

**Leonardo de Paula Batista Benevides**

**Técnicas de instanciação aplicadas à  
renderização eficiente de campos tensoriais**

**Projeto Final de Graduação**

Projeto Final apresentado ao Curso Engenharia de Computação  
como requisito parcial para obtenção do título de Engenheiro de  
Computação

Orientador: Prof. Waldemar Celes

Rio de Janeiro  
Novembro de 2012

**Leonardo de Paula Batista Benevides**

**Técnicas de instanciação aplicadas à  
renderização eficiente de campos tensoriais**

Centro Técnico Científico, Departamento de Informática,  
Engenharia de Computação.

**Prof. Waldemar Celes**  
Orientador  
Departamento de Informática — PUC–Rio

Rio de Janeiro, 21 de Novembro de 2012

Todos os direitos reservados. É proibida a reprodução total ou parcial do trabalho sem autorização da universidade, do autor e do orientador.

**Leonardo de Paula Batista Benevides**

Ficha Catalográfica

Técnicas de instanciação aplicadas à renderização eficiente de campos tensoriais / Leonardo de Paula Batista Benevides; orientador: Waldemar Celes. — Rio de Janeiro : PUC–Rio, Departamento de Informática, 2012.

v., 30 f: il. ; 29,7 cm

1. Projeto Final de Graduação - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática.

Inclui referências bibliográficas.

1. Informática – Tese. 2. Linguagem visual, Programação por usuários finais. I. Celes, Waldemar. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 510

## Resumo

; Celes, Waldemar. **Técnicas de instanciação aplicadas à renderização eficiente de campos tensoriais**. Rio de Janeiro, 2012. 30p. Projeto Final de Graduação — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Este documento apresenta os principais conceitos de um sistema de *script* de uma aplicação voltada para o pós-processamento de simulações de reservatórios de petróleo e a proposta e implementação de uma ferramenta que disponibilize parte dos recursos obtidos a partir dos *scripts* usuais aos programadores. A ferramenta implementada é uma calculadora de propriedades que permite que o usuário gere novas propriedades a partir de simples expressões aritméticas desenvolvidas em uma interface gráfica similar a uma calculadora comum. Este documento também apresenta um breve resumo dos estudos feitos acerca de linguagens de programação visual e a proposta e início de implementação de uma linguagem de programação visual simplificada.

## Palavras-chave

Linguagem visual, Programação por usuários finais.

## **Abstract**

; Celes, Waldemar. **A visual programming tool for postprocessing of oil reservoir simulations** . Rio de Janeiro, 2012. 30p.  
MsC Thesis — Department of Informática, Pontifícia Universidade Católica do Rio de Janeiro.

This document presents the main concepts of a script system of an application focused on the postprocessing of oil reservoir simulations and the proposal and the implementation of a tool that provides part of the scripts resources to users that are not programmers. This tool is an property calculator that lets the users generate new properties from simple arithmetic expressions that are developed in an graphic interface that looks like an ordinary calculator. This document also presents a brief summary of studies about visual programming languages and the proposal and the beginning of a simplified visual programming language.

## **Keywords**

Visual language, End users programming .

## Sumário

1	Introdução	9
2	O sistema de Scripts	11
2.1	Lua	11
2.2	Conceitos básicos do sistema de Script	12
2.3	Acesso e criação de dados	13
2.4	Vetores de dados de propriedades	14
3	Calculadora de propriedades	16
3.1	Sintaxe das expressões	16
3.2	Implementação	17
4	Linguagens de Programação Visual	23
5	Início da implementação de uma VPL	25
6	Considerações finais e projetos futuros	27
	Referências Bibliográficas	28
A	Exemplos práticos	29
A.1	Cálculo da razão gas-óleo (RGO)	29
A.2	Cálculo sedimentos básicos e água (BSW)	29
A.3	Cálculo do volume de óleo em condições de reservatório	29

## Lista de figuras

3.1	Interface da Calculadora	17
A.1	Exemplo de visualizaçãõ $\ell^{\frac{1}{2}}_2$ $\ell^{\frac{1}{2}}_2$ o	30

**Lista de tabelas**



# 1

## Introdução

O **Geresim** é uma aplicação desenvolvida por meio de uma parceria firmada entre o **TeCGraf/PUC-Rio** e a **PETROBRAS (E&P/ER)** com o intuito de melhorar o processo de modelagem do escoamento em meio poroso em reservatórios de petróleo. Ele é constituído de um ambiente de trabalho que permite construir, alterar, manter, importar e exportar modelos numéricos para os principais simuladores comerciais, tirando proveito de uma série de facilidades voltadas para a agilidade e a eficácia da montagem dos modelos. Adicionalmente, o **Geresim** executa uma análise de consistência que elimina a maioria dos erros mais comuns no processo de modelagem e oferece um ambiente para analisar modelos e dados de reservatórios, utilizando visualizações bidimensionais de mapas e seções, tridimensionais do modelo ou gráficas de elementos do modelo (Tecg06).

Uma das aplicações com maior usabilidade no **Geresim** é um poderoso sistema de *scripts* que possibilita acesso programável aos dados dos modelos de reservatórios visualizados e disponibiliza diversos recursos como geração de dados de forma procedimental e flexibilização nas operações com os dados. Esse acesso programável é feito através da linguagem de programação **Lua** (Ieru03).

