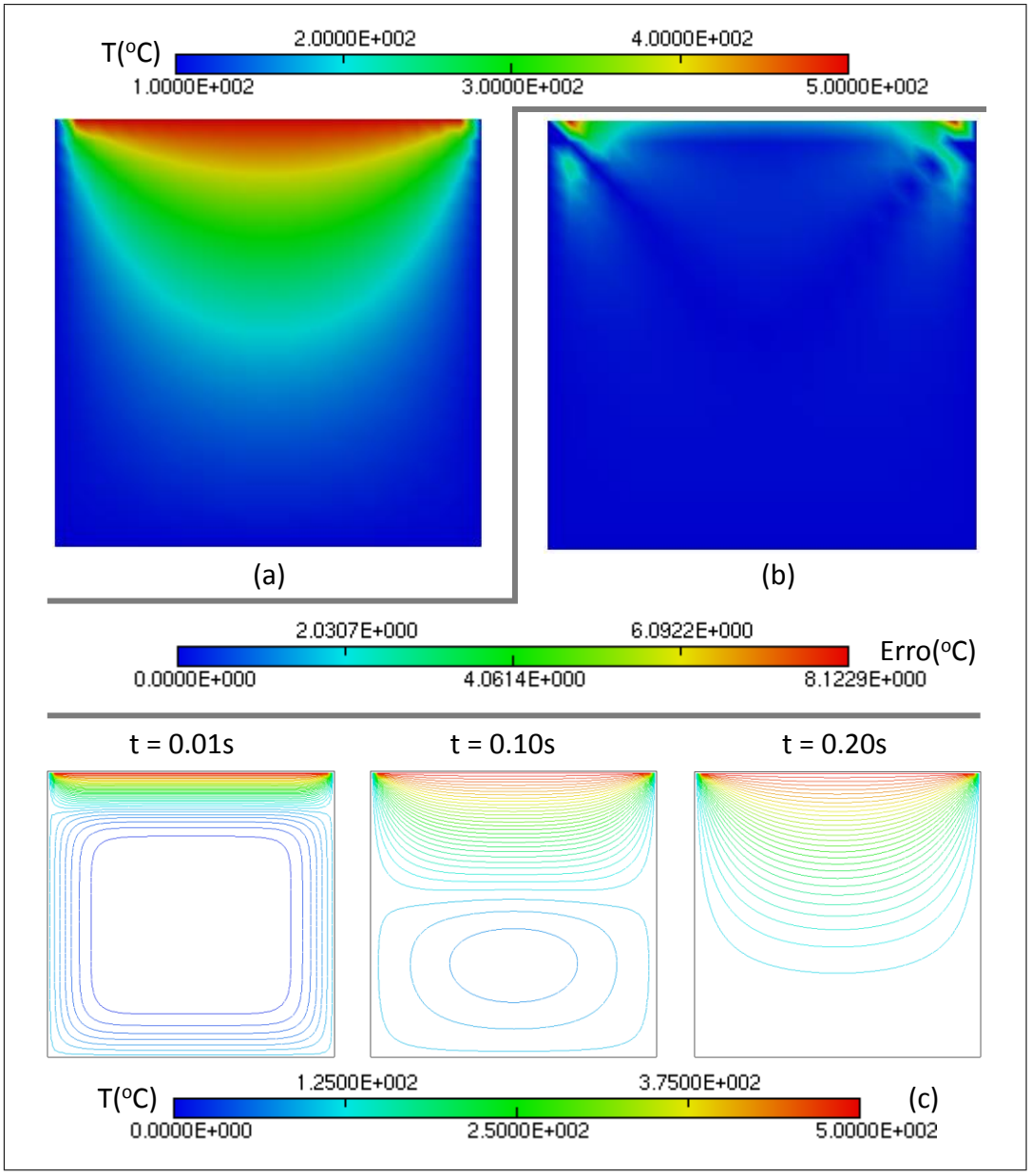


Figura 6.1: Resultados da simulação de condução de calor em uma placa. (a) Temperatura calculada em regime permanente; (b) Diferença entre as soluções analítica e por elementos finitos; (c) Distribuição de temperaturas em regime transiente para momentos selecionados.

Para a solução do problema transiente, é necessário apenas que a função de orquestração seja alterada conforme apresentado no *script* 14, introduzindo-se um laço temporal que controla a execução da análise por elementos finitos e o salvamento dos resultados obtidos para cada passo de tempo. As temperaturas

calculadas são apresentadas na figura 6.1(c), para três instantes de tempo distintos, através de um conjunto de isotermas. Os resultados obtidos são coerentes com os resultados apresentados por Lewis *et al* (2004) para o mesmo problema.

***Script* 14** *Script* de orquestração para condução de calor em uma placa em regime transiente

```
-- Script de orquestração
function ProcessScript()
  -- Cria o arquivo que irá receber os resultados salvos
  local file = utils.prepareMeshFile('plateMesh', './resultados.nf',
                                     'nf', {'T'})

  -- Passo de tempo de 1ms. Duração total da simulação de 0.2s
  local dt     = 0.001
  local endt   = 0.2
  local nsteps = endt / dt

  -- Inicializa análise transiente
  local solver = fem.initTransientSolver({'HeatPhysics'}, 'solver')

  -- Loop temporal
  for i=1, nsteps do
    -- Calcula temperatura após um passo de tempo = dt
    fem.transientStep(solver, dt)

    -- Atualiza o tempo atual da simulação
    setCurrentTime(i*dt)

    -- Adiciona resultados ao arquivo
    utils.addResultToMeshFile(file, i*dt)
  end
  utils.closeMeshFile(file)
end
```


6.1.2 Condução transiente de calor em uma camada sedimentar

Este teste apresenta um cenário básico para a modelagem de bacias, consistindo em um conjunto de camadas sedimentares sujeitas a uma temperatura fixa na superfície e a um fluxo térmico na base, cujo valor q_1 é alterado no instante $t = 0$ para um valor q_2. Assume-se que não há fluxo térmico lateral.

Em $t < 0$ o conjunto de camadas encontra-se em equilíbrio térmico. As temperaturas neste estado de equilíbrio são a condição inicial para a análise transiente. Este cenário pode ser modelado facilmente pelo *script* de orquestração através da execução de uma análise em regime permanente, usando o fluxo térmico inicial q_1 como condição de contorno, seguida pela análise transiente, agora com a nova condição de contorno q_2.

O modelo simulado é composto por 10 camadas, cada uma com $1\ km$ de espessura, discretizado através de um quadrilátero por camada. Todas as camadas

compartilham o mesmo conjunto de parâmetros para permitir a comparação dos resultados obtidos com a solução analítica dada pela equação 6-2, apresentada por Hantschel e Kauerauf (2009), onde h é a espessura total do modelo. Parâmetros utilizados: $\lambda = 2\ W/(m.^\circ C)$, $\rho = 2700\ kg/m^3$, $c_p = 860\ J/(kg.^\circ C)$, temperatura na superfície $T_s = 0\ ^\circ C$ e fluxos térmicos $q_1 = 50\ mW/m^2$ e $q_2 = 100\ mW/m^2$.

$$T(z,t) = T_s + \frac{q_2}{\lambda}z + \frac{q_1 - q_2}{\lambda}\sum_{n=-\infty}^{\infty}\frac{(-1)^n}{h\mu_n^2}\sin(\mu_n z)e^{-\frac{\mu_n^2 \lambda t}{\rho c_p}} \quad (6\text{-}2)$$

$$\mu_n = \frac{\pi(n+1/2)}{h} \quad (6\text{-}3)$$

A descrição do modelo é muito similar àquela apresentada no exemplo anterior, portanto no *script* 15 são apresentadas apenas as condições de contorno de Neumann referentes ao fluxo térmico na base das camadas e as definições das físicas utilizadas. O *script* de orquestração utilizado é dado pelo *script* 2 (seção 4.3).

A figura 6.2 apresenta o perfil de temperatura no tempo para profundidades selecionadas da bacia. Valores foram calculados a cada 10.000 anos em um intervalo total de 10 milhões de anos. O maior erro analítico observado é de 1,2% para t igual a 0.01 Myr, e menor do que 0,3% para t maior do que 0,1 Myr.

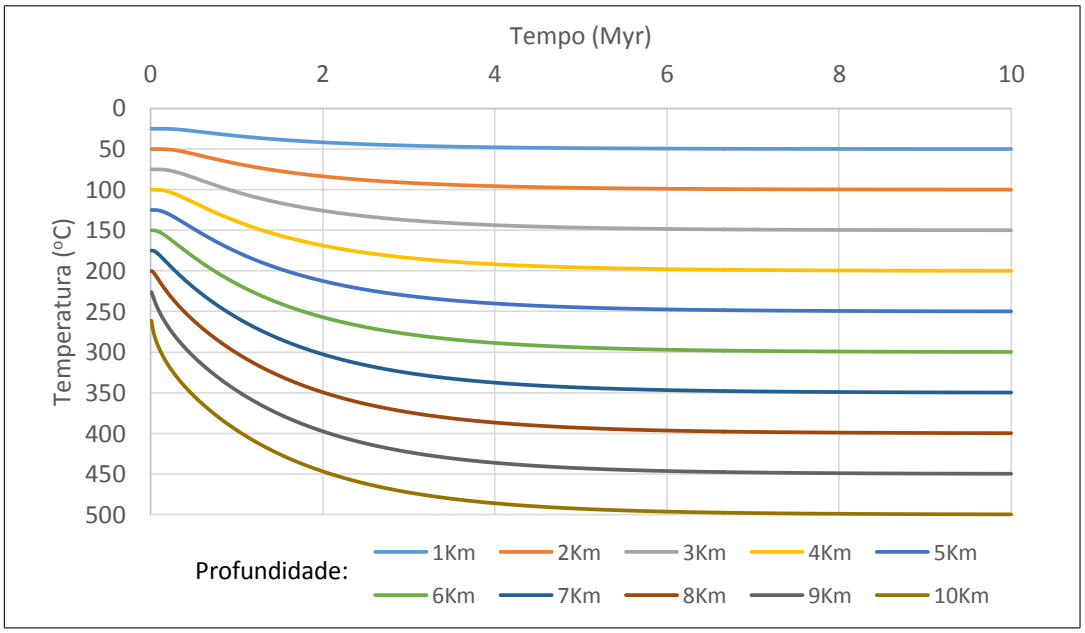


Figura 6.2: Resultados da simulação de condução transiente de calor em uma camada sedimentar.

***Script* 15** Excertos do modelo de simulação para condução transiente de calor em uma camada sedimentar.

```
-- Condição de contorno: Fluxo térmico em t < 0
BoundaryCondition {
  id   = 'Basal heat flow t<0',
  type = 'fixed heat flux',
  mesh = 'mesh',
  properties  = {
    {id = 'q',  description = 'Heat flow', unit = 'mW/m2'},
  },
  edgeValues = {    -- Valor negativo: fluxo na
    {'bottom', -50} -- direção oposta à normal
  }
}

-- Condição de contorno: Fluxo térmico em t > 0
BoundaryCondition {
  id   = 'Basal heat flow t>0',
  type = 'fixed heat flux',
  mesh = 'mesh',
  properties  = {
    {id = 'q',  description = 'Heat flow', unit = 'mW/m2'},
  },
  edgeValues = { {'bottom',  -100} }
}

-- Física usada para cálculo da temperatura inicial
PhysicalMethod {
  id       = 'HeatPhysics t<0',
  typeName = 'GemaStdHeatPhysics',
  type     = 'fem',
  mesh     = 'mesh',
  boundaryConditions = {'Surface temperature', 'Basal heat flow t<0'},
}

-- Física usada para cálculo transiente da temperatura com nova
-- condição de contorno
PhysicalMethod {
  id       = 'HeatPhysics t>0',
  typeName = 'GemaStdHeatPhysics',
  type     = 'fem',
  mesh     = 'mesh',
  boundaryConditions = {'Surface temperature', 'Basal heat flow t>0'},
}
```

6.1.3 Solidificação

Este teste apresenta um cenário de mudança de fase para um volume líquido submetido a condições de resfriamento. Além de ser um cenário básico para o tratamento da solidificação de intrusões magmáticas durante a história de uma bacia sedimentar, este teste também tem como objetivo mostrar a flexibilidade obtida através da possibilidade de utilizar funções do usuário na definição de propriedades

e atributos, e a possibilidade de inclusão de laços não-lineares através do uso do orquestrador.

Conforme apresentado na seção 5.1, a mudança de fase pode ser tratada através do conceito de calor específico efetivo. A equação 5-14 pode ser implementada através de uma função do usuário utilizada para modificar o valor de c_p de acordo com a temperatura do elemento.

O cenário simulado é apresentado na figura 6.3(a), onde um líquido, inicialmente a uma temperatura de $0\ ^{\circ}C$, é submetido a uma temperatura de $-0.15\ ^{\circ}C$ em sua face direita e a $-45.0\ ^{\circ}C$ na face esquerda, mantendo-se o topo e a base isolados. O *script* 16 apresenta a lista de propriedades utilizada e a função do usuário responsável por retornar o valor do calor específico efetivo. Os demais parâmetros do modelo são muito similares aos utilizados nos exemplos anteriores.

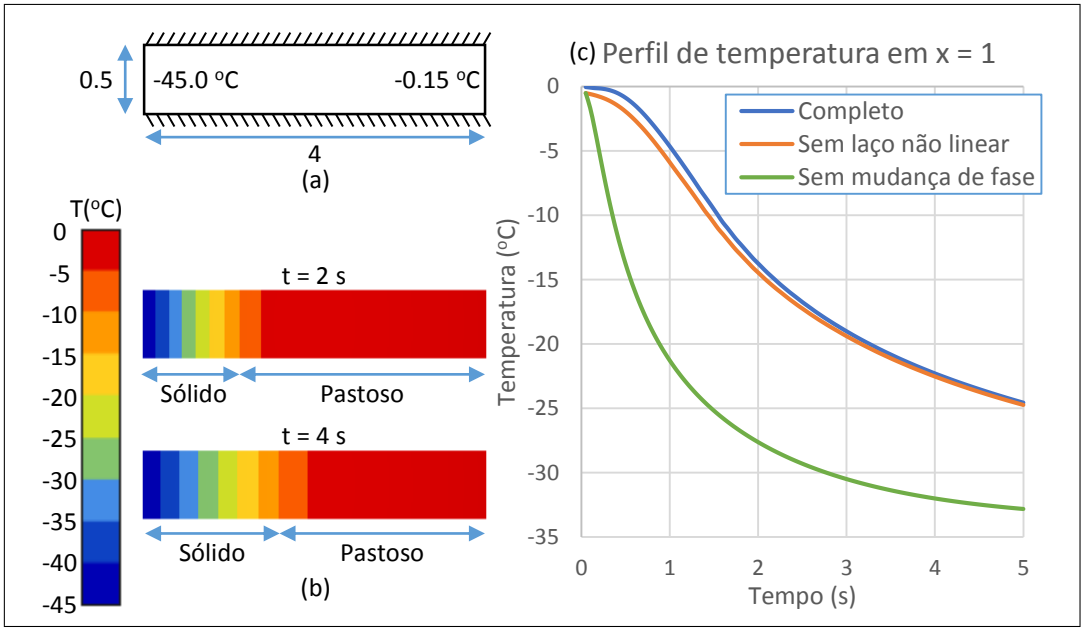


Figura 6.3: Modelo e resultados da simulação de solidificação. (a) Cenário simulado; (b) Distribuição de temperatura em instantes selecionados; (c) Evolução da temperatura ao longo do tempo para o ponto de coordenada (1.0, 0.25).

A função `ceff` recebe como parâmetros a temperatura e todas as propriedades do elemento necessárias para a avaliação da equação 5-14. Como a temperatura é um valor associado aos nós da malha, o valor recebido pela função é o resultado da interpolação dos valores nodais no ponto de integração onde o valor do calor específico está sendo avaliado.

Como neste modelo o calor específico é uma função da temperatura, o mesmo passa a apresentar um comportamento não-linear durante a etapa de solidificação. A solução adotada consiste na inclusão de um laço no orquestrador para efetuar passos não-lineares durante cada passo de tempo, até que a convergência seja alcançada. Em geral, 3 passos iterativos são suficientes, não sendo necessária a utilização de

modelos mais eficientes para a solução do sistema. O *script* 17 apresenta o *script* de orquestração utilizado.

