UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE TECNOLOGIA E GEOCIÊNCIAS
DEPARTAMENTO DE ENGENHARIA CIVIL E AMBIENTAL
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA CIVIL

ANDRESA DORNELAS DE CASTRO

IMPLEMENTAÇÃO DO MODELO DE ORDEM REDUZIDA TPWL PARA SIMULAÇÃO DE RESERVATÓRIOS DE PETRÓLEO NA PLATAFORMA MRST

Recife
2020

ANDRESA DORNELAS DE CASTRO

IMPLEMENTAÇÃO DO MODELO DE ORDEM REDUZIDA TPWL PARA SIMULAÇÃO DE RESERVATÓRIOS DE PETRÓLEO NA PLATAFORMA MRST

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Engenharia Civil da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Mestre em Engenheira Civil.

Área de concentração: Simulação e Gerenciamento em Reservatórios de Petróleo

Orientador: Prof. Dr. Bernardo Horowitz

Recife

2020

Catalogação na fonte
Bibliotecário Gabriel Luz, CRB-4 / 2222

C355i Castro, Andresa Dornelas de.
Implementação do modelo de ordem reduzida TPWL para simulação de reservatórios de petróleo na plataforma MRST / Andresa Dornelas de Castro – Recife, 2020.
212 f.: figs., quads., tabs., símbolos.

Orientador: Prof. Dr. Bernardo Horowitz.
Dissertação (Mestrado) – Universidade Federal de Pernambuco. CTG. Programa de Pós-Graduação em Engenharia Civil, 2020.
Inclui referências e apêndices.

1. Engenharia Civil. 2. Trajectory Piecewise Linearization. 3. Matlab Reservoir Simulation Toolbox. 4. Simulação de reservatórios. 5. Diferenciação automática. 6. Modelo black-oil. I. Horowitz, Bernardo (Orientador). II. Título.

UFPE

624 CDD (22. ed.) BCTG / 2020-150

ANDRESA DORNELAS DE CASTRO

IMPLEMENTAÇÃO DO MODELO DE ORDEM REDUZIDA TPWL PARA SIMULAÇÃO DE RESERVATÓRIOS DE PETRÓLEO NA PLATAFORMA MRST

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Engenharia Civil da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Mestre em Engenheira Civil.

Aprovada em: 31/ 03 / 2020.

BANCA EXAMINADORA

_____

Prof. Dr. Bernardo Horowitz (Orientador)
Universidade Federal de Pernambuco

_____

Prof. Dr. Ramiro Brito Willmersdorf (Examinador Interno)
Universidade Federal de Pernambuco

_____

Prof. Dr. Leonardo Correia de Oliveira (Examinador Externo)
Universidade Federal de Pernambuco

_____

Prof. Dr. Juan Alberto Rojas Tueros (Examinador Externo)

Universidade Federal de Pernambuco

Dedico esse trabalho a Deus; a minha família, que sempre me incentivou e apoiou ao longo dessa trajetória de estudo e aprendizagem; e a todos que contribuíram para a sua elaboração.

AGRADECIMENTOS

Muitas pessoas apoiaram e contribuíram para a realização deste trabalho. Primeiramente, agradeço a Deus pelas bençãos concedidas e por ter me proporcionado força e saúde para superar as dificuldades impostas ao longo dessa jornada.

Agradeço à Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) pelo suporte financeiro para o desenvolvimento dessa pesquisa.

Agradeço ao meu orientador Professor Bernardo Horowitz pela contribuição e suporte ao trabalho.

A todos os docentes e colegas de curso do Programa de Pós-graduação em Engenharia Civil pelo apoio e parceria na trajetória dessa conquista. Destaco especialmente aqueles que forneceram apoio direto na pesquisa, Alexandre de Souza, Jonathan Teixeira, e Juan Alberto; assim como, no meu desenvolvimento na área de simulação e gerenciamento de reservatórios de petróleo: Prof. Darlan Karlo, Dilayne Oliveira, entre outros.

À Universidade Federal de Pernambuco pela oferta de conhecimento que desde a graduação possibilitou a minha aprendizagem e crescimento pessoal.

Além das pessoas da universidade agradeço tenho muito a agradecer a meus familiares.

Aos meus pais, Sandra e Armencídio, que com bastante amor e carinho, sempre estiveram dispostos a fornecer os conselhos e ensinamentos que fundamentaram o meu desenvolvimento pessoal e profissional, bem como, pelo apoio incondicional que dispuseram para a realização dos meus sonhos.