No entanto, como requer certo conhecimento de técnicas de programação do usuário e este na maioria das vezes tem pouca ou até mesmo nenhuma experiência como programador, o sistema de *scripts* muitas vezes acaba sendo subutilizado e os que necessitam do mesmo acabam delegando aos próprios desenvolvedores do software a implementação de novos *scripts* de programação.

Nesse contexto, torna-se interessante e necessário (pois, foi sugerido pelo próprio cliente) o estudo de alternativas de programação textual que disponibilize ao usuário não o programador pelo menos parte dos recursos do sistema de *scripts* do **Geresim**. Por isso, esse projeto teve como metas principais pesquisar e desenvolver ferramentas que cumpram esse objetivo.

Este trabalho foi dividido da seguinte forma: a seção 2 apresenta de forma breve o atual sistema de script do **Geresim**; a seção 3 descreve

a implementação e funcionamento de uma calculadora de propriedades desenvolvida no projeto; a seção 4 apresenta um breve resumo dos estudos feitos acerca de linguagens de programação visual; a seção 5 apresenta o início da implementação de uma ferramenta de programação visual; a seção 6 apresenta algumas conclusões e propostas de trabalhos futuros.

## 2

### O sistema de Scripts

O sistema de *scripts* do **Geresim** possibilita acesso programático aos dados dos modelos de reservatório visualizados. Esse acesso programático é feito através da linguagem de programação **Lua**. Por isso o sistema disponibiliza diversos recursos de programação sendo possível a implementação de *scripts* diversos que podem acessar todos os dados dos modelos de reservatório.

Assim é possível gerar dados de forma procedimental, como criar novos modelos e novos dados a serem associados a modelos, manipular dados de forma flexível, como usá-los como operandos de diversas operações e sem muitas restrições e criar *scripts* temporários. Por isso o sistema pode funcionar como uma máquina de calcular que opera com os dados dos modelos, recurso que será mais discutido na seção 3.

Nessa seção será feita uma rápida descrição da linguagem de programação **Lua** e uma apresentação do funcionamento do sistema de *script* do **Geresim**.

#### 2.1

##### Lua

**Lua** é uma linguagem de programação poderosa, rápida e leve, projetada justamente para estender aplicações. Combina sintaxe simples para programação procedural com poderosas construções para descrição de dados baseadas em tabelas associativas e semântica extensiva. **Lua** é tipada dinamicamente, é interpretada a partir de bytecodes para uma máquina virtual baseada em registradores, e tem gerenciamento automático de memória com coleta de lixo incremental. Essas características fazem de **Lua** uma linguagem ideal para configuração, automação (*scripting*) e prototipagem rápida. A linguagem é usada em muitas aplicações industriais, com ênfase em sistemas embutidos e jogos, onde é atualmente a linguagem de *script* mais usada. **Lua** tem uma merecida reputação de ótimo desempenho, em vários benchmarks aparece como a linguagem mais rápida dentre as linguagens de *script* interpretadas. (Lua)

Mais do que apenas estender aplicações, **Lua** também suporta uma abordagem de desenvolvimento de software baseada em componentes, onde é possível criar aplicações juntando componentes de alto-nível ou pré-existentes. Normalmente esses componentes são escritos em linguagens compiladas e estaticamente tipadas como **C** ou **C++**. Assim, **Lua** pode ser usada conectar tais componentes, que normalmente representam conceitos mais concretos e de baixo nível (como por exemplo estruturas de dados) que são sujeitos a muitas alterações durante o desenvolvimento do programa e normalmente exigem um maior poder de processamento. (Ieru03)

## 2.2

### Conceitos básicos do sistema de Script

O sistema de macros do **Geresim** fornece acesso a diversos objetos que representam o modelo e seus dados. Para cada objeto, tem-se acesso a suas informações e a um conjunto de métodos que permitem a manipulação das informações associadas. Os objetos a que se tem acesso no sistema de macros são:

- **Model:** objeto que representa um modelo de reservatório;
- **Field:** objeto que representa um campo do modelo;
- **Region:** objeto que representa uma região do modelo;
- **Well:** objeto que representa um poço do modelo;
- **Completion:** objeto que representa uma completação de um poço do modelo;
- **Data:** objeto que representa um conjunto de valores de propriedades, podendo ser dos seguintes tipos:
  - **MapProp:** propriedade de mapa, representada por um vetor de dados de dimensão 4:  $n_i \times n_j \times n_k \times n_t$  mero de timesteps<sup>1</sup> de mapa;
  - **FieldProp:** propriedade de campo, representada por um vetor de dados de dimensão 1:  $n_t$  mero de timesteps de poço;
  - **RegionProp:** propriedade de região, representada por um vetor de dados de dimensão 2:  $n_t$  mero de regiões  $\times n_t$  mero de timesteps de poço;
  - **GroupProp:** propriedade de grupo, representada por um vetor de dados de dimensão 2:  $n_t$  mero de grupos  $\times n_t$  mero de timesteps de poço;