***Script* 16** Propriedades e funções do usuário utilizados no modelo de solidificação.

```
-- Função do usuário para cálculo do calor específico efetivo
CellFunction { id = 'ceff',
               parameters = { {src = 'T',  unit = 'degC'},
                              {src = 'Ts', unit = 'degC'},
                              {src = 'Tl', unit = 'degC'},
                              {src = 'cs', unit = 'J/(kg.degC)'},
                              {src = 'cl', unit = 'J/(kg.degC)'},
                              {src = 'cf', unit = 'J/(kg.degC)'},
                              {src = 'L',  unit = 'J/kg'},
                            },
               method = function(T, Ts, Tl, cs, cl, cf, L)
                 if T < Ts then
                   return cs
                 elseif T >= Tl then
                   return cl
                 else
                   return cf + L / (Tl - Ts)
                 end
               end
             }

-- Conjunto de propriedades
PropertySet {
  id          = 'MatProp',
  typeName    = 'GemaStdPropertySet',
  description = 'Material parameters',
  properties  = {
    {id = 'k',   description = 'Conductivity', unit = 'W/(m.degC)'},
    {id = 'rho', description = 'Density',      unit = 'kg/m3'},
    {id = 'cp',  description = 'Effective specific heat',
                 unit = 'J/(kg.degC)', functions = true},
    {id = 'L',   description = 'Latent heat of solidification',
                 unit = 'J/kg'},
    {id = 'cl',  description = 'Spec. heat for liquid phase',
                 unit = 'J/(kg.degC)'},
    {id = 'cs',  description = 'Spec. heat for solid phase',
                 unit = 'J/(kg.degC)'},
    {id = 'cf',  description = 'Spec. heat for freezing interval',
                 unit = 'J/(kg.degC)'},
    {id = 'Ts',  description = 'Solid temperature', unit = 'degC'},
    {id = 'Tl',  description = 'Liquid temperature', unit = 'degC'},
  },
  values = { {k = 1.0, rho = 1.0, cp = 'ceff', L = 70.26, cl = 1.0,
              cs = 1.0, cf = 1.0, Ts = -10.15, Tl = -0.15}
  }
}
```

***Script* 17** *Script* de orquestração utilizado no modelo de solidificação.

```
-- Script de orquestração
function ProcessScript()
  -- Cria o arquivo que irá receber os resultados salvos
  local file = utils.prepareMeshFile('mesh', './resultados.nf',
                                     'nf', {'T'})

  -- Passo no tempo de 50ms. Duração total da simulação de 5s
  local dt     = 0.05
  local endt   = 5
  local nsteps = endt / dt

  -- Inicializa análise transiente não linear
  local solver = fem.initTransientSolver({'HeatPhysics'}, 'solver',
                                         {}, true)

  -- Loop temporal
  for i=1, nsteps do
    -- Loop não linear
    local j = 1
    while j <= 10 do
      -- Executa passo
      fem.transientLinearStep(solver, dt, j)

      -- Calcula o resíduo
      r = fem.transientLinearResidual(solver, dt)

      -- Convergiu?
      if r <= 1e-3 then
        break
      end
      j = j + 1
    end
    assert(j <= 10, 'Failed to achieve convergence')

    -- Adiciona resultados ao arquivo
    utils.addResultToMeshFile(file, i*dt)
  end
  utils.closeMeshFile(file)
end
```

A figura 6.3(b) apresenta o perfil de temperatura nos instantes $t = 2\ s$ e $t = 4\ s$. A figura 6.3(c) apresenta o perfil de temperatura no tempo de um ponto localizado na coordenada (1, 0.25), comparando o resultado obtido com uma segunda versão onde a mudança de fase não é considerada, e com uma terceira, onde o laço não-linear não é executado. Os resultados obtidos são compatíveis com os resultados apresentados por Lewis *et al* (2004) para o mesmo problema.

6.2
Acoplamento Tensão - Temperatura

Este teste apresenta um cenário multifísico com acoplamento entre cálculos de temperatura e tensões/deslocamentos em estado plano de tensões. Seu objetivo é explorar a possibilidade de acoplamento forte entre os dois modelos matemáticos e o uso de diversas físicas GeMA para modularizar a solução do prolema.

O cenário simulado é apresentado na figura 6.4(a), onde uma barra é fixa em seu lado direito e livre para se movimentar nas demais direções. A borda superior é mantida em uma temperatura constante de $100.0\ ^{\circ}C$, enquanto as demais bordas são isoladas, com exceção de um pequeno trecho na borda inferior que aquece a barra com um fluxo térmico de $100W/m^2$. A metade inferior da barra possui um coeficiente de expansão térmica α cinco vezes maior do que a metade superior, o que a obriga a curvar-se para cima quando aquecida na parte inferior. Os *scripts* 18 e 19 apresentam a descrição do modelo de simulação, com suas propriedades físicas e condições de contorno.

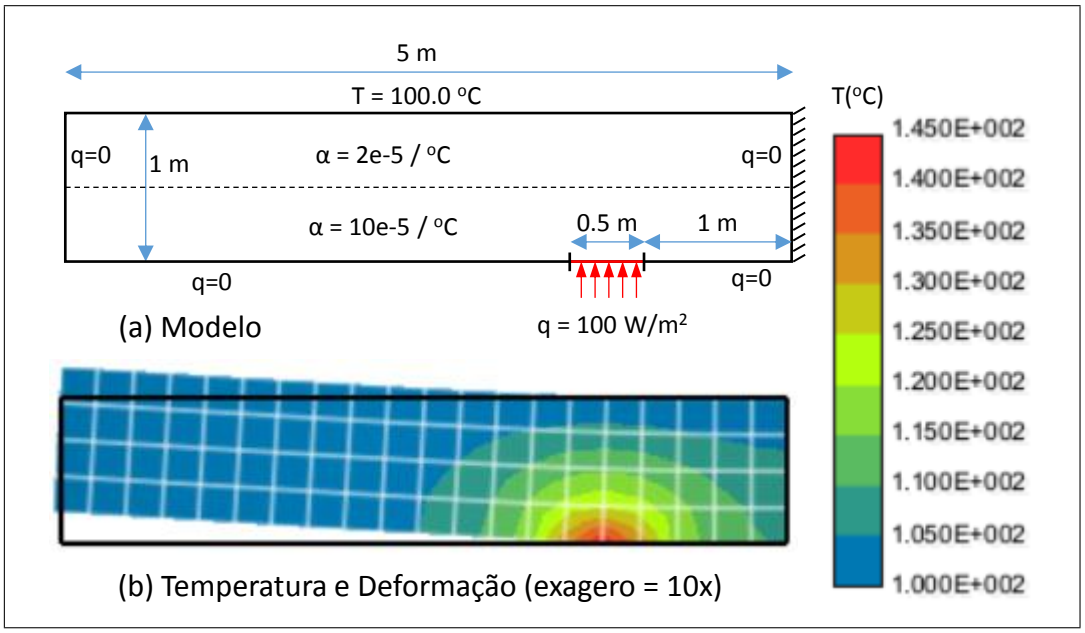


Figura 6.4: Modelo, temperatura e deformação para simulação acoplada tensão - temperatura. (a) Modelos simulado; (b) Resultados calculados.

A equação 5-27 apresenta o modelo matemático necessário para a construção de um único sistema linear de equações que acopla os cálculos de deslocamento e temperatura. Esta solução pode ser implementada através de uma única física GeMA que calcule as matrizes locais de cada elemento, levando em consideração todas as especificidades dos cálculos de temperatura, deslocamento e dos termos de acoplamento.

Considerando a forma da matriz de rigidez acoplada e a possibilidade de reuso das físicas previamente existentes para cálculos independentes de temperatura

***Script* 18** Modelo para simulação acoplada tensão - temperatura (1/2).

```
-- Variáveis de estado
StateVar{id = 'T',  description = 'Temperature', unit = 'K'}
StateVar{id = 'ux', description = 'Displacements in the X direction'}
StateVar{id = 'uy', description = 'Displacements in the Y direction'}

-- Funções do usuário para cálculo de tensões principais
CellFunction { id = 'sigma1',
               parameters = { {src = 'sxx'},
                              {src = 'syy'},
                              {src = 'sxy'},
                            },
               method = function(sxx, syy, sxy)
                 local a = 0.5*(sxx+syy)
                 local b = math.sqrt((sxx-syy)^2/4 + sxy^2)
                 return a+b
               end
             }
CellFunction { id = 'sigma2',
               parameters = { {src = 'sxx'},
                              {src = 'syy'},
                              {src = 'sxy'},
                            },
               method = function(sxx, syy, sxy)
                 local a = 0.5*(sxx+syy)
                 local b = math.sqrt((sxx-syy)^2/4 + sxy^2)
                 return a-b
               end
             }

-- Propriedades associadas aos elementos do modelo
PropertySet {
  id          = 'MatProp',
  typeName    = 'GemaStdPropertySet',
  description = 'Material parameters',
  properties  = {
    {id = 'k',     description = 'Conductivity', unit = 'W/(m.K)'},
    {id = 't',     description = 'Element thickness', unit = 'm'},
    {id = 'E',     description = 'Elasticity modulus'},
    {id = 'nu',    description = 'Poisson ratio'},
    {id = 'alpha', description = 'Thermal expansion factor', unit = '1/K'},
  },
  values = {
    {id = 'normal', k = 1.0, t = 0.01, E = 1e6, nu = 0.3, alpha = 2e-5},
    {id = 'high',   k = 1.0, t = 0.01, E = 1e6, nu = 0.3, alpha = 10e-5},
  }
}
-- Nós e elementos da malha.
-- Conjunto completo de valores omitidos por brevidade
local meshNodes    = { {0.0, 0.0}, ... }
local meshElements = { { 1,  2, 23, 22, MatProp = 'high'},
                       ...
                       {43, 44, 65, 64, MatProp = 'normal'},
                       ...
                     }
```

***Script* 19** Modelo para simulação acoplada tensão - temperatura (2/2).

```
1  -- Definição da malha
2  Mesh {
3    -- Atributos gerais
4    id          = 'mesh',
5    typeName    = 'GemaStdMesh.elem',
6    description = 'Mesh discretization',
7
8    -- Número de dimensões do problema e unidade usada p/ coordenadas
9    coordinateDim  = 2,
10   coordinateUnit = 'm',
11
12   -- Variáveis de estado armazenadas por nó, propriedades armazenadas
13   -- por elemento e atributos armazenados por ponto de Gauss
14   stateVars       = {'T', 'ux', 'uy'},
15   cellProperties  = {'MatProp'},
16   gaussAttributes = {
17     {id = 's1', description = 'Sigma 1', functions = true, defVal = 'sigma1'},
18     {id = 's2', description = 'Sigma 2', functions = true, defVal = 'sigma2'}
19   },
20
21   -- Geometria
22   nodeData = meshNodes,
23   cellData = { {cellType = 'quad4', cellList = meshElements} },
24   boundaryEdgeData = {
25     {id = 'bottomHeatedBorder', cellList = { {15, 1}, {16, 1} } },
26     {id = 'topBorder',          cellList = { ... } },
27     {id = 'rightBorder',        cellList = { ... } },
28   }
29 }
30
31 -- Condições de contorno para temperatura e tensões
32 BoundaryCondition {
33   id   = 'Tbc1',
34   type = 'fixed temperature',
35   mesh = 'mesh',
36   properties  = { {id = 'T', unit = 'K'} },
37   nodeValues  = { {'topBorder', 100} },
38 }
39 BoundaryCondition {
40   id   = 'Tbc2',
41   type = 'fixed heat flux',
42   mesh = 'mesh',
43   properties  = { {id = 'q',  unit = 'W/m2'} },
44   edgeValues  = { {'bottomHeatedBorder', -100} },
45 }
46 BoundaryCondition {
47   id   = 'ubc',
48   type = 'node displacements',
49   mesh = 'mesh',
50   properties  = { {id = 'ux',  defVal = -9999},
51                   {id = 'uy',  defVal = -9999} },
52   nodeValues  = { {'rightBorder',   0.0, 0.0} },
53 }
```

e tensão, uma solução mais interessante consiste em seguir a receita proposta na figura 4.19 e compor o método de solução através do uso de três físicas GeMA que cooperam para a construção da matriz de rigidez.

Desta forma, o método de solução adotado reutiliza as físicas individuais de cálculo de temperatura e deslocamentos, adicionando uma terceira física responsável pelos termos de acoplamento necessários. Esta nova física herda da física padrão de cálculo de tensões para reutilizar as diversas funções de validação e inicialização da física, reimplementando apenas as funções de cálculo das matrizes locais e retorno dos graus de liberdade envolvidos.

As principais vantagens do uso de três físicas, ao invés de uma, consistem na simplificação do código gerado e na menor probabilidade de erros ao reutilizar código previamente testado. Isto diminui o tempo necessário para a implementação da simulação e facilita a manutenibilidade do sistema. Uma desvantagem desta abordagem consiste na necessidade de calcular a matriz de deformações-deslocamento (B) tanto na física de tensões quanto na física de acoplamento. Na prática, porém, o custo adicional é irrelevante no tempo total de simulação.

O *script* 20 apresenta o método de solução adotado. Os deslocamentos e a temperatura calculada são apresentados na figura 6.4(b).

A física adotada calcula as tensões σ_{xx}, σ_{yy} e σ_{xy} nos pontos de Gauss dos elementos. Funções do usuário podem ser adicionadas para cálculo das tensões principais σ_1 e σ_2, associando-as a novos atributos `s1` e `s2` criados no modelo. A figura 6.5 apresenta os resultados obtidos. Outros tipos de tensão, tais como tensões de von Misses, podem ser calculadas de maneira similar.

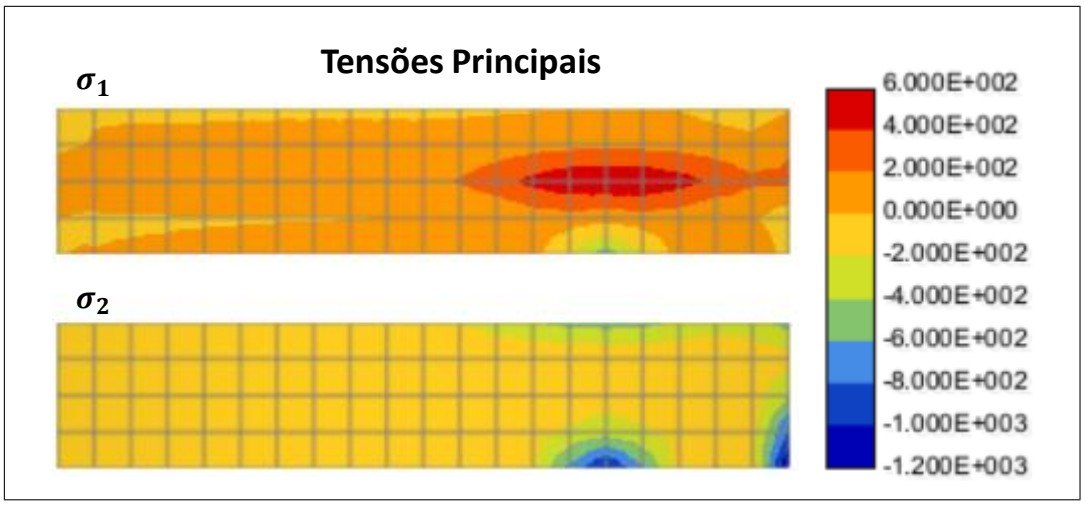


Figura 6.5: Tensões principais para simulação acoplada tensão - temperatura.

***Script* 20** Método de solução para simulação acoplada tensão - temperatura.

```
1  -- Resolvedor numérico utilizado
2  NumericalSolver {
3    id          = 'solver',
4    typeName    = 'GemaStdNumSolver',
5    description = 'Simple matrix solver',
6  }
7
8  -- Física para cálculo de temperatura
9  PhysicalMethod {
10   id       = 'HeatPhysics',
11   typeName = 'GemaStdHeatPhysics',
12   type     = 'fem',
13
14   mesh               = 'mesh',
15   boundaryConditions = {'Tbc1', 'Tbc2'},
16 }
17
18 -- Física para cálculo de tensões e deslocamentos
19 PhysicalMethod {
20   id       = 'StressPhysics',
21   typeName = 'GemaStdStressPhysics.plate',
22   type     = 'fem',
23
24   mesh               = 'mesh',
25   boundaryConditions = {'ubc'},
26   skipStress = true,     -- Tensões calc. pela física de acoplamento
27   properties = {h='t'}, -- Tradução do nome da propriedade. Física
28                         -- espera que a espessura seja denominada 'h'
29 }
30
31 -- Física para acoplamento tensão - temperatura
32 PhysicalMethod {
33   id       = 'TempStressPhysics',
34   typeName = 'GemaStdStressPhysics.plateTemperature',
35   type     = 'fem',
36
37   mesh       = 'mesh',
38   properties = {h = 't'},
39   referenceT = 100,  -- Temperatura de referência para o material
40 }
41
42 -- Script de orquestração
43 function ProcessScript()
44   -- Executa análise FEM acoplada.  As 3 físicas fornecidas como
45   -- parâmetro cooperam na construção da matriz de rigidez global
46   fem.solve({'HeatPhysics', 'StressPhysics', 'TempStressPhysics'},
47             'solver')
48
49   -- Salva as variáveis de estado e as tensões calculadas nos
50   -- pontos de Gauss no arquivo resultados.nf
51   utils.saveMeshFile('mesh', './resultados.nf', 'nf', {'ux','uy','T'},
52                      {'sxx', 'syy', 'sxy', 's1', 's2'},
53                      {saveDisplacements=true})
54 end
```

6.3 Prototipação rápida

Este teste revisita o cenário de condução de calor em uma placa em regime permanente, apresentado na seção 6.1.1. Seu objetivo é avaliar a possibilidade de implementação de físicas em Lua conforme apresentado na seção 4.4.2. Para tanto, uma física básica de condução de calor foi implementada em Lua conforme apresentado no *script* 21.