<sup>1</sup>explicação de timestep

- **WellProp**: propriedade de  $\text{poil}_{\frac{1}{2}}\text{o}$ , representada por um vetor de dados de dimensã $\frac{1}{2}\text{o}$  2:  $\text{nĩl}_{\frac{1}{2}}\text{mero de poil}_{\frac{1}{2}}\text{os x nĩl}_{\frac{1}{2}}\text{mero de timesteps de poil}_{\frac{1}{2}}\text{o}$ ;
- **CompProp**: propriedade de completã $\frac{1}{2}\text{i}_{\frac{1}{2}}\text{o}$ , representada por um vetor de dados de dimensã $\frac{1}{2}\text{o}$  3:  $\text{nĩl}_{\frac{1}{2}}\text{mero de poil}_{\frac{1}{2}}\text{os x nĩl}_{\frac{1}{2}}\text{mero de mĩl}_{\frac{1}{2}}\text{ximo de completã}_{\frac{1}{2}}\text{i}_{\frac{1}{2}}\text{es por poil}_{\frac{1}{2}}\text{o x nĩl}_{\frac{1}{2}}\text{mero de timesteps de poil}_{\frac{1}{2}}\text{o}$ ;
- **ExtraProp**: propriedade avulsa, representada por um vetor de dados de dimensã $\frac{1}{2}\text{o}$  1:  $\text{nĩl}_{\frac{1}{2}}\text{mero de timesteps de poil}_{\frac{1}{2}}\text{o}$ ;
- **Slice**: referiã $\frac{1}{2}\text{ncia}$  para uma “fatia” de um determinado conjunto de dados.

Pode-se ainda criar referiã $\frac{1}{2}\text{ncias}$  para outras poriã $\frac{1}{2}\text{i}_{\frac{1}{2}}\text{es}$  de um dado. Por exemplo,  $\text{iĩl}_{\frac{1}{2}}\text{ vĩl}_{\frac{1}{2}}\text{lido}$  criar uma referiã $\frac{1}{2}\text{ncia}$  para um mapa associado a um determinado timestep.

Existem dois tipos de vetores de dados: dados originais associados ao modelo e dados criados dinamicamente pelo sistema de *script*. Os dados originais podem ser consultados, mas  $\text{nĩl}_{\frac{1}{2}}\text{o}$  podem ser alterados.  $\text{Jĩl}_{\frac{1}{2}}\text{ os dados criados dinamicamente podem ser livremente alterados. Analogamente, referiã}_{\frac{1}{2}}\text{ncias a dados originais nĩl}_{\frac{1}{2}}\text{o sũl}_{\frac{1}{2}}\text{o editiã}_{\frac{1}{2}}\text{veis, enquanto que referiã}_{\frac{1}{2}}\text{ncias a dados criados dinamicamente sũl}_{\frac{1}{2}}\text{o editiã}_{\frac{1}{2}}\text{veis (na verdade, altera-se o vetor de dados referenciado).}$

Logicamente, o programa de *script* pode tambĩ $\frac{1}{2}\text{m}$  incluir a manipulaĩ $\frac{1}{2}\text{i}_{\frac{1}{2}}\text{o}$  de valores dos tipos existentes em **Lua**: booleanos, valores escalares, cadeias de caracteres (*string*), entre outros.

## 2.3

### Acesso e criaĩ $\frac{1}{2}\text{i}_{\frac{1}{2}}\text{o de dados}$

O objeto *Model* representa um resultado de simulaĩ $\frac{1}{2}\text{i}_{\frac{1}{2}}\text{o}$ , com suas propriedades de mapa e de  $\text{poil}_{\frac{1}{2}}\text{o}$ . A funiã $\frac{1}{2}\text{i}_{\frac{1}{2}}\text{o}$  que implementa o *script* recebe um ou mais modelos como parĩ $\frac{1}{2}\text{metro de entrada}$  e a partir desses objetos *Model*, tem-se acesso a seus atributos e seus mĩ $\frac{1}{2}\text{todos}$ .

$\text{iĩl}_{\frac{1}{2}}$  atravĩ $\frac{1}{2}\text{s}$  dos atributos de um objeto *Model* que se diã $\frac{1}{2}\text{o}$  o acesso programĩ $\frac{1}{2}\text{vel}$  aos dados dos modelos de reservatĩ $\frac{1}{2}\text{rio}$ . Esses dados  $\text{nĩl}_{\frac{1}{2}}\text{o}$  podem ser alterados no *script*,  $\text{iĩl}_{\frac{1}{2}}$  possĩ $\frac{1}{2}\text{vel}$  apenas consulta-los e usar os valores retornados. Assim  $\text{iĩl}_{\frac{1}{2}}$  possĩ $\frac{1}{2}\text{vel}$  acessar dados de geometria, como o  $\text{nĩl}_{\frac{1}{2}}\text{mero de cĩl}_{\frac{1}{2}}\text{lulas nas direiã}_{\frac{1}{2}}\text{i}_{\frac{1}{2}}\text{es I, J ou K, acessar dados relacionados a timesteps, acessar dados relacionados aos demais objetos do modelo, como o nĩl}_{\frac{1}{2}}\text{mero de poil}_{\frac{1}{2}}\text{os ou de regiã}_{\frac{1}{2}}\text{es, acessar os prĩl}_{\frac{1}{2}}\text{prios objetos e tambĩl}_{\frac{1}{2}}\text{m a todas as propriedades do modelo.}$

Como dito acima, o objeto *Model* também provê um conjunto de métodos que possibilitam o acesso à estrutura do reservatório, a criação de novos modelos, criação de novos vetores de propriedade, verificar a existência de timesteps e propriedades e associar os novos dados vetoriais criados durante o *script* aos modelos.