A tabela `TPhysics` define as características gerais da física, informando ao código de suporte em C++ os tipos de elementos suportados, as variáveis de estado utilizadas, bem como as propriedades e condições de contorno necessárias. Cada entrada dos campos `stateVars` e `properties` define dados que devem estar presentes no modelo para que a física possa ser executada, identificando as unidades utilizadas internamente pela física, a dimensão esperada para o dado e se o mesmo é ou não obrigatório. De forma similar, a entrada `bc` indica os tipos de condições de contorno suportadas e os atributos de interesse em cada uma delas.

Desta forma, a tabela `TPhysics` indica que esta física requer que exista no modelo uma variável de estado `T` e que os elementos da malha estejam associados a uma propriedade `t`, que indica sua espessura, e a uma propriedade `k` que informa sua condutividade. Esta última possui duas entradas opcionais na tabela, pois a mesma pode ser escalar ou vetorial (para tratar cenários anisotrópicos). A função `checkProperties()`, executada pelo módulo de suporte em C++ após a tabela anterior ter sido interpretada, é usada para garantir que uma destas opções esteja disponível.

A função `TPhysics.fillElementData` é a responsável por retornar a matriz de rigidez e o vetor de forças locais a um elemento, sendo a equivalente em Lua da função homônima presente na interface para físicas de elemento finitos em C++. Esta função pode ser escrita manualmente para maior flexibilidade, ou gerada através de uma chamada a `CreateFillElementDataFunction`, conforme apresentado no exemplo.

Esta chamada recebe como parâmetros uma tabela que indica quais propriedades descritas na tabela `TPhysics` (`k` e `t`) e quais matrizes auxiliares (`B`, representando a matriz que contém as derivadas parciais cartesianas das funções de forma do elemento) estarão disponíveis para consulta como variáveis globais pela função `fillElemData`, passada como segundo parâmetro na chamada a `CreateFillElementDataFunction`[1].

[1]Estas variáveis são globais apenas em relação à função `fillElemData`, e não ao ambiente Lua como um todo. Isto é obtido através de uma funcionalidade da linguagem Lua que permite alterar o ambiente global visto por uma função em particular.

***Script* 21** Física para cálculo de temperatura implementada em Lua.

```
-- Tabela com definições básicas para uma física em Lua
TPhysics = {
  dimension      = 2,
  coordinateUnit = 'm',
  elementTypes   = {'quad4', 'quad9', 'tri3', 'tri6'},
  stateVars  = {{id = 'T', description = 'temperature', unit = 'K'}},
  properties = {{id = 't', description = 'plane width', unit = 'm'},
                {id = 'k', description = 'conductivity',
                 unit = 'W/(m.K)', required = false},
                {id = 'k', description = 'conductivity',
                 unit = 'W/(m.K)', dim = 2, required = false}},
  bc = {{id = 'fixedTBc', type = 'fixed temperature', bcType = 'node',
         bcAccessors = {{id = 'T', acId = 'TBc',
                         description = 'node temperature', unit = 'K'},
         }}},
}

-- Verifica se a condutividade está definida (escalar ou vetorial)
function TPhysics:checkProperties()
  assert(self.accessors.k, 'Missing conductivity property')
  return true
end

-- Obtém a contribuição para a matriz de rigidez em um ponto de Gauss
local function fillElemData()
  -- Obtém a matriz de condutividade através do valor de k (caso
  -- isotrópico) ou dos valores de k(1) e k(2) (caso anisotrópico)
  local kx = (type(k) == 'number' and k) or k(1)
  local ky = (type(k) == 'number' and k) or k(2)
  local Cond = Matrix{{kx, 0}, {0, ky}}

  -- Calcula a matriz retornada (true indica que esta é simétrica)
  return B:t() * Cond * B * t, true
end

-- Função para preenchimento da matriz local de um elemento com base
-- na função auxiliar fillElemData(). Primeiro parâmetro define os
-- dados que estarão disponíveis automaticamente para a função.
TPhysics.fillElementData = CreateFillElementDataFunction(
                             {'k', 't', B = true}, fillElemData)

-- Função para retornar condições de contorno de Dirichlet
TPhysics.fixedBoundaryConditions = CreateFixedBoundaryConditionsFunction(
                                     {'fixedTBc', 'TBc', 'T'})

-- Física para cálculo de temperatura implementada em Lua
PhysicalMethod {
  id       = 'HeatPhysics',
  typeName = 'GemaStdProxyPhysics',
  type     = 'fem',
  mesh               = 'mesh',
  boundaryConditions = {'Border temperature'},
  methods            = TPhysics, -- Tabela c/ def. da implementação
}
```

A função `fillElemData`, por sua vez, tem como responsabilidade retornar uma matriz com a contribuição local de um ponto de integração para a matriz completa do elemento, sabendo que os valores de `k`, `t` e `B` disponíveis foram avaliados no próprio ponto de integração. Da mesma maneira, esta função deve retornar também sua contribuição para o vetor de forças, se houver (no exemplo não há contribuição pois não há geração de calor interna à placa).

O *script* 22 apresenta uma função equivalente à função criada pela chamada a `CreateFillElementDataFunction`, ilustrando como o código em `fillElemData` é integrado ao laço que percorre os pontos de integração do elemento (linhas 37 a 41). É interessante observar que esta função possui uma estrutura muito semelhante à implementação equivalente escrita em C++.

Voltando ao *script* 21, a função `fixedBoundaryConditions` tem como objetivo retornar uma lista de tuplas contendo os graus de liberdade a serem removidos do problema através da condição de contorno que fixa temperaturas nas bordas da placa. Utilizando a mesma estratégia anterior, esta função é criada dinamicamente através da chamada à função `CreateFixedBoundaryConditionsFunction`. Finalmente, a tabela `TPhysics` é associada com a física "HeatPhysics" através do atributo `methods` na linha 53.

Para avaliar o custo introduzido pela física em Lua, foram efetuados testes comparando seu desempenho com a física padrão para cálculos de temperatura implementada em C++. A figura 6.6(a) apresenta uma comparação entre o tempo decorrido para cálculo das matrizes locais pela função `fillElementData`, quando implementada em Lua e em C++, para discretizações da placa com quantidade crescente de triângulos[2]. Conforme esperado, o tempo gasto em Lua é da ordem de 15 vezes maior do que o tempo gasto em C++, entretanto o aumento no tempo total de simulação é de 32% para uma malha com 80.000 triângulos, decrescendo para apenas 9% para uma malha com 320.000 triângulos (figura 6.6(b)). Os tempos medidos são apresentados na tabela 6.1.

Para uma melhor compreensão da origem deste resultado, a figura 6.7 decompõe o tempo total de simulação nos tempos necessários para a carga do modelo, execução das físicas, montagem da matriz global e do vetor global de forças a partir das matrizes e vetores locais (*assembler*), aplicação das condições de contorno de Dirichlet ao problema, com consequente ajuste na matriz e no vetor globais, resolução do sistema linear de equações e outras contribuições. Estes valores estão separados em três conjuntos de testes: o primeiro utiliza a física implementada em C++ e matrizes esparsas para a solução numérica, o segundo,

[2]Os valores apresentados correspondem à média de três medições de tempo, efetuadas em um laptop com processador Core i7-4510U @ 2.00GHz com 16GB RAM e sistema operacional Windows 8.1, 64-bits.

***Script* 22** Função equivalente à gerada por CreateFillElementDataFunction().

```
1  function TPhysics:fillElementData(e, Ke, fe)
2    local shape  = e:shape()
3    local kAc    = self.accessors.k
4    local tAc    = self.accessors.t
5    local k, t
6
7    -- Preenche a matriz X com as coordenadas dos nós do elemento
8    local X = e:nodeMatrix(self.accessors.coord)
9
10   -- Preenche valor das propriedades se estas não forem funções
11   if not kAc:info():canStoreFunctions() then k = kAc:value(e) end
12   if not tAc:info():canStoreFunctions() then t = tAc:value(e) end
13
14   -- Obtém a regra de integração a ser utilizada. Utiliza um
15   -- cache local para armazenar regras por tipo de elemento
16   local ir = self.ir[e:type()]
17   if not ir then
18     ir = self.mesh:elementIntegrationRule(e:type(), 1)
19     self.ir[e:type()] = ir
20   end
21
22   -- Laço para cada ponto de Gauss
23   local symmetric = true
24   for i = 1, ir:numPoints() do
25     -- Obtém o ponto de integração e seu peso
26     local ip, w = ir:integrationPoint(i)
27
28     -- Preenche valor das propriedades se estas forem funções
29     if kAc:info():canStoreFunctions() then k = kAc:value(e, ip) end
30     if tAc:info():canStoreFunctions() then t = tAc:value(e, ip) end
31
32     -- Calcula a matriz B (com as derivadas parciais cartesianas
33     -- das funções de forma) e o determinante do Jacobiano
34     local B, detJ = shape:shapeCartesianPartials(ip, X, true)
35
36     -- Código equivalente à função fillElemData()
37     local kx   = (type(k) == 'number' and k) or k(1)
38     local ky   = (type(k) == 'number' and k) or k(2)
39     local Cond = Matrix{{kx, 0}, {0, ky}}
40
41     local KAux, sym, fAux = B:t() * Cond * B * t, true, nil
42
43     -- Integra matriz e vetor KAux/fAux em Ke/fe
44     if KAux then
45       Ke:set(Ke + w * detJ * KAux)
46       symmetric = symmetric and sym
47     end
48     if fAux then
49       fe:set(fe + w * detJ * fAux)
50     end
51   end
52
53   return true, symmetric  -- OK + flag indicando se Ke é simétrica
54 end
```

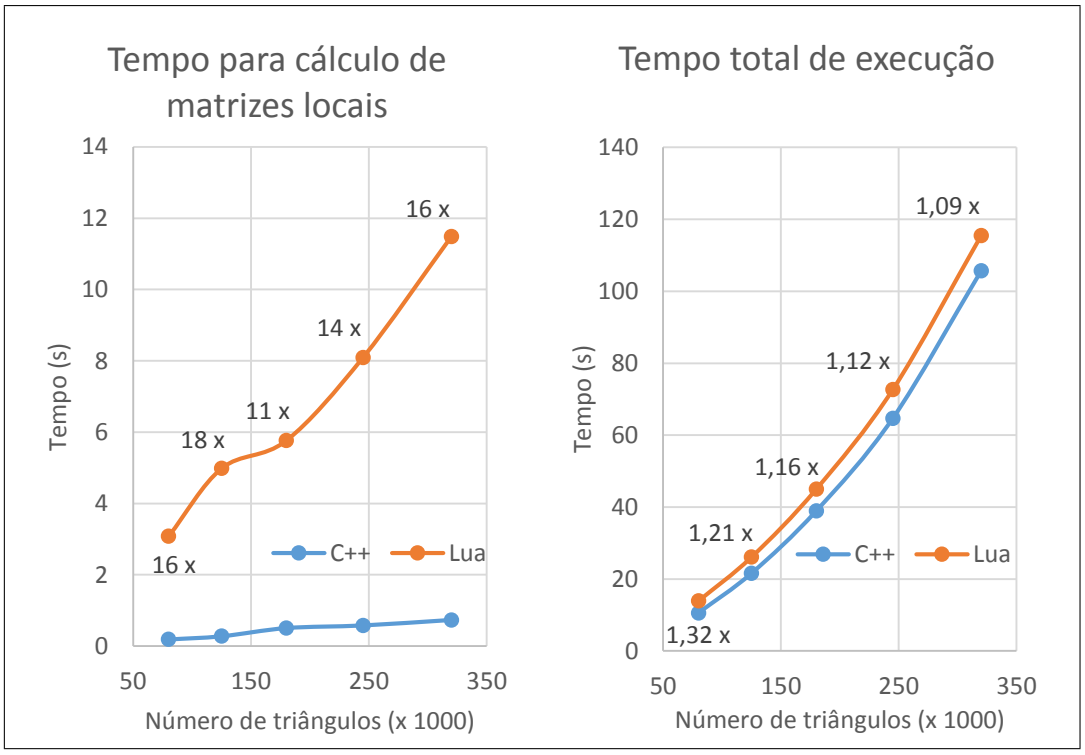


Figura 6.6: Comparação de eficiência entre físicas implementadas em C++ e Lua. Valores numéricos apresentados sobre a curva Lua representam a razão entre os tempos em Lua e C++. Valores medidos para solução com uso de matrizes esparsas.

Triângulos (x1000)	Tempo para cálculo de matrizes locais (s)		Tempo total de execução (s)	
	C++	Lua	C++	Lua
80	0,19	3,08	10,55	13,92
125	0,28	4,99	21,56	26,10
180	0,50	5,77	38,91	44,99
245	0,58	8,09	64,67	72,67
320	0,73	11,48	105,68	115,44

Tabela 6.1: Tempos medidos comparando a eficiência entre físicas implementadas em C++ e Lua.

utiliza a física em Lua e matrizes esparsas, enquanto o terceiro utiliza a física em Lua mas monta a matriz completa para a solução numérica. Este terceiro conjunto de testes só é viável para placas discretizadas com um número limitado de triângulos.

Para todos os testes que utilizam matrizes esparsas, o tempo dominante, representando em média cerca de 70% do tempo total de simulação, consiste nas operações combinadas de *assembler* e aplicação das condições de contorno. Estas duas operações são as únicas que modificam a matriz esparsa global durante a execução da simulação e são implementadas pela biblioteca Armadillo. Sua fatia

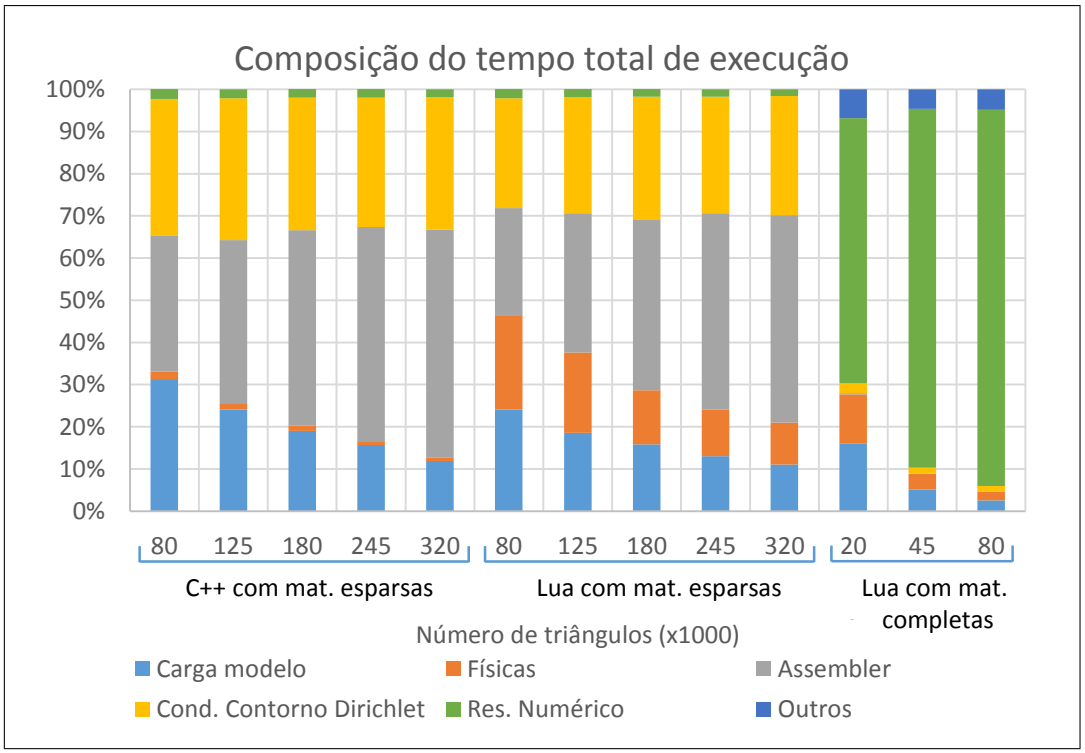


Figura 6.7: Composição do tempo total de execução por número de triângulos, tipo de física e tipo de matriz utilizada.

de tempo proporcional cresce com o aumento do número de triângulos, o que faz com que a importância global do tempo gasto pelas físicas diminua. A resolução do sistema linear *per si* leva cerca de 2% do tempo total de execução.

Ao serem analisados os testes onde são montadas matrizes completas, a composição do tempo total de execução muda radicalmente. Neste novo cenário, o tempo de execução é dominado pela solução do sistema linear de equações, o que torna todos os demais tempos praticamente irrelevantes quando o número de triângulos aumenta. Além de ser limitado quanto ao tamanho máximo do problema que pode ser resolvido, a solução com matrizes completas também é muito mais demorada que a solução com matrizes esparsas. No exemplo estudado, o caso com 80.000 triângulos é executado em cerca de 14 segundos no cenário com físicas em Lua e matrizes esparsas contra 176 segundos com o uso de matrizes completas.

Cabe salientar que o tempo para leitura do modelo pode ser otimizado através do uso de outros formatos de dados para o armazenamento da malha que não o uso de tabelas em Lua.

6.4 Simuladores externos e troca de dados entre malhas

Este teste apresenta um cenário de acoplamento hidro-mecânico onde são utilizados simuladores externos para a execução em separado das simulações hidráulica e mecânica. A simulação hidráulica é executada apenas em uma pequena parte do domínio de simulação através de uma malha triangular. Já a simulação mecânica é executada sobre uma malha de quadriláteros. O *script* de orquestração é o responsável por coordenar a execução dos dois simuladores e por executar o processo de transferência de dados entre as malhas.

O cenário simulado tem como objetivo calcular o campo de deformações geradas pela produção de hidrocarbonetos no entorno de um reservatório, utilizando um modelo axi-simétrico, centrado no poço produtor (figura 6.8).

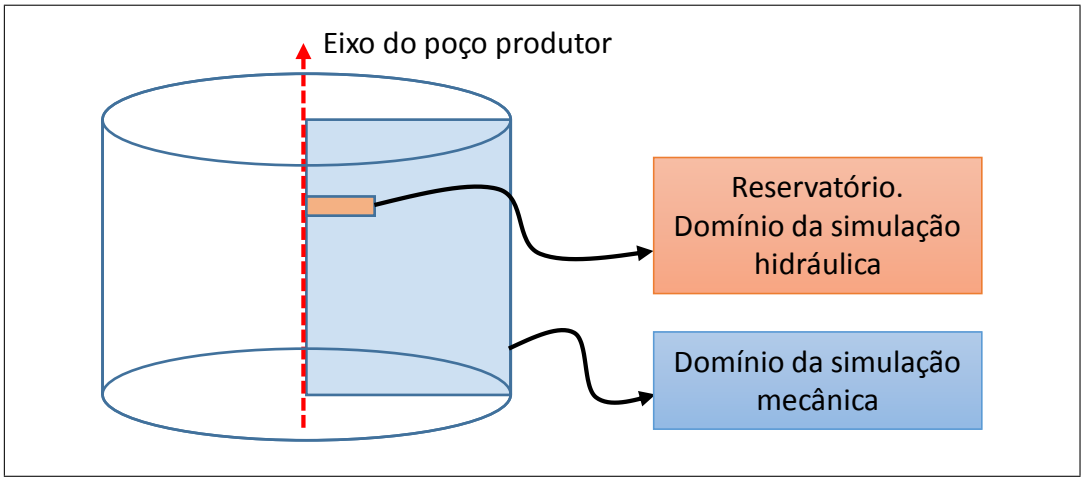


Figura 6.8: Domínio para simulação de deformações mecânicas no entorno de um reservatório.

A simulação é composta por três passos. No primeiro, são calculadas as diferenças de pressão no reservatório induzidas pela produção de hidrocarbonetos. Este cálculo é realizado pelo simulador GeoFlux3d sobre uma malha de triângulos que discretiza apenas a região do reservatório. Os valores de poro-pressões calculados são armazenados nos nós da malha e contém valores para diversos passos de tempo.

O segundo passo da simulação consiste em transferir os dados de poro-pressões calculados, nos diversos passos de tempo da simulação hidráulica, para um subconjunto da malha mecânica, cujo domínio engloba o reservatório e sua vizinhança. Esta é uma malha de quadriláteros e os dados de poro-pressões devem ser armazenados nos pontos de Gauss de seus elementos.

O terceiro passo consiste em executar novamente o simulador GeoFlux3d, agora para que este calcule as deformações nas camadas vizinhas ao reservatório ocasionadas pela diminuição da pressão no mesmo.

O *script* 23 apresenta o *script* de orquestração utilizado, seguindo os passos acima. Os processos para execução do simulador externo e para geração de seus arquivos de entrada foram escritos diretamente em Lua. Sua implementação não é apresentada por brevidade e por não trazerem nenhuma contribuição adicional para a discussão.

Quando a malha hidráulica é lida do arquivo neutro na linha 22, todos os passos de tempo gerados pela simulação são carregados e armazenados no histórico associado ao atributo `'Porep'`. O conjunto completo de dados é transferido para a malha mecânica quando o processo de mapeamento é chamado na linha 42.

É interessante notar que, a nível da orquestração, o uso do simulador externo para os cálculos hidráulicos e/ou mecânicos podem ser facilmente substituídos pelo uso de uma simulação interna ao *framework*, caso os modelos necessários sejam implementados através de novos *plugins* com as físicas apropriadas.

A figura 6.9 apresenta a malha hidráulica calculada e seu mapeamento para a malha mecânica em um instante de tempo, enquanto a figura 6.10 apresenta as deformações calculadas na direção y.

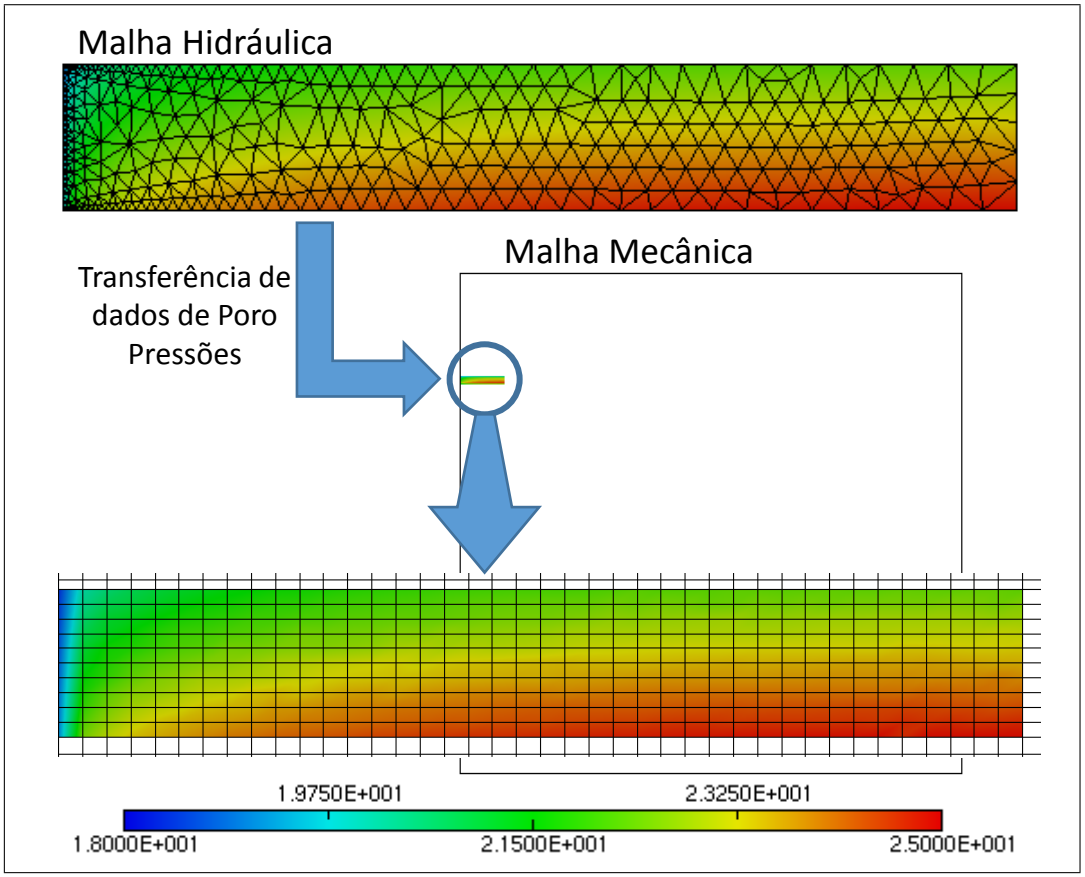


Figura 6.9: Transferência de dados entre malhas.

***Script* 23** *Script* de orquestração para cálculo de deformações no entorno de um reservatório.

```
function ProcessScript()
  -- Arquivos usados na definição das simulações
  local datFile_H     = 'input_H.dat'        -- Modelo GeoFlux3d
  local posFile_H     = 'aux_H_mesh.pos'     -- Malha (arq.neutro)
  local datFile_M     = 'output_M.dat'       -- Modelo GeoFlux3d
  local posFile_M     = 'output_M_mesh.dat' -- Malha (arq.neutro)
  local datFile_M_ref = 'input_M.dat'    -- Modelo de referência
  local posFile_M_ref = 'aux_M_mesh.dat' -- Malha de referência

  --------------- Simulação Hidráulica ---------------
  -- Executa GeoFlux3d para cálculo de poro-pressões
  runGeoFlux3d(datFile_H)

  -------------- Transferência de dados --------------
  -- Converte resultados do sim. hidráulico para arquivo neutro
  convertGid2Pos(datFile_H, posFile_H)

  -- Converte malha do simulador mecânico para arquivo neutro
  convertGid2Pos(datFile_M_ref, posFile_M_ref)

  -- Carrega a malha hidráulica e os valores nodais de poro-pressão
  utils.loadMeshFromFile('HydraulicMesh', posFile_H, 'nf', 'Porep')

  -- Carrega a malha mecânica
  utils.loadMeshFromFile('MechanicalMesh', posFile_M_ref, 'nf')

  -- Adiciona novo atributo 'PoreP' aos pontos de Gauss da malha
  -- mecânica, com propriedades iguais às lidas da malha hidráulica
  addGaussAttribute('MechanicalMesh', 'HydraulicMesh', 'Porep')

  -- Cria índice para consultas espaciais à malha hidráulica
  local bucketIndex = utils.buildBucketIndex('HydraulicMesh',
                                             'element', 10)

  -- Define a tolerância usada no mapeamento através de uma
  -- heurística baseada no tamanho das arestas da malha hidráulica
  local tol = getAdaptedNoDataEps('HydraulicMesh')

  -- Transfere dados da malha hidráulica para a malha mecânica.
  -- Pontos fora do domínio hidráulico recebem o valor 0.0
  utils.meshMapping('MechanicalMesh', true, bucketIndex, 'Porep',
                    true, 0.0, tol)

  --------------- Simulação Mecânica ---------------
  -- Cria o arquivo de dados de entrada para o simulador GeoFlux3d
  createDataFile(datFile_M_ref, datFile_M, 'MechanicalMesh', 'Porep')

  -- Executa a simulação mecânica
  runGeoFlux3d(datFile_M)

  -- Converte resultados gerados pelo sim. mecânico para arq. neutro
  convertGid2Pos(datFile_M, posFile_M)
end
```

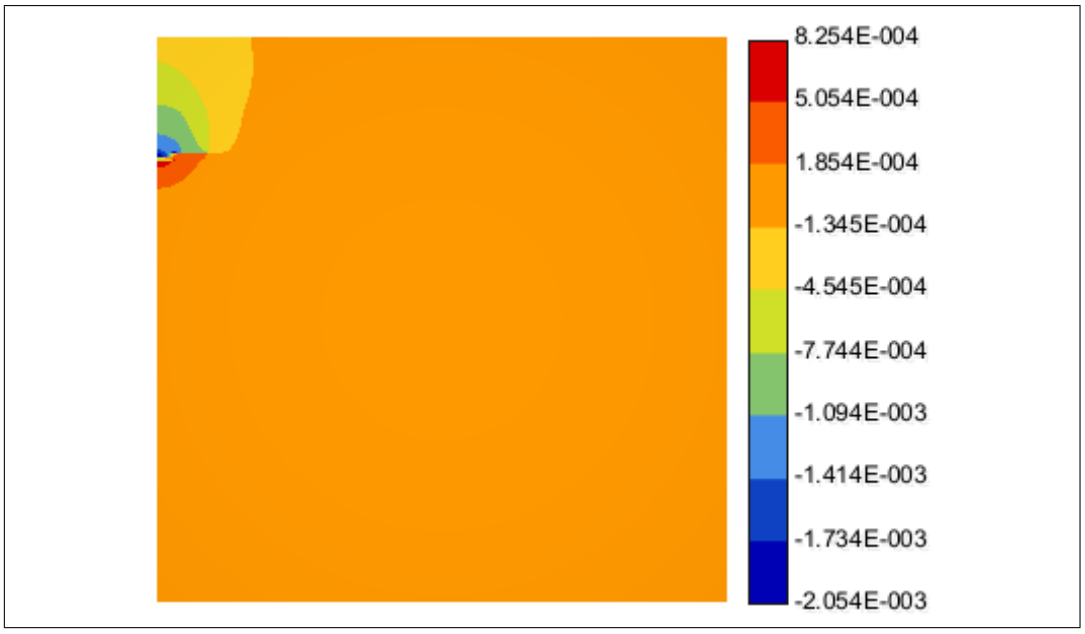


Figura 6.10: Deformações calculadas no eixo Y.

6.5 Modelagem de bacias

Este último caso de teste explora um cenário 2D de modelagem de bacias sedimentares com base nos conceitos e nos modelos matemáticos apresentados na seção 5.3. O modelo de bacia simulado não representa um caso real, tendo sido construído para ilustrar as características do simulador.

Em termos do *framework* GeMA, este caso de teste é bastante interessante por integrar diversas funcionalidades, apresentadas isoladamente nos exemplos anteriores, em uma única simulação. Resumidamente, este teste engloba a utilização acoplada de físicas implementadas em C++ com base no método de elementos finitos (temperatura), com físicas implementadas em Lua e baseadas em modelos analíticos (descompactação, maturação e geração). A malha de simulação é dinâmica e heterogênea, evoluindo no tempo para acompanhar o processo de deposição de camadas e para o tratamento de intrusões ígneas, cuja presença leva à necessidade de utilização de passos adaptativos de tempo na simulação. O modelo térmico é não-linear, uma vez que a condutividade térmica e o calor específico são funções da temperatura e da porosidade. O cálculo de geração de hidrocarbonetos pode ser ainda acoplado ao cálculo de temperaturas através de uma dependência adicional da fração de óleo e gás convertidas por parte da condutividade térmica. As condições de contorno aplicadas ao topo e à base da bacia são dinâmicas e variam com o tempo.

O modelo da bacia é fornecido ao simulador através de uma malha contendo a geometria atual das camadas e de uma tabela que fornece a idade de início e término da deposição de cada camada. Se a camada for uma rocha geradora, devem

ser especificados ainda a cinética associada e seu teor de carbono orgânico total (COT). Ambos os dados são utilizados no cálculo de geração de hidrocarbonetos. Para camadas intrusivas, assume-se que o evento acontece instantaneamente em um momento de tempo especificado. Um segundo parâmetro fornece a temperatura da intrusão.

Para facilitar o cálculo de compactação e a evolução da malha de simulação no tempo, algumas restrições foram impostas ao formato da malha. Esta deve ser uma malha estruturada, composta por "linhas" de quadriláteros e/ou triângulos, verticalmente alinhados. Cada camada deve ser composta por uma única linha da malha. Se houver necessidade de mudança de material no interior de uma camada (mudança de faces), esta deve ocorrer ao longo da diagonal de um triângulo, de forma que qualquer aresta vertical da malha possua sempre o mesmo material em ambos os seus lados, conforme proposto por Wangen (1991). Isto permite que o cálculo de compactação seja feito ao longo das linhas verticais da malha sem considerações sobre os parâmetros a serem utilizados. A figura 6.11, em conjunto com a tabela 6.2, apresenta a malha para a simulação do exemplo, juntamente com a idade e demais parâmetros de cada camada.

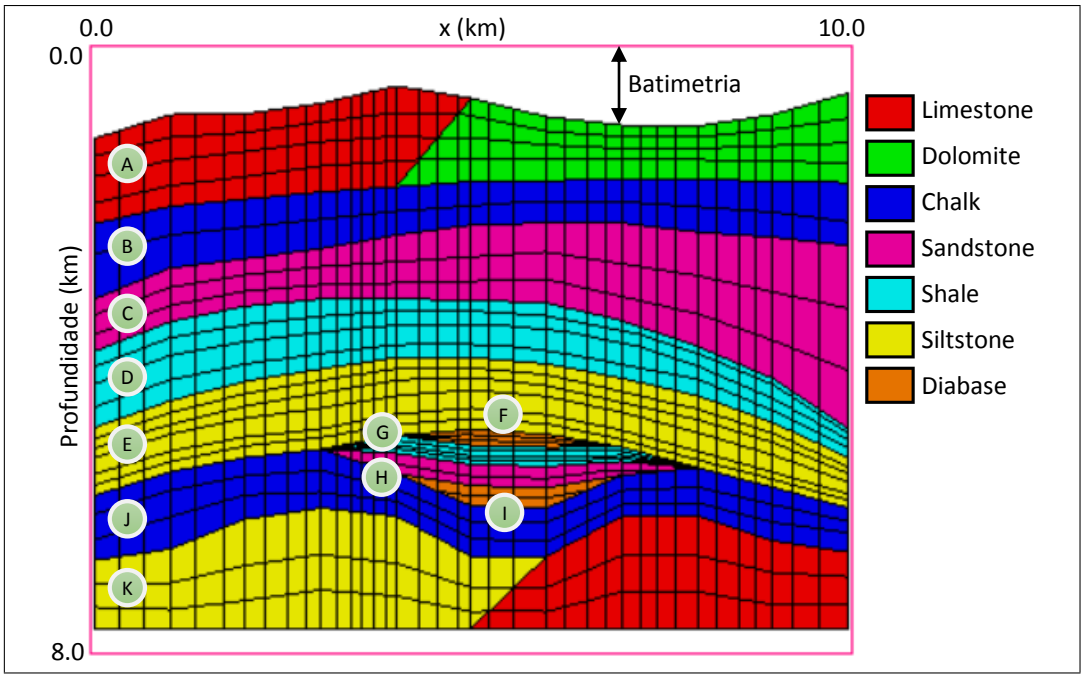


Figura 6.11: Modelo da bacia simulada. Marcações A a K relacionam camadas com entradas da tabela 6.2.