## 2.4

### Vetores de dados de propriedades

Como visto anteriormente, os valores das propriedades são representados em vetores de dados. Vetores de propriedade de mapa, por exemplo, têm dimensão 4 ( $n_i \times n_j \times n_k \times n_{\text{mero de timesteps de mapa}}$ ) e podem ser usados em operações com outros vetores de mapa de mesmas dimensões. É permitida a criação de “fatias” (*slices*) que fazem referência a uma parte de um vetor 4D, ou seja, pode-se representar uma propriedade de uma camada de um reservatório através de uma referência ao vetor 4D, onde a terceira dimensão é fixada.

As dimensões de uma fatia que representa uma camada são:  $n_i \times n_j \times 1 \times n_{\text{mero de timesteps de mapa}}$ . Conceitualmente, uma camada é um vetor 3D,  $j$  que o índice  $k$  varia. No entanto, internamente, ela é representada por um vetor 4D, como se os valores associados à camada em questão fossem replicados para as demais camadas do grid. Desta forma, pode-se fazer operações entre um grid e uma camada específica. Generalizando, pode-se fazer operações entre vetores e fatias de vetores de mesmas dimensões. Também é possível associar uma fatia a um modelo, para ser visualizada através da interface gráfica. Neste caso, os valores são replicados ao longo das dimensões fixadas.

Conceitualmente, uma propriedade de campo é um vetor 1D ( $n_{\text{mero de timesteps}}$ ), uma propriedade de região é um vetor 2D ( $n_{\text{mero de regiões}} \times n_{\text{mero de timesteps}}$ ), uma propriedade de grupo é um vetor 2D ( $n_{\text{mero de grupos}} \times n_{\text{mero de timesteps}}$ ), uma propriedade de poço é um vetor 2D ( $n_{\text{mero de poços}} \times n_{\text{mero de timesteps}}$ ), e uma propriedade de completamento é um vetor 3D ( $n_{\text{mero de poços}} \times n_{\text{mero de completamentos}} \times n_{\text{mero de timesteps}}$ ).

Em todas as operações que envolvem dois vetores de dados como operandos, a operação só é possível se os timesteps associados aos vetores forem compatíveis. O primeiro operando é usado com referência na operação. Para cada timestep existente no primeiro operando, deve haver um timestep correspondente no segundo operando. A correspondência entre timesteps é feita buscando-se a ocorrência entre datas próximas,

dentro de uma determinada tolerância, que pode ser configurada pelo programador. Se a operação entre dois vetores puder ser realizada, o sistema reporta uma mensagem de erro. De posse de um vetor de dados, pode-se fazer diversas operações e invocar diversos métodos associados. É possível realizar as operações aritméticas <sup>2</sup> de soma, subtração, multiplicação, divisão e potenciação. Essas operações resultam em um novo vetor de dados, de mesmas dimensões, onde cada valor representa a operação aplicada aos valores associados aos vetores fornecidos. (Tecg06)

Estão disponíveis também operadores relacionais e lógicos e alguns métodos de manipulação de valores.

<sup>2</sup>Também é possível realizar operações aritméticas entre vetores de propriedades e valores escalares.

### 3

## Calculadora de propriedades

Como visto anteriormente, um recurso interessante do sistema de *script* é a possibilidade de usá-lo como uma espécie de máquina de calcular que opera sobre os dados de modelos de reservatórios. Sendo assim, a primeira solução encontrada para resolver o problema apresentado na introdução, oferecer parte dos recursos do sistema de *script* para o usuário, foi implementar justamente uma calculadora de propriedades onde o resultado das operações fossem propriedades que pudessem ser posteriormente visualizadas pelo usuário. Sendo assim, a ferramenta proposta deveria ter uma interface gráfica na forma de uma calculadora e gerar um código compatível com o sistema de *script* e que obviamente resultasse na propriedade desejada pelo usuário.

### 3.1

#### Sintaxe das expressões

Inicialmente a calculadora foi pensada para funcionar apenas para propriedades de mapa. As propriedades de mapa são os valores associados às células dos modelos, por isso o acesso aos dados dessas propriedades é sempre independente dos demais objetos vistos na seção 2.2. Portanto, a sintaxe das expressões deveria ser a sintaxe comum de expressões aritméticas levemente modificada para atender aos requisitos:

1. Possibilitar o uso não somente de propriedades mas também de “fatias” de propriedades com o timestep fixado.
2. Permitir o uso de funções, para incluir outras funcionalidades do sistema de *script*.
3. Ser compatível com **Lua**, para facilitar a implementação.

As “fatias” foram incluídas da seguinte forma: uma fatia é o nome de propriedade (identificador) seguido de uma data entre aspas, como por exemplo: SO“10/10/2010”. Em **Lua**, essa expressão é encarada como uma chamada de função.