O *script* 24 apresenta as propriedades do material associadas com cada uma das camadas. É interessante notar que as dependências dos parâmetros térmicos da porosidade e temperatura, $Q(\phi)$, $\lambda(\phi, T, F_o, F_g)$, $c_p(\phi, T)$ e $\rho(\phi)$, conforme descrito na seção 5.3, são implementadas através de funções do usuário. Estas propriedades são marcadas como esparsas e, como todos os elementos compartilham a mesma

Grupo	Camada	Idade deposição (Myr)		Outros
		Início	Término	
A	1	5.0	0.0	
	2	10.0	5.0	
	3	15.0	10.0	
	4	20.0	15.0	
B	5	35.0	20.0	
	6	50.0	35.0	
C	7	57.0	50.0	
	8	63.0	57.0	
	9	70.0	63.0	
D	10	72.0	70.0	Geradora. Cinética tipo IIS, (Behar *et al*, 1997). COT = 5.0%
	11	74.0	72.0	
	12	76.0	74.0	
	13	78.0	76.0	
E	14	80.0	78.0	
	15	82.0	80.0	
	16	84.0	82.0	
	17	86.0	84.0	
	18	88.0	86.0	
	19	90.0	88.0	
F	20	66.0	66.0	Intrusão. Temperatura = $1000\ ^{\circ}C$
	21	66.0	66.0	
	22	66.0	66.0	
G	23	100.5	100.0	Geradora. Cinética tipo IIS, (Behar *et al*, 1997). COT = 5.0%
	24	101.0	100.5	
	25	101.5	101.0	
	26	102.0	101.5	
H	27	103.0	102.0	
	28	104.0	103.0	
I	29	60.0	60.0	Intrusão. Temperatura = $1100\ ^{\circ}C$
	30	60.0	60.0	
J	31	109.0	104.0	
	32	114.0	109.0	
	33	120.0	114.0	
K	34	130.0	120.0	
	35	140.0	130.0	
	36	150.0	140.0	

Tabela 6.2: Parâmetros da bacia simulada.

***Script* 24** Propriedades associadas com as camadas do modelo.

```
-- Conjunto de propriedades associadas aos elementos do modelo
PropertySet {
  id          = 'LithoPSet',
  typeName    = 'GemaStdPropertySet',
  description = 'Lithological parameters for facies',
  properties  = {
    {id = 'phic',  description = 'Compaction coef. for Athy law', unit='1/km'},
    {id = 'phi0',  description = 'Initial porosity for Athy law', unit='%'},
    {id = 'g_rho', description = 'Grain density', unit = 'kg/m3'},
    {id = 'g_k',   description = 'Grain conductivity @20degC', unit='W/(m.K)'},
    {id = 'g_cp',  description = 'Grain spec. heat @20degC', unit='J/(kg.K)'},
    {id = 'g_Q',   description = 'Grain radiogenic heat', unit='uW/m3'},
    {id = 't',     description = 'Element thickness', unit='m',
                   defVal = 1.0,         sparse = true},
    {id = 'rho',   description = 'Mixed fluid grain density', unit='kg/m3',
                   defVal = 'rho_phi',   sparse = true, functions = true},
    {id = 'k',     description = 'Mixed fluid grain conduct.', unit='W/(m.K)',
                   defVal = 'k_phi_T',   sparse = true, functions = true},
    {id = 'cp',    description = 'Mixed fluid grain spec. heat', unit='J/(kg.K)',
                   defVal = 'cp_phi_T', sparse = true, functions = true},
    {id = 'Q',     description = 'Mixed fluid grain radiog. heat', unit='uW/m3',
                   defVal = 'Q_phi',     sparse = true, functions = true},
    {id = 'L',     description = 'Latent heat of solidification',
                   unit = 'J/kg',        sparse = true},
    {id = 'l_cp',  description = 'Specific heat for the liquid phase',
                   unit = 'J/(kg.K)',    sparse = true},
    {id = 'f_cp',  description = 'Spec. heat for the freezing interval',
                   unit = 'J/(kg.K)',    sparse = true},
    {id = 'l_rho', description = 'Density for the liquid phase',
                   unit = 'kg/m3',       sparse = true},
    {id = 'Ts',    description = 'Solid temperature',
                   unit = 'degC',        sparse = true},
    {id = 'Tl',    description = 'Liquid temperature',
                   unit = 'degC',        sparse = true},
  },
  values = {
    {id = 'limestone', phic = 0.52, phi0 = 51.0, g_rho = 2680, g_k = 2.00,
                       g_cp = 845,  g_Q  = 1.40},
    {id = 'dolomite',  phic = 0.39, phi0 = 35.0, g_rho = 2790, g_k = 4.20,
                       g_cp = 860,  g_Q  = 0.29},
    {id = 'chalk',     phic = 0.90, phi0 = 70.0, g_rho = 2680, g_k = 2.90,
                       g_cp = 850,  g_Q  = 0.60},
    {id = 'sandstone', phic = 0.31, phi0 = 41.0, g_rho = 2720, g_k = 3.95,
                       g_cp = 855,  g_Q  = 0.70},
    {id = 'shale',     phic = 0.83, phi0 = 70.0, g_rho = 2700, g_k = 1.64,
                       g_cp = 860,  g_Q  = 2.03},
    {id = 'siltstone', phic = 0.51, phi0 = 55.0, g_rho = 2710, g_k = 2.01,
                       g_cp = 940,  g_Q  = 0.96},
    {id = 'diabase',   phic = 0.001, phi0 = 0.0, rho = 'rho_eff',
                       g_rho = 2800, l_rho = 1000, k = 2.60, cp = 'cp_eff',
                       g_cp = 800, l_cp = 2900, f_cp = 2900, L = 2.54e5,
                       Ts = 950, Tl = 1000, Q = 0.18},
  }
}
```

função, seu gasto de memória é mínimo e independente do número de elementos na malha.

As condições de contorno que definem a temperatura na superfície da bacia e o fluxo térmico em sua base são funções do tempo e da posição ao longo da seção geológica, $T_s(x,t)$ e $q(x,t)$. Da mesma forma, a batimetria também varia com o tempo e a posição. Estas informações são fornecidas na forma de tabelas em Lua onde cada linha está associada com uma idade e com uma lista de pares contendo uma posição e um valor (temperatura, fluxo térmico ou batimetria). Os valores das condições de contorno vistos pelo modelo de elementos finitos são implementados através de funções do usuário que consultam estas tabelas e efetuam interpolações lineares para obter valores nas posições e tempos desejados.

Para acompanhar a evolução da deposição e compactação das camadas, a simulação utiliza uma segunda malha, inicialmente vazia, que vai sendo montada e atualizada na medida em que as camadas são depositadas. A cada passo de tempo, o simulador calcula a fração da camada a ser depositada, garantindo uma taxa de deposição de grãos constante por camada. Para calcular o tamanho da camada depositada, esta fração é descompactada com base na espessura presente. A seguir, as camadas inferiores são compactadas devido aos novos sedimentos e as profundidades dos nós da malha são ajustadas de acordo.

Se o passo de tempo envolver o início da deposição de uma nova camada, novos elementos são incluídos na malha de acompanhamento, tendo sua temperatura inicial ajustada de acordo com a temperatura da superfície. Caso a nova camada esteja no presente, total ou parcialmente, sobre uma intrusão que ainda não ocorreu na história da simulação, a camada é "costurada" na camada abaixo da intrusão. Intrusões são sempre incluídas por inteiro em um passo de tempo, sendo inseridas entre duas camadas, forçando uma reorganização dos elementos na camada superior (figura 6.12).

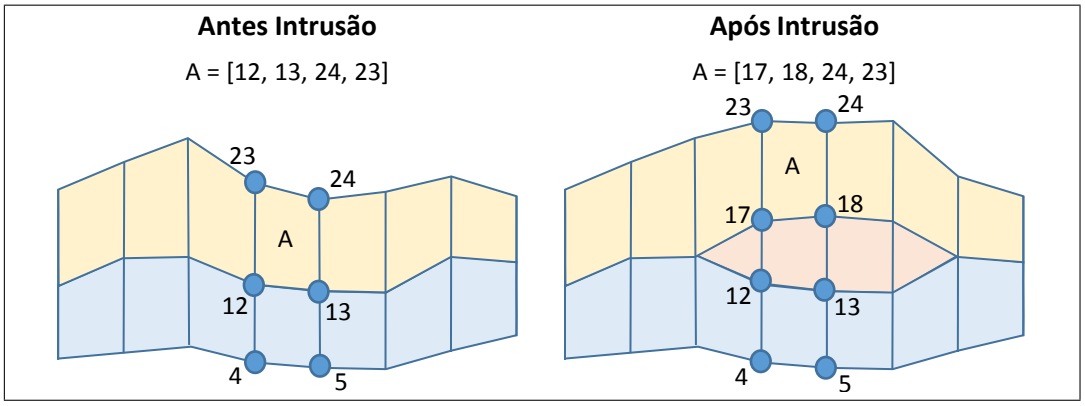


Figura 6.12: Ajustes na geometria da malha devido à intrusões.

Para os cálculos cinéticos de reflectância da vitrinita (R_o) e fração convertida de matéria orgânica em hidrocarbonetos (F) nas camadas geradoras, é necessária

informação sobre a variação de temperatura ocorrida no passo de tempo, obtida utilizando-se um histórico com dois estados associados à variável nodal que armazena dados de temperatura. Os valores de R_o e F podem ser calculados no centroide dos elementos ou em seus pontos de Gauss, e são armazenados em atributos criados pela simulação.

Os *scripts* 25 a 27 apresentam o *script* de orquestração. É interessante notar o tratamento dado na definição do passo de tempo utilizado em cada iteração da simulação. A simulação de testes utiliza um passo de tempo padrão de um milhão de anos. Este passo, porém, pode ser reduzido se o término da deposição da camada atual ocorrer antes do final do passo de tempo, fazendo com que este seja ajustado para coincidir com o final da deposição da camada. Além disso, após uma intrusão são aplicados passos refinados para capturar o aquecimento e a dissipação do calor no entorno da intrusão em passos de tempo de 100, 200, 500, 1.000, 2.000, 5.000, 10.000, 20.000, 50.000, 100.000, 200.000 e 500.000 anos após a intrusão.

***Script* 25** *Script* de orquestração para modelagem de bacias (1/3).

```
1  function ProcessScript()
2    -- Prepara o contexto de simulação, inicializando o processo de
3    -- elementos finitos e os processos de compactação e cinética.
4    local c = initSimulationContext()
5
6    -- Tempo será expresso em Milhões de anos
7    setCurrentTimeUnit('Myr')
8
9    local t  = 0.0       -- Tempo global de simulação em Myr
10   local dt = Model.dt  -- Passo de tempo básico em Myr
11   local edt            -- Passo de tempo efetivo após deposição
12   local it = 1
13   local intrusion_it = nil
14   local addedLayer, intrusions, intrudedLayers
15   local updateDof = false
16   local tol = Model.nonLinearTol -- Tolerância para loop não linear
17
18   local secsPerMYr = 60*60*24*365.25*1e6
19
20   -- Loop de simulação
21   while true do
22     -- Evolui a malha com passo de tempo sugerido 'dt'. Retorna o
23     -- passo de tempo efetivamente usado, se uma nova camada foi
24     -- inserida e se há intrusões pendentes a serem tratadas
25     -- ainda neste passo de tempo
26     edt, t, addedLayer, intrusions = c.compaction:addTimeStep(it, dt)
27     if not edt then
28       break -- Simulação finalizada. Chegamos ao presente
29     end
30
31     -- Atualiza o tempo corrente de simulação
32     setCurrentTime(t)
```

***Script* 26** *Script* de orquestração para modelagem de bacias (2/3).

```
34      -- Calcula a porosidade das células
35      c.compaction:updateCellPorosity()
36
37      -- Se foram adicionados novos elementos à malha precisamos
38      -- atualizar as bordas das condições de contorno e as
39      -- condições iniciais.
40      if addedLayer then
41        updateMeshAfterNewLayer(c, addedLayer)
42        updateDof = true -- Avisa ao processo que a malha foi alterada
43      end
44
45      -- Calcula temperatura e cinética para o passo de tempo corrente
46      local i = 1
47      local r = 1
48      while i <= Model.maxNonLinearIter do
49        -- Calcula a temperatura
50        fem.transientLinearStep(c.heat, edt * secsPerMYr, i, updateDof)
51        updateDof = false -- Necessário apenas na primeira iteração
52
53        -- Calcula Ro + fração convertida
54        c.kinetic:calcRoAndConvFraction(edt, i)
55
56        -- Convergiu?
57        r = fem.transientLinearResidual(c.heat, edt * secsPerMYr)
58        if r <= tol then
59          break
60        end
61
62        i = i + 1
63      end -- Loop não linear
64      assert(i <= Model.maxNonLinearIter, 'No convergence achieved')
65
66      -- Salva resultados em formato medit (nf não suporta evol. da malha)
67      utils.saveMeshFile('evolvingMesh', './resultados_T.'..it,
68                         'medit', {'T'})
69      utils.saveMeshFile('evolvingMesh', './resultados_phi.'..it,
70                         'medit', nil, {'phim'})
71      utils.saveMeshFile('evolvingMesh', './resultados_ro.'..it,
72                         'medit', nil, {'Ro'})
73      utils.saveMeshFile('evolvingMesh', './resultados_F.'..it,
74                         'medit', nil, {'F'})
75
76      -- Copia temperatura corrente para o histórico
77      c.em:saveNodeValueState('T', 'copy')
78
79      -- Há intrusões pendentes? Se sim, adiciona a camada intrusiva à
80      -- malha e ajusta temperaturas iniciais
81      if intrusions then
82        intrudedLayers = c.compaction:addPendingIntrusions(it)
83        c.compaction:updateCellPorosity()
84        updateMeshAfterIntrusion(c, intrudedLayers)
85        updateDof = true
86      end
```

***Script* 27** *Script* de orquestração para modelagem de bacias (3/3).

```
88      -- Prepara próxima iteração, ajustando passo de tempo e tolerância.
89      if intrudedLayers then
90        -- Vamos iniciar passos de tempo após intrusão
91        intrudedLayers = nil
92        intrusion_it   = 1
93        dt  = Model.dtAfterIntrusion[1] / 1000  -- kyr to Myr
94        tol = Model.intrusionTol or tol -- Ajusta tol após intrusão
95      elseif intrusion_it then
96        intrusion_it = intrusion_it + 1
97        if intrusion_it > #Model.dtAfterIntrusion then
98          -- Tratamento especial para intrusões finalizado
99          intrusion_it = nil
100         dt           = Model.dt
101         tol          = Model.nonLinearTol
102       else
103         -- Tratamento de intrusões continua. Ajusta dt de acordo com a
104         -- definição dos passos adaptativos definidos em dtAfterIntrusion
105         dt = (Model.dtAfterIntrusion[intrusion_it] -
106               Model.dtAfterIntrusion[intrusion_it-1]) / 1000
107       end
108     end
109
110     it = it + 1
111   end -- Loop principal
112 end
```

A figura 6.13 apresenta os valores de porosidade, temperatura, reflectância da vitrinita e fração convertida calculados em instantes de tempo selecionados. Para facilitar a interpretação, as condições de contorno foram mantidas constantes no tempo e no espaço com temperatura na superfície igual a $10\ ^{\circ}C$ e fluxo térmico na base de $45\ mW/m^2$.

É interessante notar a influência das intrusões nos resultados calculados de R_o e fração convertida. A evolução térmica nos instantes de tempo que se sucedem a uma intrusão pode ser observada na figura 6.14. É interessante notar como seu efeito é localizado e dissipa-se relativamente rápido. Em 100.000 anos o efeito na temperatura da bacia já é pouco relevante. Seus efeitos na maturação e geração de hidrocarbonetos de rochas próximas são, no entanto, permanentes.

A figura 6.15 apresenta o efeito observado do acoplamento entre os cálculos de geração e temperatura através da deformação das isotermas. Isto ocorre pois os hidrocarbonetos possuem condutividade térmica bem menor do que a água. O efeito é sutil e relevante apenas em grandes acumulações de óleo e gás. Para possibilitar sua observação na figura 6.15, o teor de COT da camada foi alterado para 50%.