Posteriormente, foram incluídas na calculadora também propriedades de campo, região, grupo e poço. No caso de tais propriedades, abriu-se a possibilidade de incluir também os respectivos elementos nas expressões. Dessa maneira, a fatia da propriedade associada a um elemento (grupo, região ou poço) seria referenciada da seguinte maneira: nome do elemento seguido do nome da propriedade entre colchetes, como por exemplo: WELL[BHP]. Isso indicaria a fatia da propriedade BHP associada ao poço WELL. Em **Lua**, essa expressão é encarada como uma indexação de uma tabela.

### 3.2

#### Implementação

A calculadora foi implementada na linguagem de programação **Lua** e a interface gráfica foi implementada utilizando o toolkit para construção de interfaces gráficas IUP. (IUP) A implementação foi dividida em módulos: Calculator.lua, que implementa a interface gráfica e controla a entrada de dados do usuário, CalcComp.lua, responsável pela geração de código e validação das expressões e por último CalcLib.lua, que implementa algumas funções para serem incluídas no sistema de *script* que garantem o funcionamento do código gerado.

#### 3.2.1

##### Módulo Calculator.lua



Figura 3.1: Interface da Calculadora

A interface é composta pelos seguintes campos:

- Um campo de texto onde é possível entrar o nome da nova propriedade, caso não seja especificado um nome, é usado a própria expressão como nome.

- Lista que indica o tipo de propriedade que será calculada, atualmente os tipos suportados são propriedade de mapa, campo, região e ponto.
- Um toggle e a lista de elementos, que é preenchida de acordo com o valor da lista de tipos, se o tipo for mapa, esses dois controles são habilitados, pois se aplicam a propriedades destes tipos.
- Lista de propriedades, que é preenchida de acordo com o valor da lista de tipos.
- Um toggle e lista de timesteps, que é preenchida de acordo com o valor da lista de tipos.
- Botão de inserir propriedade.
- “Teclado” formado por botões que disponibilizam os números, os operadores e algumas funções para o usuário.
- Um campo de texto onde pode-se visualizar a expressão.
- Botões de calcular e limpar expressão.

A interface funciona da seguinte forma: o usuário pode entrar números e operadores ou movimentar o cursor livremente através de seu teclado ou clicando no teclado virtual da interface. Já a entrada de variáveis (propriedades) e funções são feitas através da interface gráfica. O objetivo disso é evitar a geração de dígitos inconsistentes.

Para se inserir funções basta-se clicar no respectivo botão da função no teclado virtual para que a função seja inserida na posição corrente do cursor.

Para se inserir propriedades é preciso clicar no botão de inserir ao lado da lista de timesteps. O valor inserido depende dos valores das listas e dos toggles. Se nenhum dos toggles estiver habilitado, será inserido apenas o nome da propriedade selecionada na lista de propriedades. Se o toggle da lista de elementos estiver habilitado, será inserido a propriedade selecionada na lista de propriedades indexando o elemento selecionado na lista de elementos (exemplo: WELL[BHP]). Caso o toggle de timestep esteja habilitado, será adicionado ao fim a data selecionada na lista de timesteps (exemplo: SO“10/10/2010”).

Para que não haja casos de entrada de números, operadores, funções ou propriedades em posições indesejadas (como por exemplo no meio do nome de uma propriedade) foram implementadas algumas funções que controlam o movimento e posição do cursor, evitando cenários desse tipo.

## 3.2.2

**Módulo CalcComp.lua**

Quando se pensa em gerar código de código para um problema como este, a primeira abordagem que nos vem à cabeça é compilar as expressões e gerar o código desejado a partir delas. Por fim, compilar códigos, por mais simples que sejam as expressões encontradas nesse caso, não é uma tarefa trivial. O desenvolvimento de um mini-compilador envolveria a implementação de um parser, a elaboração de uma gramática, o uso de técnicas como criação de árvores de sintaxe e etc. Por isso, a abordagem que foi seguida vai praticamente na direção oposta. Em vez de compilar as expressões e gerar um código de *script* a partir delas, o que fizemos foi adaptar o ambiente do sistema de *script* para que as próprias expressões pudessem ser executadas dentro desse ambiente e gerar o resultado esperado. Por isso que era imprescindível que as expressões fossem compatíveis com **Lua**.

As adaptações feitas da seguinte maneira: é desempenhada uma iteração sobre os identificadores presentes na expressão (nomes de propriedades ou dos demais elementos, como nomes de pontos), é verificado se o identificador se refere a uma propriedade ou a um elemento e então é gerado um código que inicializa uma variável local de mesmo nome da propriedade ou elemento com uma chamada de função que retorna um objeto especial que possibilite a execução do código da expressão. Essa função recebe como parâmetro uma referência para o objeto que representa o modelo de reservatório, um *string* indicando o tipo de dado sendo trabalhado (propriedade de mapa, de pontos e etc) e um *string* contendo o nome da propriedade ou objeto. Além disso, também é gerado o código que armazena o resultado da expressão em uma variável temporária e o da chamada de função que associa o vetor de dados resultante ao modelo para ser posteriormente visualizado.