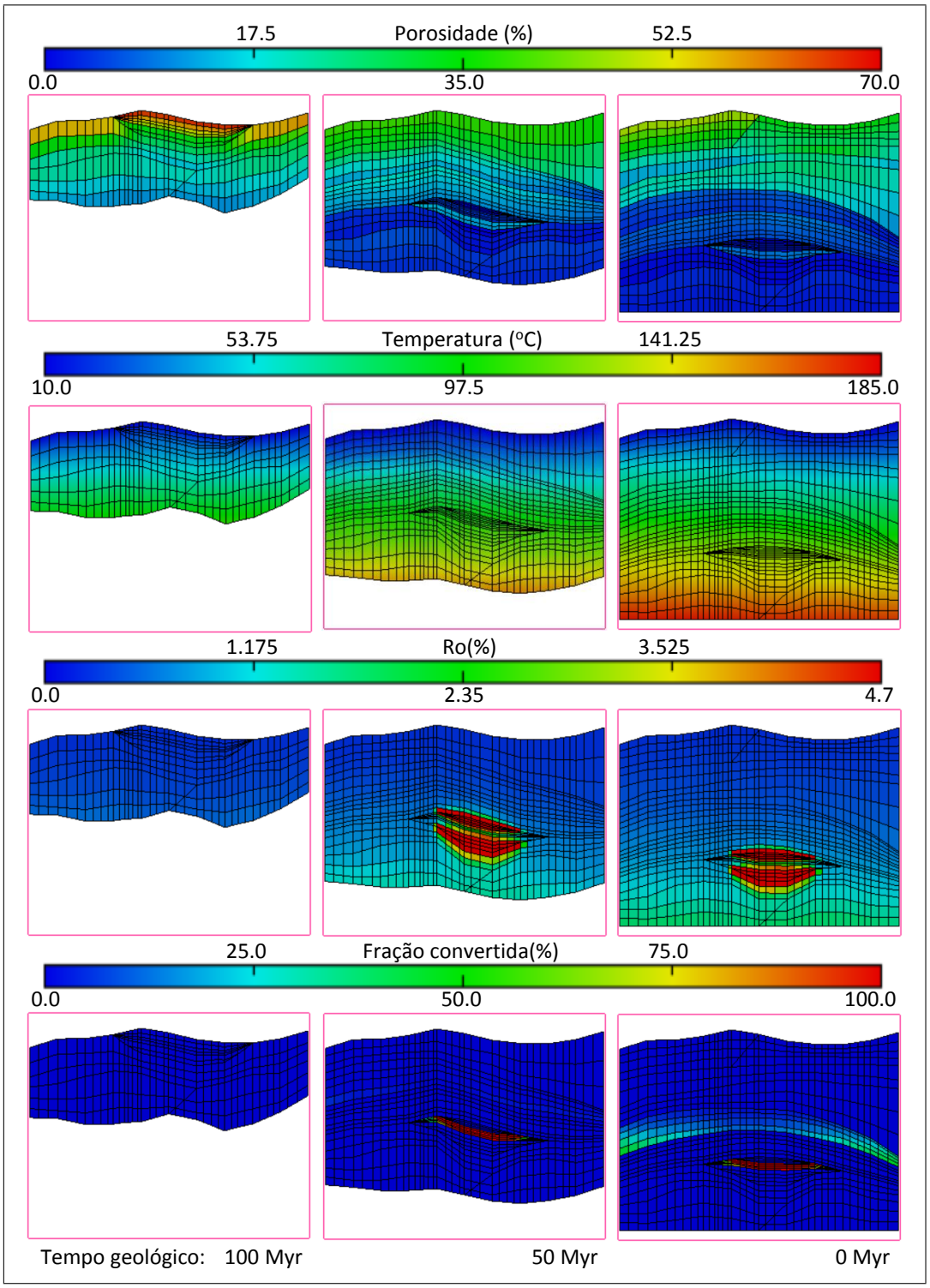


Figura 6.13: Resultados obtidos na análise da bacia simulada em instantes de tempo selecionados.

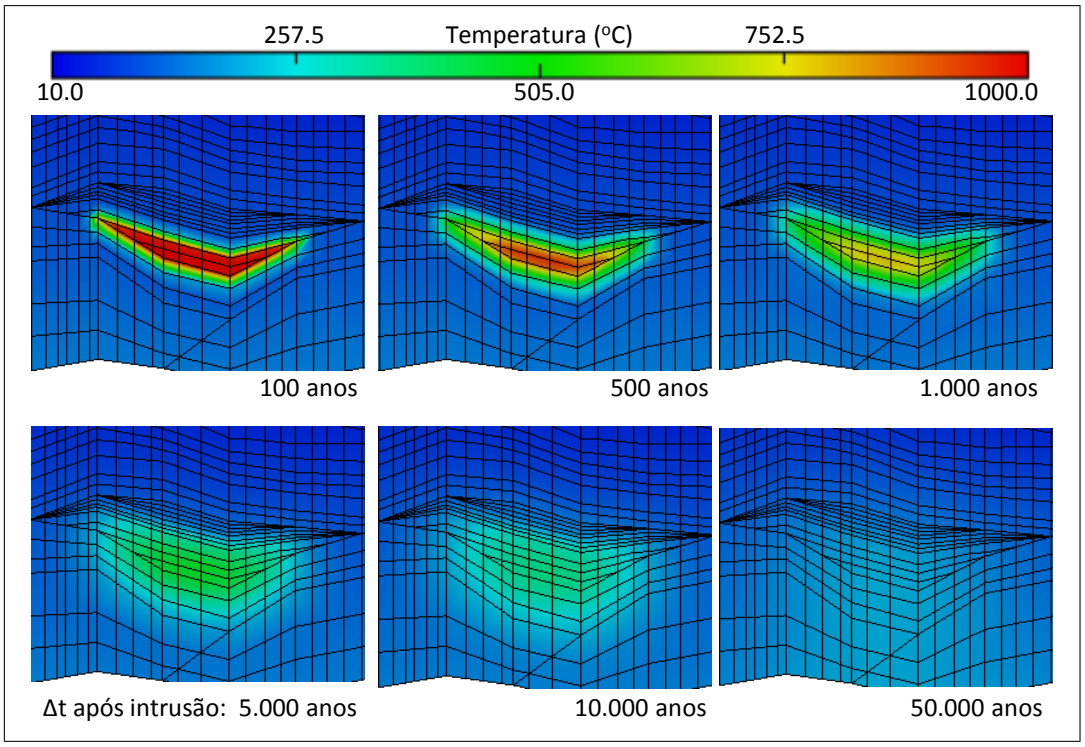


Figura 6.14: Evolução da temperatura em instantes subsequentes à uma intrusão magmática.

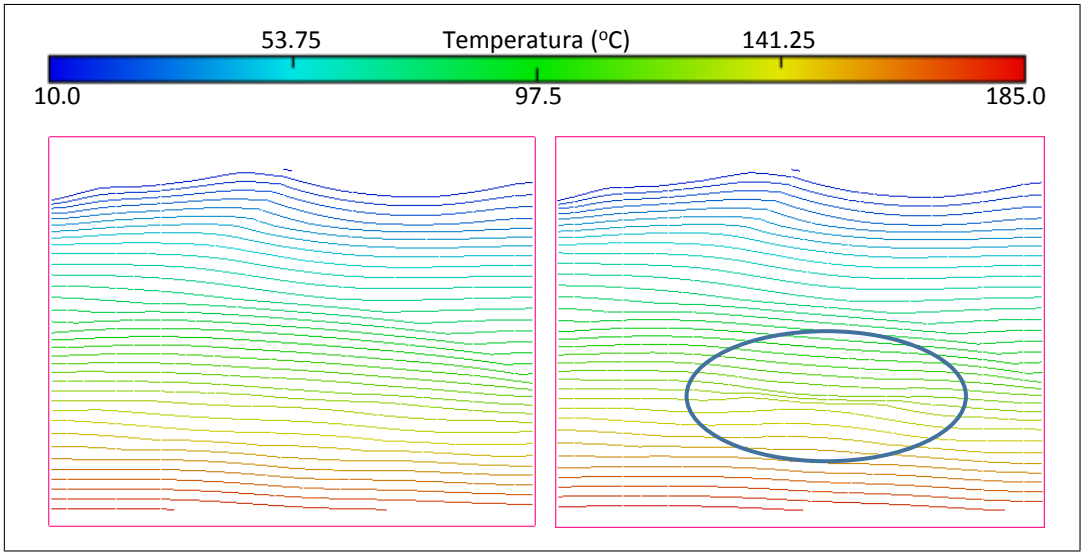


Figura 6.15: Efeito da geração de hidrocarbonetos na temperatura. Em destaque a alteração das isotermas.

6.6 Testes de regressão

O desenvolvimento e maturação de uma biblioteca de multifísica ocorre ao longo do tempo, evoluindo através de diversas versões contendo melhorias incrementais sobre um conjunto básico de funcionalidades. Independentemente de quaisquer outras características, para que a mesma seja adotada em um ambiente de produção e não apenas acadêmico, é fundamental que seus usuários tenham confiança em sua estabilidade e na corretude dos resultados gerados ao longo das versões.

Para que isso seja possível, além dos testes de validação da biblioteca, é necessária a adoção de um processo de testes de regressão que auxilie a garantir a qualidade das versões geradas ao longo do tempo. Isto é particularmente importante pois em projetos de longa duração geralmente há necessidade de evolução e manutenção de código pré-existente por equipes que não participaram do desenvolvimento original do código, o que aumenta a possibilidade de introdução de erros.

Segundo Myers (2004), testes de regressão consistem na re-execução de testes previamente executados para garantir que novas versões não introduzam erros em partes já testadas do sistema. Para que estes testes possam ser executados frequentemente, com baixo custo, é necessário que os mesmos sejam automatizados. Uma boa prática consiste em automatizar os testes de validação efetuados para que estes possam ser re-executados no futuro.

Há dois tipos básicos de testes de regressão: testes unitários e testes funcionais. Os primeiros têm como objetivo validar conjuntos individuais de rotinas e métodos enquanto testes funcionais são usados para validar o sistema como um todo em face a um conjunto pré-determinado de entradas.

Na implementação atual do *framework*, as principais classes da biblioteca base possuem testes unitários para garantir sua corretude, com especial ênfase nos testes das estruturas de dados utilizadas para associar valores com entidades da malha, classes de suporte a funções de forma e regras de integração. Estes testes incluem tanto situações baseadas em testes determinísticos quanto testes baseados em conjuntos randômicos de dados. Alguns *plugins* provendo processos básicos também implementam testes unitários, com destaque para os processos de transferência de dados entre malhas e composição da matriz de rigidez em análises por elementos finitos.

Testes funcionais da biblioteca têm como base a execução de simulações completas que utilizam assertivas e primitivas de "log" no *script* de orquestração para validação dos resultados obtidos. Após a execução, o "log" final obtido como resultado é comparado com um gabarito previamente preparado. O teste

é considerado como bem sucedido se não houver diferenças entre ambos. O conjunto atual de testes conta tanto com modelos completos para testes das físicas implementadas quanto testes sintéticos para validação da sintaxe usada para descrição dos modelos de simulação, das interfaces com o ambiente Lua e de funções do usuário.

7
Conclusões e trabalhos futuros

Em linha com os requisitos básicos enunciados no capítulo 1, o *framework* GeMA apresenta um ambiente naturalmente extensível. O uso de uma arquitetura baseada em interfaces abstratas e *plugins* permite que novas funcionalidades sejam facilmente adicionadas por terceiros sem que o código existente precise ser modificado ou mesmo recompilado. Além disso, promove uma separação de conceitos e o desacoplamento entre os módulos do sistema, facilitando sua evolução e manutenção. No capítulo 4 foram apresentados os principais mecanismos de extensibilidade utilizados.

Ao contrário dos *frameworks* apresentados no capítulo 3 em geral, e do sistema Elmer em particular, que também se utiliza de *plugins* para a implementação de novas físicas, no ambiente GeMA o mecanismo de extensão permite não só a adição de novas físicas como também a adição de novos métodos de discretização, novos métodos para solução dos sistemas de equações gerados e até mesmo novas formas para o armazenamento dos dados associados ao modelo.

Apesar de seu foco atual na simulação através do método de elementos finitos, todo o ambiente do *framework* GeMA está preparado para inclusão de novos tipos de processos que implementem o suporte para outros métodos de discretização. Em particular, no exemplo de modelagem de bacias apresentado no capítulo 6, novos processos baseados em modelos analíticos foram implementados e integrados com o método de elementos finitos para composição da simulação.

O uso de um *script* de orquestração baseado em Lua como componente central das simulações é, em grande parte, o responsável pela flexibilidade apresentada pelo *framework* na composição de simulações multifísicas, permitindo que o usuário defina facilmente o modo de acoplamento e as ações a serem tomadas em cada passo de tempo. Neste sentido, seu objetivo é bastante similar ao orquestrador utilizado pelo *framework* Rocstar, no entanto a flexibilidade obtida através do uso de um *script*, ao invés de uma API em C++ como no *framework* Rocstar, facilita a sua alteração e utilização por um grupo diverso de usuários. Além de ilustrar situações forte e fracamente acopladas, os exemplos no capítulo 6 apresentam ainda situações onde o *script* de orquestração permite a solução de problemas não-lineares.

Por ser uma linguagem de extensão de aplicações completa e eficiente, a adoção de Lua como base para a orquestração possibilita a criação de processos e físicas diretamente nesta linguagem. Isto permite que os diversos níveis de laços que compõem uma simulação sejam implementados tanto em Lua como em C++, de acordo com a conveniência para o problema em estudo.

A implementação de físicas em Lua apresentada na seção 6.3 mostrou-se uma alternativa bastante interessante para a prototipação rápida de novas físicas para o método de elementos finitos, permitindo que estas sejam implementadas essencialmente através de uma simples expressão que representa diretamente a forma fraca da equação, integrada em cada elemento da malha. Embora através de estratégias de implementação bastante distintas, o efeito alcançado é similar ao obtido pelo *framework* MOOSE.

Os testes efetuados com cálculos de temperatura através de física implementada em Lua mostram a viabilidade desta opção, com aumento no tempo total de simulação variando de 9 a 32% para os casos estudados. Neste cenário, uma análise dos tempos medidos mostrou que a principal componente para o tempo total de simulação são as operações de manipulação e construção da matriz esparsa global representando o sistema de equações a ser resolvido. Naturalmente, eventuais otimizações destas operações irão modificar o balanço global de tempos e consequentemente a razão entre os tempos totais em Lua e em C++. No entanto, independentemente de alterações nesta razão, a opção de implementação de novos modelos através de físicas em Lua é atrativa, pois permite o teste e a validação dos modelos de forma mais rápida, mesmo que, por eficiência, seja necessário migrar a implementação para um *plugin* em C++ a posteriori.

A utilização da linguagem Lua para a descrição dos dados do modelo facilita que sua leitura seja feita de maneira genérica pelo *framework*, enquanto a interpretação de sua semântica seja feita em parte por este e em parte pelos *plugins* que o estendem, de maneira similar ao encontrado no sistema Elmer. No entanto, por ser uma linguagem completa, Lua permite que as descrições contenham declarações dinâmicas, responsáveis por criar parte dos dados durante sua interpretação. Esta organização favorece ainda a possibilidade de recuperação de metadados sobre a simulação diretamente do modelo e seu uso, por exemplo, na construção de interfaces gráficas genéricas para sua edição.

A separação clara entre os dados do modelo, o método de solução e a construção de novos processos e físicas possibilita o uso do ambiente GeMA por usuários de perfis variados, exercendo papéis distintos na adequação de simulações existentes a novos conjuntos de dados e/ou na criação de novas simulações, com ou sem a necessidade de criação de novas físicas. Durante a execução do projeto no qual o desenvolvimento do *framework* GeMA está inserido, estes papeis vêm sendo

explorados com sucesso pela equipe de projeto, tanto na construção de novas físicas quanto na criação de simulações acoplando simuladores externos.

A integração de simuladores externos ao *framework* pode ser feita através de troca de arquivos conforme ilustrado na seção 6.4. Embora não muito eficiente, esta opção pode ser a única disponível para automatizar cenários envolvendo simuladores comerciais externos. Através do *script* de orquestração, processos para transferência eficiente de dados entre malhas foram executados permitindo a integração de simuladores operando sobre malhas distintas, em escalas diferentes.

Havendo acesso ao código fonte ou a uma API de programação para o simulador externo, este pode ser integrado ao *framework* GeMA de maneira mais eficiente como um componente que publica seus próprios processos. Para isso, além da criação de processos para publicar suas funcionalidades, é necessário estabelecer uma forma de troca de dados com o mesmo, o que pode ser alcançado através de classes que implementam as interfaces abstratas propostas para o tratamento de malhas, acessando os dados internos do componente.

Esta estratégia tem efeito semelhante ao método utilizado pelo *framework* Rocstar para integração com simuladores externos. Embora a linguagem de descrição de dados utilizada por este possua a vantagem de facilitar a integração com simuladores escritos em outras linguagens, tais como Fortran, esta possibilidade também existe para classes escritas no *framework* GeMA, ainda que mais trabalhosa, uma vez que a classe de interface com as malhas pode efetuar as traduções necessárias para o acesso a dados mantidos pelo simulador externo em Fortran.

7.1 Contribuições

A utilização de interfaces abstratas e *plugins* como ferramentas para permitir a extensibilidade de um sistema não é um conceito novo, tanto em termos de engenharia de *software* quanto no desenvolvimento de *frameworks*. O conceito de orquestração configurável também não é novo no âmbito de *frameworks* multifísicos. Entretanto, a arquitetura utilizada, tendo como núcleo o uso combinado de interfaces abstratas com *plugins* e a adoção de uma linguagem de extensão como componente central da arquitetura, permeando todas as etapas de definição e execução das simulações, confere à mesma flexibilidade e extensibilidade inovadoras, cujos principais benefícios foram apresentados e testados ao longo deste trabalho.

Através do uso desta arquitetura, o *framework* GeMA apresenta algumas características que o distinguem dos *frameworks* apresentados no capítulo 3 e, até onde se tem conhecimento, dos demais *frameworks* existentes. São elas:

– Suporte a múltiplos métodos de discretização do modelo matemático através da possibilidade de adição de novos processos ao ambiente GeMA.

– Orquestração configurável através do uso da linguagem de extensão Lua, conferindo flexibilidade na definição dos passos necessários para a execução de uma simulação. Por esta descrição ser feita através de uma linguagem interpretada de fácil aprendizado, a construção de novas simulações multifísicas com base em modelos matemáticos já suportados pelo *framework* pode ser feita sem a necessidade de habilidades com ambientes de compilação, ou mesmo conhecimentos profundos de programação, aumentando o número de potenciais usuários.

– Possibilidade de integração com simuladores externos, sem alteração de seu código fonte, através de trocas de arquivos. Esta opção é interessante para incorporar o uso de simuladores comerciais em uma simulação.

A possibilidade de uso do *framework* GeMA para prototipação rápida de novas físicas é uma outra contribuição importante deste trabalho, principalmente em meios acadêmicos. Neste cenário, a possibilidade da descrição de novas físicas em Lua alinha-se com a tendência observada em universidades e centros de pesquisa de utilização de ferramentas de alto nível para o aprendizado, prototipação e validação de novos conceitos.

7.2 Trabalhos futuros

Há diversas linhas que podem ser seguidas na evolução do presente trabalho, tanto para melhorar os serviços oferecidos quanto para aumentar a eficiência das simulações realizadas com uso do *framework*. Algumas destas possibilidades incluem:

– Implementação de suporte a elementos 3D, tais como tetraedros e hexaedros, com funções de forma lineares e quadráticas.

– Complementar a implementação do conceito de monitor de resultados descrito na seção 4.6. A implementação atual do *framework* inclui os processos necessários para salvamento de dados, porém o salvamento automático de resultados com base em uma descrição dos mesmos e a emissão de eventos de acompanhamento ainda não foram concluídos.

– Estudo e avaliação de novos métodos para troca de dados entre malhas. Em particular é interessante estudar novos métodos de indexação espacial e compará-los com o método adotado. Também é necessário estudar e implementar métodos para tratamento de elementos com funções de forma

quadráticas, bem como para a transferência de dados entre modelos distintos de discretização, incluindo o suporte a métodos "meshless", tais como os métodos SPH (*Smoothed-particle hydrodynamics*) e DEM (*Diffuse element method*).

- Condução de um estudo mais aprofundado para avaliação de eficiência das simulações geradas com uso do *framework* GeMA e definição de estratégias para melhoria da mesma. Em particular, é importante estudar e avaliar como melhorar a eficiência da manipulação de matrizes esparsas que, segundo as medições efetuadas, chega a consumir 70% do tempo de simulação.
- Estudo e implementação de métodos para suporte à paralelização das simulações, tanto em ambientes de memória compartilhada, contando com múltiplos processadores, quanto em ambientes distribuídos. Este suporte pode ocorrer em diversos níveis, incluindo:
  - Suporte à descrição das partições do domínio pelo usuário ou à utilização de ferramentas para particionamento automático do mesmo;
  - Paralelização da construção das matrizes globais contendo o sistema de equações a ser resolvido;
  - Paralelização da solução do sistema de equações resultante;
  - Paralelização, pelo orquestrador, de processos independentes que possam ser executados concomitantemente.
- Desenvolvimento de uma interface gráfica genérica para descrição dos modelos de simulação lidos pelo *framework* GeMA. Esta interface deve ser extensível para acomodar automaticamente novos *plugins* introduzidos pelo usuário no ambiente. Através do uso de metadados fornecidos pelos mesmos, os formulários de entrada de dados para configuração de processos, físicas, resolvedores numéricos, malhas e conjuntos de propriedades podem ser individualizados de acordo com os atributos suportados pelo *plugin*. Este ambiente pode servir ainda como um editor textual (ou mesmo visual) para a construção do *script* de orquestração e para receber os eventos gerados pelo monitoramento da execução da simulação.

No âmbito do projeto no qual o desenvolvimento do *framework* GeMA está inserido, algumas destas possibilidades incluem:

- Implementação de novos métodos de discretização a serem disponibilizados junto dos *plugins* padrão que acompanham o ambiente GeMA. Em particular, há necessidade de desenvolvimento de novas físicas para cálculos de dissolução por processos químicos, baseados no método de volumes finitos e a necessidade de implementação dos métodos SPH e DEM.

– Melhoria da capacidade de integração com simuladores existentes através do estudo dos formatos adotados por simuladores comerciais, em especial do sistema Abaqus, e implementação de rotinas para leitura e escrita dos mesmos. Implementação de classes para integração do sistema GeoFlux3D, escrito em Fortran, como um componente do *framework* GeMA.

Referências Bibliográficas

ANDERSON, E.; BAI, Z.; BISCHOF, C.; BLACKFORD, S.; DEMMEL, J.; DONGARRA, J.; DU CROZ, J.; GREENBAUM, A.; HAMMARLING, S.; MCKENNEY, A. ; SORENSEN, D.. **LAPACK Users' Guide**. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999. 4.5

ATHY, L. F.. **Density, porosity and compaction of sedimentary rocks**. Bulletin of the American Association of Petroleum Geophysicists, 14(1):1–24, jan 1930. 5.3.1

BANKS, J.. **Introduction to simulation**. In: PROCEEDINGS OF THE 31ST CONFERENCE ON WINTER SIMULATION: SIMULATION—A BRIDGE TO THE FUTURE - VOLUME 1, WSC '99, p. 7–13, New York, NY, USA, 1999. ACM. 2

BEARDSMORE, G. R.; CULL, J. P.. **Crustal Heat Flow, a guide to measurement and modelling**. Cambridge University Press, 2001. 5.3.3

BEGHINI, L. L.; PEREIRA, A.; ESPINHA, R.; MENEZES, I. F.; CELES, W. ; PAULINO, G. H.. **An object-oriented framework for finite element analysis based on a compact topological data structure**. Advances in Engineering Software, 68:40 – 48, 2014. 3, 4.4

BEHAR, F.; VANDENBROUCKE, M.; TANG, Y.; MARQUIS, F. ; ESPITALIE, J.. **Thermal cracking of kerogen in open and closed systems: determination of kinetic parameters and stoichiometric coefficients for oil and gas generation**. Organic Geochemistry, 26(5-6):321–339, mar 1997. **??**, **??**

BIRSAN, D.. **On plug-ins and extensible architectures**. Queue, 3(2):40, mar 2005. 4.1

BLANCHETTE, J.; SUMMERFIELD, M.. **C++ GUI Programming with Qt 4 (2nd Edition) (Prentice Hall Open Source Software Development Series)**. Prentice Hall, 2008. 4.6.2

BOX, G.. **Robustness in the strategy of scientific model building**. In: ROBUSTNESS IN STATISTICS, p. 201–236. Elsevier BV, 1979. 2

BRAUN, R. L.; BURNHAM, A. K.. **Analysis of chemical reaction kinetics using a distribution of activation energies and simpler models**. Energy & Fuels, 1(2):153–161, 1987. 5.3.3

BRANDYBERRY, M.; CAMPBELL, M.; WASISTHO, B.; NAJJAR, F.; JACKSON, T.; SCHWENK, D. ; DICK, W.. **Rocstar simulation suite: An advanced 3-d multiphysics, multiscale computational framework for tightly coupled, fluid-structure-thermal applications**. In: 48TH AIAA/ASME/SAE/ASEE JOINT PROPULSION CONFERENCE & EXHIBIT. American Institute of Aeronautics and Astronautics (AIAA), jul 2012. 3.2

CERVANTES, H.; CHARLESTON-VILLALOBOS, S.. **Using a lightweight workflow engine in a plugin-based product line architecture**. In: Gorton, I.; Heineman, G.; Crnković, I.; Schmidt, H.; Stafford, J.; Szyperski, C. ; Wallnau, K., editors, COMPONENT-BASED SOFTWARE ENGINEERING, volumen 4063 de **Lecture Notes in Computer Science**, p. 198–205. Springer Berlin Heidelberg, 2006. 4.1

CRISFIELD, M. A.. **Non-Linear Finite Element Analysis of Solids and Structures**. Wiley, 1991. 5.2.2

DEMING, D.; CHAPMAN, D. S.. **Thermal histories and hydrocarbon generation: example from utah-wyoming thrust belt**. AAPG Bulletin, 73(12):1455–1471, 1989. 5.3.2

DEMMEL, J. W.; EISENSTAT, S. C.; GILBERT, J. R.; LI, X. S. ; LIU, J. W. H.. **A supernodal approach to sparse partial pivoting**. SIAM J. Matrix Analysis and Applications, 20(3):720–755, 1999. 4.5

DI PRIMIO, R.; HORSFIELD, B.. **From petroleum-type organofacies to hydrocarbon phase prediction**. AAPG Bulletin, 90(7):1031–1058, jul 2006. 5.7

DUBOIS-PÈLERIN, Y.; ZIMMERMANN, T. ; BOMME, P.. **Object-oriented finite element programming: Ii. a prototype program in smalltalk**. Computer Methods in Applied Mechanics and Engineering, 98(3):361 – 397, 1992. 3, 4.4

DUBOIS-PÈLERIN, Y.; ZIMMERMANN, T.. **Object-oriented finite element programming: Iii. an efficient implementation in c++**. Computer Methods in Applied Mechanics and Engineering, 108(1–2):165 – 183, 1993. 3, 4.4

FELIPPA, C. A.. **Introduction to finite element methods**. Lecture Notes for the course Introduction to Finite Elements Methods at the Aerospace Engineering

Sciences Department of the University of Colorado at Boulder, 2004. 2, 2.1, 4.2, 6, 1, 4.7, 4.22, 5.2.1, 5.2.2, 5.2.2

FISH, J.. **Bridging the scales in nano engineering and science**. Journal of Nanoparticle Research, 8(5):577–594, sep 2006. 1.1, 2.2

FREY, P.. **MEDIT : An interactive Mesh visualization Software**. Technical Report RT-0253, INRIA, Dec. 2001. 2

GAMMA, E.; HELM, R.; JOHNSON, R. ; VLISSIDES, J.. **Design Patterns: Elements of Reusable Object-Oriented Software**. Addison-Wesley Professional, 1994. 1, 4.6.2

GASTON, D.; NEWMAN, C.; HANSEN, G. ; LEBRUN-GRANDIÉ, D.. **Moose: A parallel computational framework for coupled systems of nonlinear equations**. Nuclear Engineering and Design, 239(10):1768 – 1778, 2009. 3.3

GATTASS, M.; FILHO, W.; FONSECA, G. ; DE MATEMÁTICA APLICADA E COMPUTACIONAL, S. B.. **Computação gráfica aplicada ao método dos elementos finitos: minicurso**. SBMAC, 1991. 2, 2.1

HANTSCHEL, T.; KAUERAUF, A. I.. **Fundamentals of Basin and Petroleum Systems Modeling**. Springer, 2009. 5.3, 5.3.2, 5.5, 5.3.2, 6.1.2

HOLMAN, J.. **Heat transfer**. Mechanical engineering series. McGraw-Hill, 1989. 6.1.1

HORSTEMEYER, M. F.. **Multiscale modeling: A review**. In: Leszczynski, J.; Shukla, M., editors, PRACTICAL ASPECTS OF COMPUTATIONAL CHEMISTRY, p. 87–135. Springer Science + Business Media, 2009. 2.2

HUANG, C.; LAWLOR, O. ; KALÉ, L. V.. **Adaptive mpi**. In: PROCEEDINGS OF THE 16TH INTERNATIONAL WORKSHOP ON LANGUAGES AND COMPILERS FOR PARALLEL COMPUTING (LCPC 2003), LNCS 2958, p. 306–322, College Station, Texas, October 2003. 3.2

IERUSALIMSCHY, R.; DE FIGUEIREDO, L. H. ; FILHO, W. C.. **Lua—an extensible extension language**. Software: Practice and Experience, 26(6):635–652, 1996. 1.2

IERUSALIMSCHY, R.. **Programming in Lua**. Roberto Ierusalimschy, 2003. 1.2

IERUSALIMSCHY, R.; DE FIGUEIREDO, L. H. ; CELES, W.. **Lua 5.1 Reference Manual**. Lua.org, 2006. 4.4.2

JIAO, X.; ZHENG, G.; ALEXANDER, P. A.; CAMPBELL, M. T.; LAWLOR, O. S.; NORRIS, J.; HASELBACHER, A. ; HEATH, M. T.. **A system integration framework for coupled multiphysics simulations**. Engineering with Computers, 22(3-4):293–309, aug 2006. 3.2

KALÉ, L.; KRISHNAN, S.. **Charm++: A portable concurrent object oriented system based on c++**. In: Paepcke, A., editor, PROCEEDINGS OF OOPSLA'93, p. 91–108. ACM Press, September 1993. 3.2

KNOLL, D.; KEYES, D.. **Jacobian-free newton–krylov methods: a survey of approaches and applications**. Journal of Computational Physics, 193(2):357 – 397, 2004. 3.3

LETHBRIDGE, P.. **Multiphysics analysis**. The Industrial Physicist, jan 2005. 2.2

LEWIS, R. W.; NITHIARASU, P. ; SEETHARAMU, K. N.. **Fundamentals of the Finite Element Method for Heat and Fluid Flow**. John Wiley & Sons, 2004. 1.3, 2.1, 4.3.1, 4.4.1, 5.1, 5.1, 5.1, 6.1.1, 6.1.1, 6.1.3

LI, X.; DEMMEL, J.; GILBERT, J.; IL. GRIGORI; SHAO, M. ; YAMAZAKI, I.. **Superlu users' guide**. Technical Report LBNL-44289, Lawrence Berkeley National Laboratory, September 1999. `http://crd.lbl.gov/~xiaoye/SuperLU/`. Last update: August 2011. 4.5

LI, G.; JIN, X. ; ALUM, N.. **Meshless methods for numerical solution of partial differential equations**. In: Yip, S., editor, HANDBOOK OF MATERIALS MODELING, p. 2447–2474. Springer Netherlands, 2005. 2.1

LOGAN, D. L.. **A First Course in the Finite Element Method**. Cengage Learning, 2011. 5.2.3

LYLY, M.; RUOKOLAINEN, J. ; JÄRVINEN, E.. **Elmer - a finite element solver for multiphysics.** Technical report, CSC-report on scientific computing., 1999-2000. 3.1

MARTHA, L. F.; JUNIOR, E. P.. **An object-oriented framework for finite element programming**. In: PROCEEDINGS OF THE FIFTH WORLD CONGRESS ON COMPUTATIONAL MECHANICS, 2002. 4.4

MYERS, G. J.. **The Art of Software Testing, Second Edition**. Wiley, 2004. 6.6

NIE, J.; HOPKINS, D.; CHEN, Y. ; HSIEH, H.. **Development of an object-oriented finite element program with adaptive mesh refinement for multi-physics applications**. Advances in Engineering Software, 41(4):569 – 579, 2010. 4.4

NOVASCONE, S.; SPENCER, B.; ANDRS, D.; WILLIAMSON, R.; HALES, J. ; PEREZ, D.. **Results from tight and loose coupled multiphysics in nuclear fuels performance simulations using bison**. In: PROCEEDINGS OF THE 2013 INTERNATIONAL CONFERENCE ON MATHEMATICS AND COMPUTATIONAL METHODS APPLIED TO NUCLEAR SCIENCE AND ENGINEERING-M AND C 2013. 2013. 2.2, 3.3

ODEN, J. T.; BELYTSCHKO, T.; FISH, J.; HUGHES, T. J.; JOHNSON, C.; KEYES, D.; LAUB, A.; PETZOLD, L.; SROLOVITZ, D. ; YIP, S.. **Revolutionizing engineering science through simulation: Report of the national science foundation blue ribbon panel on simulation-based engineering science**. Technical report, NSF, USA, may 2006. 1

PATZÁK, B.; BITTNAR, Z.. **Design of object oriented finite element code**. Advances in Engineering Software, 32(10–11):759 – 767, 2001. 4.4

PEIRÓ, J.; SHERWIN, S.. **Finite difference, finite element and finite volume methods for partial differential equations**. In: Yip, S., editor, HANDBOOK OF MATERIALS MODELING, p. 2415–2446. Springer Netherlands, 2005. 2, 2.1

POST, D. E.; KENDALL, R. P.. **Software project management and quality engineering practices for complex, coupled multiphysics, massively parallel computational simulations: Lessons learned from ASCI**. International Journal of High Performance Computing Applications, 18(4):399–416, nov 2004. 1.1

RÅBACK, P.; MALINEN, M.. **Overview of Elmer**. CSC – IT Center for Science, 2014. 3.1

ROGERS, G. F.. **Framework-Based Software Development in C++**. Prentice Hall, 1997. 1