Segue abaixo o exemplo de um código gerado pelo módulo para a expressão  $OIL1[QO] + QW \text{ "10/07/1981"}$ :

```
calc.GetMacroEnv()
local OIL1 = calc.CreateElem(rf,"well","OIL1")
local QO = calc.CreateProp(rf,"well","QO")
local QW = calc.CreateProp(rf,"well","QW")
local newprop = OIL1[QO] + QW"10/07/1981"
calc.AttachProp(rf,temp,'newprop','well')
```

Nesse caso, QO e QW são propriedades de pontos, OIL1 é o nome de um ponto e rf representa um objeto do tipo *Model*.

A implementação dessas funções e dos objetos retornados por elas será discutida na próxima seção.

O módulo também disponibiliza uma função que testa a expressão com o intuito de prevenir erros de sintaxe e que funções sejam chamadas sem que seus argumentos tenham sido especificados.

### 3.2.3

#### Módulo CalcLib.lua

O módulo CalcLib.lua é responsável por implementar as funções presentes no código gerado pelo módulo CalcComp.lua. Essas funções servem para adaptar o ambiente do sistema de *script* para que as prioridades expressões possam ser executadas dentro desse ambiente e gerar o resultado esperado. As funções são armazenadas em uma tabela que é inserida no ambiente de *script* (tabela calc). O principal desafio de implementar esse módulo foi desenvolver objetos que se comportassem ora como operandos de expressões aritméticas e ora como funções (no caso de propriedades com o timestep especificado, que é interpretado em **Lua** como uma chamada de função). Uma limitação que também teve que ser superada foi a de que, em operações aritméticas entre vetores de dados e escalares no sistema de *script*, os escalares devem sempre ser o segundo operando.

Para solucionar esses problemas, foi utilizado o recurso de metamecanismos presente em **Lua**, que nos permite especificar como determinados objetos devem se comportar em algumas situações. Isso é disponibilizado por meio da implementação de *metatables*, que são tabelas que definem através de seus *metamétodos* como os objetos associados a ela devem se comportar.

Os objetos retornados pela função *CreateProp*, que representam as propriedades nas expressões, são tabelas que possuem campos que especificam o nome da propriedade, o tipo da propriedade e se o valor utilizado nos cálculos será a propriedade associada a um determinado elemento ou o vetor de dados inteiro. Seu comportamento (determinado por sua *metatable*) é o seguinte:

- O *metamétodo* de todas as operações binárias ajusta a ordem dos termos, armazena o resultado da operação em um objeto intermediário e o retorna.
- O *metamétodo* responsável pela situação em que o objeto chamado como uma função, armazena a “fatia” da propriedade referente

ao timestep passado como parâmetro em um objeto intermediário e o retorna.

- O metamodelo todo responsável pelo tratamento de índices e o responsável por buscar os vetores de dados que serão usados nos cálculos, por isso os demais metamodelos acessam esses valores apenas indexando a própria tabela.

O objeto intermediário, que apenas uma tabela que armazena o resultado das operações, se mostrou necessário para resolver a limitação da ordem dos termos em operações envolvendo escalares citada acima. Ele associado à mesma *metatable* do objeto que representa as propriedades e por isso sempre ajusta a ordem dos termos nas operações.

Os objetos que representam nas expressões os elementos (pois os, registo e etc), são criados pela função *CreateElem*. Eles são tabelas contendo campos com o nome do elemento, o seu tipo e uma referência para o modelo que pertence. A *metatable* que é associada a eles contém apenas o metamodelo todo responsável pelo tratamento de índices. Ao ser indexada por um objeto que representa uma propriedade, esse metamodelo todo extrai o nome da propriedade e cria um novo objeto para mesma propriedade (chamando a função *CreateProp*) por isso especificando que a propriedade está associada agora ao elemento. Portanto, o vetor de dados utilizado nos cálculos será referente a “fatia” da propriedade referente a esse elemento.

A função *AttachProp* associa o resultado da expressão ao modelo.

Este módulo também implementa as funções disponíveis na interface da calculadora:

- **max**: retorna o maior valor encontrado dentre os vetores de dados passados como parâmetros.
- **min**: retorna o menor valor encontrado dentre os vetores de dados passados como parâmetros.
- **sum**: retorna o somatório dos valores do vetor de dados.
- **abs**: retorna um novo vetor de dados de mesma dimensão do vetor passado como parâmetro com os valores sendo igual aos respectivos valores absolutos do vetor original.
- **sqrt**: retorna um novo vetor de dados de mesma dimensão do vetor passado como parâmetro com os valores sendo igual às respectivas raízes quadradas dos valores do vetor original.

- **delta**: retorna um novo vetor de dados de mesma dimensão  $\frac{1}{2}$  do vetor passado como parâmetro com os valores sendo igual diferença  $\frac{1}{2}$  entre posições  $\frac{1}{2}$  vizinhas.

## 4

### Linguagens de Programação Visual

Ultimamente o uso de ferramentas de programação visual vem crescendo cada vez mais, principalmente em aplicações voltadas para áreas da engenharia como *LabVIEW*, *Simulink*, *CircuitMaker* entre outros. Mesmo tendo apresentado relativa dificuldade na resolução de problemas de grande porte o uso destas ferramentas em sistemas que apresentam um domínio específico, como as citadas acima, tem se mostrado bastante interessante pois é possível manter-se dentro do estilo de comunicação do domínio utilizando artefatos visuais que reflitam necessidades particulares, diagramas e o vocabulário específico do domínio sem nunca precisar deixar esse estilo de comunicação. Além disso, linguagens de programação visual costumam ser mais intuitivas e de mais fácil aprendizado do que linguagens de programação convencionais.

Pelos motivos citados acima, e pela calculadora de propriedades oferecer relativamente poucos recursos ao usuário, se mostrou interessante um estudo dessas ferramentas de programação visual e investigar se esse tipo de recurso solucionaria o nosso problema.

Dentre os sistemas que utilizam programação visual, o que tem se destacado mais é o *LabVIEW* (*Laboratory Virtual Instrument Engineering Workbench*) [WhNF06], que é um ambiente de programação visual muito usado nos campos de aquisição de dados, controle de instrumentos e automação industrial. Por ser um instrumento amplamente usado por engenheiros em diversas áreas, e como esses são os usuários finais do **Geresim**, essa ferramenta se mostra uma interessante fonte de pesquisa e um possível modelo a ser seguido, já que é possível que parte dos usuários finais tenha uma certa experiência com o *LabVIEW*.

A linguagem de programação do *LabVIEW*, chamada de G, é formada por elementos visuais e textuais. Os elementos visuais são basicamente pequenas caixas (que podem representar valores, operadores e chamadas de funções) e linhas que as conectam. Elementos compostos como funções condicionais e de repetição (loop) são representados por grandes caixas retangulares cercando o corpo da composição.

A linguagem segue o paradigma de fluxo de dados. O princípio central do paradigma é definir a computação como um conjunto de dependências de dados. Um programa escrito nesse paradigma define um grafo de nós operadores (funções) unidos por arcos conectores. Os arcos são as dependências de dados. Assim o paradigma de fluxo de dados expressa a computação como uma série de transformações nos valores dos dados. Não há conceitos como variáveis, ponteiros e atribuições, o foco está em quais funções serão aplicadas nos valores para produzir o retorno esperado. Outra característica desse paradigma é o paralelismo. A ordem de execução é baseada somente nas dependências de dados, o que é relativamente similar ao design de circuitos eletrônicos.

Conceitualmente os programas desenvolvidos no *LabVIEW* são compostos de funções, que recebem o nome de instrumentos virtuais (VIs). O código de um VI consiste basicamente de três elementos: terminais, nós e fios conectores. Os elementos terminais são os valores manipulados, podendo ser valores de entrada, saída ou valores constantes. Esses elementos são representados por pequenas caixas coloridas com a cor do tipo do terminal (por exemplo, laranja para números). Os nós são operadores, chamados de funções e nós estruturais (if-then-else, case, while etc). Operadores e chamadas de funções são representados por ícones com etiquetas ou imagens, e os nós estruturais são representados por grandes caixas retangulares que delimitam a composição que deve ser executada dentro da estrutura. Os fios conectores são usados para ligar terminais e nós formando expressões.



## 5

### Implementação de uma VPL

Após o estudo de VPLs e algumas experiências utilizando o *LabVIEW* (na matéria Automática Industrial, cursada nesse período), foi iniciada a implementação de uma VPL bastante simplificada que segue o paradigma de fluxo de dados. Infelizmente não houve tempo suficiente para desenvolver mais a ferramenta e por isso a implementação encontra-se em estado inicial.

Ao contrário do que foi feito na calculadora de propriedades, inicialmente seria implementado algo mais genérico para depois ser especializado para o ambiente de pós-processamento do **Geresim**. A princípio, a linguagem seria composta apenas de blocos e fios conectores que denotariam as dependências de dados entre os blocos. Um bloco poderia ter mais de um dado de entrada e mais de um dado de saída. Sua compiladora geraria um código **Lua**.

No código gerado, cada bloco seria representado por uma tabela. Essa tabela armazenaria a função associada ao bloco e uma lista de objetos parâmetros, que seriam tabelas armazenando uma referência a outro bloco e o índice do dado de saída deste outro bloco. Todos os blocos estariam associados a uma *metatable* cujo metamétodo de indexação executaria a função do bloco gerando assim os dados de saída. Portanto, antes de poder executar a função associada a este bloco, seria preciso obter todos os parâmetros. E isso seria feito iterando a lista de parâmetros e indexando os respectivos blocos com os respectivos índices. Caso os dados de saída desses demais blocos não estejam disponíveis, o metamétodo de indexação agiria então nesses blocos. Logo, recursivamente toda a cadeia de blocos seria executada. Dessa maneira, a compiladora seria apenas uma iteração sobre os fios conectores e a partir deles se geraria a lista de parâmetros de cada bloco.

A partir da base descrita acima, já seria possível implementar um bloco que desempenhe a função de um operador condicional e talvez seja possível implementar um bloco de iteração.

No atual estágio da implementação já é possível manipular os blo-

cos com o mouse e conecta-los com os fios conectores. É possível também manipular blocos já conectados, fazendo com que os fios conectores se redesenhem se ajustando a nova disposição dos blocos. Essa implementação foi desenvolvida em **C++**, utilizando a API gráfica *OpenGL*. (OpenGL) A parte **Lua** ainda não foi desenvolvida.