RUOKOLAINEN, J.; MALINEN, M.; RÅBACK, P.; ZWINGER, T.; PURSULA, A. ; BYCKLING, M.. **Elmer Solver Manual**. CSC – IT Center for Science, 2014. 3.1

SANDERSON, C.. **Armadillo: An open source c++ linear algebra library for fast prototyping and computationally intensive experiments.** Technical report, NICTA, 2010. 4.5

SCHÄFER, M.. **Computational Engineering - Introduction to Numerical Methods**. Springer, 2006. 2.1

SCLATER, J. G.; CHRISTIE, P. A. F.. **Continental stretching: An explanation of the post-mid-cretaceous subsidence of the central north sea basin**. Journal of Geophysical Research, 85(B7):3711–3739, 1980. 5.3.1

SCRIMIERI, D.; AFAZOV, S. M.; BECKER, A. A. ; RATCHEV, S. M.. **Fast mapping of finite element field variables between meshes with different densities and element types**. Advances in Engineering Software, 67:90–98, 2014. 4.2.4

SEKIGUCHI, K.. **A method for determining terrestrial heat flow in oil basinal areas**. Tectonophysics, 103(1–4):67 – 79, 1984. Terrestrial Heat Flow Studies and the Structure of the Lithosphere. 5.3.2

SOMERTON, W.. **Thermal Properties and Temperature-Related Behavior of Rock/Fluid Systems (Developments in Petroleum Science)**. Elsevier Science, 1992. 5.3.2

SWEENEY, J. J.; BURNHAM, A. K.. **Evaluation of a simple model of vitrinite reflectance based on chemical kinetics**. The American Association of Petroleum Geologists Bulletin, 74(10):1559–1570, 1990. 5.3.3, 5.6

WANGEN, M.. **Modelling heat and fluid flow in sedimentary basins by the finite element method**. Int. J. Numer. Anal. Methods Geomech., 15(10):705–733, oct 1991. 6.5

WANGEN, M.. **Physical Principels of Sedimentary Basin Analysis**. Cambridge University Press, 2010. 5.1, 5.3

WAPLES, D. W.; WAPLES, J. S.. **A review and evaluation of specific heat capacities of rocks, minerals, and subsurface fluids. part 1: Minerals and nonporous rocks**. Natural Resources Research, 13(2):97–122, jun 2004. 5.3.2

WELLER, H. G.; TABOR, G.; JASAK, H. ; FUREBY, C.. **A tensorial approach to computational continuum mechanics using object-oriented techniques**. Comput. Phys., 12(6):620–631, Nov. 1998. 3

WHEELER, M. F.; WHEELER, J. A. ; PESZYNSKA, M.. **A distributed computing portal for coupling multi-physics and multiple domains in porous media**. In: COMPUTATIONAL METHODS IN WATER RESOURCES, p. 167–174, 2000. 3

ZIENKIEWICZ, O. C.; TAYLOR, R. L. ; ZHU, J. Z.. **The Finite Element Method: Its Basis and Fundamentals**. Elsevier, 6a edition, 2005. 2.1, 4.2.2, 4.10, 5.2.2

ZIMMERMANN, T.; DUBOIS-PÈLERIN, Y. ; BOMME, P.. **Object-oriented finite element programming: I. governing principles**. Computer Methods in Applied Mechanics and Engineering, 98(2):291 – 303, 1992. 3, 4.4

A
Principais interfaces abstratas

Este apêndice tem como objetivo apresentar conceitualmente as principais características das interfaces abstratas propostas pelo *framework* GeMA. Por ser uma descrição conceitual, a lista de métodos por classe não é completa e a própria assinatura apresentada das funções não corresponde exatamente à assinatura real. Prefixos aos nomes das classes foram removidos, bem como parâmetros com valores padrão sem relevância para o entendimento do objetivo básico das funções.

Mesh
`ValueInfo* nodeCoordInfo()` Retorna metadados com informações sobre as coordenadas dos nós da malha.
`int numNodes()` Retorna o número de nós presentes na malha.
`ValueAccessor* nodeCoordAccessor(Unit desiredUnit)` Retorna um *accessor* para consulta às coordenadas dos nós da malha. Valores retornados são convertidos para a unidade especificada.
`ValueInfo* nodeValueInfo(string id)` Retorna metadados com informações sobre a variável de estado ou atributo de nome `id`.
`ValueAccessor* nodeValueAccessor(string id, Unit desiredUnit)` Retorna um *accessor* para consulta e alteração de valores associados à variável de estado ou atributo de nome `id`. Valores retornados são convertidos para a unidade especificada.
`ValueAccessor* nodeValueAccessor(string id, int snum, bool locked, Unit desiredUnit)` Semelhante ao método anterior. O parâmetro `snum` define o estado a ser consultado e o parâmetro `locked` o comportamento do *accessor* retornado quando novos estados são criados.
`bool addNodeValueSet(ValueInfo* info)` Adiciona um novo conjunto de dados aos nós da malha, recebendo como parâmetro os metadados que o definem.

Tabela A.1: Principais métodos da classe **Mesh**.

CellMesh (estende **Mesh**)
`int numCells()` Retorna o número de células da malha.
`Cell* cell(int cellIndex)` Retorna uma célula da malha, dado seu índice (entre 0 e `numCells()`).
`StringList cellGroupIds()` Retorna uma lista contendo o nome dos grupos de células associados à malha.
`int numCellsInGroup(int groupIndex)` Retorna o número de células em um grupo identificado por seu índice na lista retornada por `cellGroupIds()`.
`Cell* cellInGroup(int groupIndex, int cellIndex)` Retorna uma célula da malha pertencente a um grupo, identificada pelo número do grupo e pelo índice da célula no grupo (entre 0 e `numCellsInGroup()`).
`Map<String, CellBoundary*>& cellBoundaryGroups()` Retorna um mapa contendo a lista de bordas associadas à malha, indexado pelo nome dado à borda.
`List<PropertySet*>& propertySets()` Retorna uma lista contendo os grupos de propriedades associados à malha.
`ValueInfo* cellPropertyInfo(String id)` Retorna metadados com informações sobre a propriedade de nome `id`.
`CellAccessor* cellPropertyAccessor(String id, Unit desiredUnit)` Retorna um *accessor* para consulta de valores associados à propriedade de nome `id`. Valores retornados são convertidos para a unidade especificada.
`ValueInfo* cellAttributeInfo(String id)` Retorna metadados com informações sobre o atributo de nome `id`.
`CellAccessor* cellAttributeAccessor(String id, Unit desiredUnit)` Retorna um *accessor* para consulta e alteração de valores associados ao atributo de nome `id`. Valores retornados são convertidos para a unidade especificada.
`CellAccessor* cellAttributeAccessor(String id, int snum, bool locked, Unit desiredUnit)` Semelhante ao método anterior. O parâmetro `snum` define o estado a ser consultado e o parâmetro `locked` o comportamento do *accessor* retornado quando novos estados são criados.
`bool addCellAttributeSet(ValueInfo* info)` Adiciona um novo conjunto de dados às células da malha, recebendo como parâmetro os metadados que o definem.

Tabela A.2: Principais métodos adicionados pela classe **CellMesh**.

Cell
`int cellId()` Retorna o identificador da célula.
`CellType type()` Retorna o tipo de célula (Quad4, Quad9, Tri3, ...).
`CellGeometry geometry()` Retorna um objeto **CellGeometry** para consulta à geometria de células deste tipo.
`int numNodes()` Retorna o número de nós da célula.
`int nodeIndex(int localIndex)` Retorna o índice global na malha de um nó da célula identificado por `localIndex` (entre 0 e `numNodes()`). A ordem local dos nós na célula é definida pelo objeto **CellGeometry** associado a este tipo de célula.
`int propertyIndex(int propertySet)` Dado o índice de um grupo de propriedades pertencente à malha, retorna a "linha" na tabela de propriedades a que esta célula está associada.

Tabela A.3: Principais métodos da classe **Cell**.

ElementMesh (estende **CellMesh**)
`IntegrationRule* elementIntegrationRule(CellType type, int ruleSet)` Retorna a regra de integração associada com o tipo de célula fornecido como parâmetro, para o conjunto de regras selecionado.
`BorderIntegrationRule* borderIntegrationRule(CellType type, int ruleSet)` Retorna a regra de integração associada a bordas do tipo de célula fornecido como parâmetro, para o conjunto de regras selecionado.
`ValueInfo* gaussAttributeInfo(String id)` Retorna metadados com informações sobre o atributo associado com os pontos de integração dos elementos, de nome `id`.
`GaussAccessor* gaussAttributeAccessor(String id, Unit desiredUnit)` Retorna um *accessor* para consulta e alteração de valores associados ao atributo associado com os pontos de integração dos elementos, de nome `id`. Valores retornados são convertidos para a unidade especificada.
`GaussAccessor* gaussAttributeAccessor(String id, int snum, bool locked, Unit desiredUnit)` Semelhante ao método anterior. O parâmetro `snum` define o estado a ser consultado e o parâmetro `locked` o comportamento do *accessor* retornado quando novos estados são criados.
`bool addGaussAttributeSet(ValueInfo* info)` Adiciona um novo conjunto de dados aos pontos de integração dos elementos da malha, recebendo como parâmetro os metadados que o definem.

Tabela A.4: Principais métodos adicionados pela classe **ElementMesh**.

Element (estende **Cell**)
`Shape* shape()` Retorna a função de forma associada com este tipo de elemento.

Tabela A.5: Principais métodos adicionados pela classe **Element**.

ValueAccessor
`ValueInfo* info()` Retorna os metadados associados ao conjunto de dados associado.
`int size()` Retorna o número de valores presentes no conjunto de dados associado (o que significa que os dados podem ser indexados de 0 a `size()`-1.
`int valueSize()` Retorna o tamanho de cada valor retornado pelo *accessor* (1 para valores escalares, número de linhas * número de colunas para vetores e matrizes).
`Unit unit()` Retorna a unidade em que os dados são retornados pelo *accessor*.
`const double* valueAt(int index, const Vector* coord)` Consulta os dados armazenados no índice fornecido, retornando um ponteiro para um vetor interno onde os mesmos estão armazenados. O tamanho do vetor retornado é igual a `valueSize()`. O conteúdo deste vetor é válido apenas até que outra chamada seja feita ao *accessor*. O parâmetro `coord`, se diferente de `NULL`, fornece o ponto do elemento onde os valores estão sendo consultados. Isto é necessário quando o *accessor* avalia funções que dependem de parâmetros interpolados no elemento. Esta função é a base de uma família de funções que possui variantes que passam `NULL` como coordenada e/ou retornam os dados obtidos como vetores, matrizes ou mesmo escalares.
`const double* partialAt(int index, int dof, const Vector* coord)` Retorna a derivada parcial, calculada numericamente, do valor no índice fornecido, em relação ao grau de liberdade indicado por `dof` (o qual está associado à uma variável de estado). Retorna 0.0 se o valor não estiver associado a uma função do grau de liberdade. O parâmetro `coord`, se diferente de `NULL`, fornece o ponto do elemento onde os valores estão sendo consultados. Esta função é a base de uma família de funções que possui variantes que passam `NULL` como coordenada e/ou retornam os dados obtidos como vetores, matrizes ou mesmo escalares.
`bool setValue(int index, const double* value)` Altera os dados armazenados no índice fornecido. Esta função é a base de uma família de funções que possui variantes que recebem o valor a ser alterado como um escalar, vetor ou matriz.

Tabela A.6: Principais métodos da classe **ValueAccessor**.

FemPhysics (1/2)
`const ElementDof* dofMapping(CellType type)` Retorna objeto especificando os graus de liberdade de cada nó do tipo de elemento especificado.
`Unit dofUnit(int dof)` Retorna a unidade em que os valores do grau de liberdade especificado são calculados pela física.
`bool supportsCellType(CellType type)` Retorna um *booleano* indicando se a física suporta ou não elementos do tipo especificado.

`bool beforeElementStiffnessLoop(`
`const FemMatrixSet& elemMatrices,`
`const FemVectorSet& elemVectors)`
Função chamada pelo processo de elementos finitos para notificar as físicas de que o processo de *assembly* irá começar. Útil, por exemplo, para a criação de matrizes necessárias nos cálculos uma única vez.

`void afterElementStiffnessLoop()` Função chamada pelo processo de elementos finitos para notificar as físicas que o processo de *assembly* está terminado.

`bool fillElementData(const Element* e,`
`FemMatrixSet& elemMatrices,`
`FemVectorSet& elemVectors)`
Função chamada pelo processo de elementos finitos para que a física retorne as contribuições do elemento `e` para as matrizes / vetores globais especificados por `elemMatrices` e `elemVectors`.

`bool supportsBc(const BoundaryCondition* bc)` Retorna um *booleano* indicando se a condição de contorno recebida como parâmetro influencia ou não a física.

`bool fillElementDataForBc(const Element* e,`
`const BoundaryCondition* bc,`
`int bcIndex, int bcListIndex,`
`int border,`
`FemMatrixSet& elemMatrices,`
`FemVectorSet& elemVectors)`
Função chamada pelo processo de elementos finitos para que a física retorne as contribuições da condição de contorno especificada, associada à borda `border` do elemento `e`, para as matrizes / vetores globais especificados por `elemMatrices` e `elemVectors`. Os parâmetros `bcIndex` e `bcListIndex` identificam a posição do elemento na condição de contorno e permite o acesso a seus dados.

Tabela A.7: Principais métodos da classe **FemPhysics** (1/2).

FemPhysics (2/2)
`bool fixedNodalForces(List<int>& nodes, List<int>& dof, List<double>& values)` Função chamada pelo processo de elementos finitos para que a física retorne valores que serão adicionados ao vetor global de forças, simplificando o tratamento de situações onde as forças aplicadas sobre nós são especificadas pelo modelo.
`bool fixedBoundaryConditions(List<int>& nodes, List<int>& dof, List<double>& values)` Função chamada pelo processo de elementos finitos para que a física retorne valores de condições de tipo "Dirichlet", fixando o valor de um conjunto de graus de liberdade.
`bool calcDerivedResults()` Função chamada pelo processo de elementos finitos para que a física calcule resultados derivados após a solução do sistema de equações.

Tabela A.8: Principais métodos da classe **FemPhysics** (2/2).

IntegrationRule
`int numPoints()` Retorna o número de pontos de integração da regra.
`void integrationPoint(int index, Vector& natCoord, double* weight)` Dado o índice de um ponto de integração, retorna sua coordenada e seu peso.

Tabela A.9: Principais métodos da classe **IntegrationRule**.

Shape
`void shapeValues(const Vector& ncoord, Vector& N)` Avalia as funções de forma no ponto `ncoord` e preenche o vetor `N`.
`void shapePartials(const Vector& ncoord, Matrix& dN, bool transposed)` Avalia as derivadas parciais das funções de forma em relação às coordenadas naturais no ponto `ncoord` e preenche a matriz `dN`. O parâmetro `transposed` controla a orientação dos valores na matriz.
`void shapeCartesianPartials(const Vector& ncoord, const Matrix& X, Matrix& dN, double* scaledDetJ, bool transposed)` Avalia as derivadas parciais das funções de forma em relação às coordenadas cartesianas no ponto `ncoord`, preenchendo a matriz `dN` e o parâmetro `scaledDetJ` com o determinante do Jacobiano da transformação. A matriz `X` fornece as coordenadas dos nós do elemento. O parâmetro `transposed` controla a orientação dos valores na matriz de resultado.
`void jacobian(const Vector& ncoord, const Matrix& X, Matrix& J, bool transposed)` Calcula o Jacobiano da transformação no ponto `ncoord`, preenchendo a matriz `J`. A matriz `X` fornece as coordenadas dos nós do elemento. O parâmetro `transposed` controla a orientação dos valores na matriz de resultado.
`double interpolate(const Vector& nodeValues, const Vector& ncoord)` Dado um vetor com valores para cada nó do elemento, interpola os mesmos no ponto `ncoord`.
`bool cartesianToNatural(const Vector& coord, const Matrix& X, Vector& ncoord)` Dado um ponto em coordenadas cartesianas `coord` e as coordenadas de cada nó do elemento (`X`), calcula suas coordenadas naturais, retornando *true* se o ponto estiver dentro do elemento.
`void naturalToCartesian(const Vector& ncoord, const Matrix& X, Vector& coord)` Dado um ponto em coordenadas naturais `ncoord`, preenche `coord` com suas coordenadas cartesianas. A matriz `X` fornece as coordenadas dos nós do elemento.

Tabela A.10: Principais métodos da classe **Shape**.

SolverMatrix
`int nlin()` Retorna o número de linhas da matriz.
`int ncol()` Retorna o número de colunas da matriz.
`double at(int lin, int col)` Retorna o valor armazenado na linha e coluna especificadas.
`void set(int lin, int col, double value)` Altera o valor na linha e coluna especificadas.
`void mul(const Vector& a, Vector& b)` Multiplica a matriz (X) pelo vetor a armazenando o resultado em b ($b = X * a$).
`void setSymmetric(bool sym)` Permite especificar se a matriz é simétrica ou não. Alguns resolvedores numéricos podem utilizar esta informação no processo de solução de sistemas lineares.

Tabela A.11: Principais métodos da classe **SolverMatrix**.