## 6

### Considerações finais e projetos futuros

Neste trabalho foi apresentado um sistema de *script* de uma aplicação voltada para o pós-processamento de simulações e de reservatórios de petróleo (**Geresim**) e a proposta e implementação de uma ferramenta que disponibiliza parte dos recursos obtidos a partir dos *scripts* usuais dos programadores. Essa ferramenta é uma calculadora de propriedades que permite que o usuário gere novas propriedades a partir de simples expressões aritméticas desenvolvidas em uma interface gráfica similar a uma calculadora comum. A ferramenta ainda foi utilizada por usuários finais, mas está integrada ao sistema e após ser testada e aprovada por outros membros da equipe de desenvolvimento da aplicação criou uma boa expectativa em relação ao cumprimento de seu objetivo. A ferramenta estará disponível na próxima versão do programa, que será lançada em breve.

Adicionalmente, foram feitos estudos acerca de linguagens de programação visual. Dentre os sistemas que utilizam esse recurso, o que se mostrou mais interessante foi o *LabVIEW*, pois é amplamente usado por usuários que apresentam um perfil similar aos usuários do **Geresim**. Após esse estudo foi iniciada a implementação de uma VPL bastante simplificada que segue o paradigma de fluxo de dados. Infelizmente não houve tempo suficiente para desenvolver mais a ferramenta e por isso sua finalização passa a ser um projeto para o futuro.

Outro projeto futuro é aumentar o escopo de atuação da calculadora incluindo as propriedades de completimento e incluindo a opção de se trabalhar com camadas e seções das propriedades de mapa.

## Referências Bibliográficas

[IUP] <http://www.tecgraf.puc-rio.br/iup/>. 3.2

[Ieru03] IERUSALIMSKY, R.. **Programming in Lua**. Lua.org, Rio de Janeiro, 2003. 1, 2.1

[Lua] <http://www.lua.org>. 2.1

[OpenGL] MASON WOO, JACKIE NEIDER, T. D. D. S.. **OpenGL Programming Guide**. Addison-Wesley, Reading, Massachusetts, 6 edition, 2008. 5

[Tecg06] TECGRAF. **Manual do geresim**, 2006. 1, 2.4

## A

### Exemplos práticos

#### A.1

##### Cálculo da razão gas-óleo (RGO)

Expressão:  $QG / QO$

Código gerado:

```
calc.GetMacroEnv()  
local QG = calc.CreateProp(rf,"well","QG")  
local QO = calc.CreateProp(rf,"well","QO")  
local RGO = QG /QO  
calc.AttachProp(rf,RGO,'RGO','well')
```

#### A.2

##### Cálculo sedimentos básicos e água (BSW)

Expressão:  $QW / (QW + QO)$

Código gerado:

```
calc.GetMacroEnv()  
local QW = calc.CreateProp(rf,"well","QW")  
local QO = calc.CreateProp(rf,"well","QO")  
local BSW = QW / (QW + QO )  
calc.AttachProp(rf,BSW,'BSW','well')
```

#### A.3

##### Cálculo do volume de óleo em condições de reservatório

Expressão:  $H * PHI * SO * DX * DY$

Código gerado:

```
calc.GetMacroEnv()  
local H = calc.CreateProp(rf,"map","H")  
local PHI = calc.CreateProp(rf,"map","PHI")  
local SO = calc.CreateProp(rf,"map","SO")
```

2-PHASE (OIL-WATER) DEPLETION MXDRM001 4 WELLS DEPLETION OPERATION COMPARE W1  
 Propriedade: H \* PHI \* SO \* DX \* DY Data: 01/01/1995 Time Step: 0 (0 dias) Unidade: Adimensional

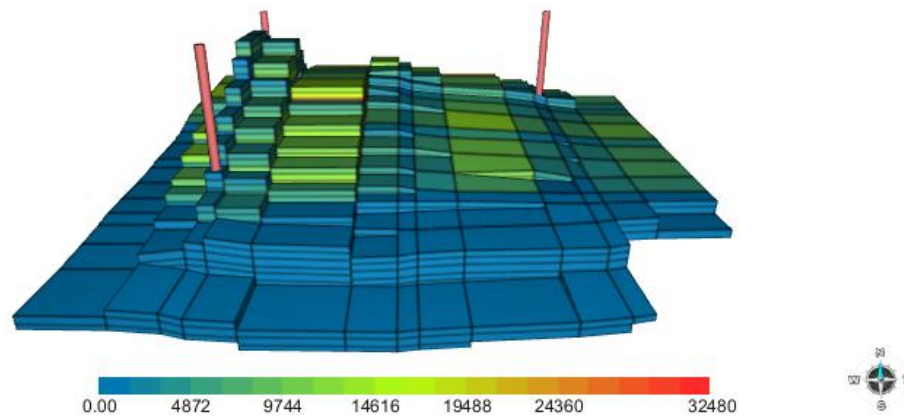


Figura A.1: Exemplo de visualização

```
local DX = calc.CreateProp(rf,"map","DX")
local DY = calc.CreateProp(rf,"map","DY")
local temp = H * PHI * SO * DX * DY
calc.AttachProp(rf,temp,'H * PHI * SO * DX * DY ', 'map')
```