Às minhas irmãs, Alane e Ana Clara, que sempre estiveram presentes ao meu lado, compartilhando os momentos de alegria e descontração e, auxiliando na superação de momentos difíceis e conturbados. À Abmael Júnior pela atenção e companheirismo nessa trajetória.

Enfim, agradeço a todos que direta ou indiretamente fizeram parte dessa etapa decisiva em minha vida.

RESUMO

Esta dissertação tem como objetivo apresentar a implementação da técnica de modelo substituto denominada *Trajectory Piecewise Linearization* (TPWL) para simulação de reservatórios, utilizando como simulador o *Matlab Reservoir Simulation Toolbox* - MRST. O TPWL é uma técnica que reduz a complexidade do problema através da linearização de suas equações governantes em torno de estados convergidos e armazenados durante uma simulação de treinamento. O simulador utilizado foi desenvolvido pela SINTEF Digital e consiste em uma ferramenta de livre acesso ao código fonte, compostos por uma variedade métodos computacionais empregados na solução de diversos problemas relacionados ao escoamento superficial e subterrâneo. O MRST tem se tornado cada vez mais popular em instituições acadêmicas ao redor do mundo, atraindo interesse de diversos pesquisadores. Nesse trabalho, aborda-se uma descrição detalhada de como utilizar essa ferramenta para a simulação de reservatórios, enfatizando especialmente o modelo *black-oil* com diferenciação automática para o cálculo das matrizes jacobianas requeridas na solução de problemas não lineares do tipo Newton. Além disso, apresenta-se detalhes sobre como utilizar esse simulador para a exportação e armazenamento dos dados necessários para a implementação do TPWL. Essa metodologia é denominada semi-intrusiva, pois requer além do conhecimento do simulador, eventuais alterações no código que não afetam as soluções das equações. Por fim, estuda-se a aplicação da técnica TPWL em modelos *benchmarks*, ratificando que esse método é acurado na vizinhança da trajetória de treinamento.

Palavras-chave: *Trajectory Piecewise Linearization*. *Matlab Reservoir Simulation Toolbox*. Simulação de reservatórios. Diferenciação automática. Modelo *black-oil*.

ABSTRACT

The goal of this dissertation is to present the implementation of the substitute model technique called Trajectory Piecewise Linearization (TPWL) for reservoir simulation, using the Matlab Reservoir Simulation Toolbox - MRST as a simulator. TPWL is a technique that reduces the complexity of the problem by linearizing its governing equations around converged and stored states during a training simulation. The simulator used was developed by SINTEF Digital and consists of a tool with free access to the source code, composed of a variety of computational methods used to solve various problems related to surface and underground flow. MRST has become increasingly popular in academic institutions around the world, attracting interest from several researchers. In this work, a detailed description of how to use this tool for the simulation of reservoirs is discussed, especially emphasizing the black-oil model with automatic differentiation for the calculation of Jacobian matrices required in the solution of Newton non-linear problems. In addition, details are presented on how to use the simulator for the export and storage of the data necessary for the implementation of the TPWL. This methodology is called semi-intrusive, as it requires, besides the knowledge of the simulator, any changes in the code that do not affect the solutions of the equations. Finally, the application of the TPWL technique in benchmark models is studied, confirming that this method is accurate in the vicinity of the training trajectory.

Keywords: Trajectory Piecewise Linearization. Matlab Reservoir Simulation Toolbox. Reservoir simulation. Automatic differentiation. Black-oil Model.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE TABELAS

LISTA DE ALGORITMOS

LISTA DE SÍMBOLOS

- **Siglas**

AD	– *Automatic Differentiation;*
BHPs	– Pressões do fundo de poço;
IPR	– *Inflow-performance relation;*
MRST	– *Matlab Reservoir Simulation Toolbox;*
MRST AD-OO	– *MATLAB Reservoir Simulation Toolbox - Automatic Differentiation - Object Oriented;*
MRST-core	– Módulo central do MRST;
POO	– Programação Orientada a Objeto;
POD	– *Proper Orthogonal Decomposition ;*
SPE10	– *The 10th SPE Comparative Solution Project;*
SI	– Sistema internacional de Unidades;
TPWL	– *Trajectory Piecewise Linearization;*
TPFA	– *Two Point Flux Approximation*
VPL	– Valor Presente Líquido, ou simplesmente;

- **Símbolos**

ϕ	– porosidade, razão entre o volume de vazios e o volume total de uma rocha;
α	– fases do modelo black-oil, podendo ser água (w), óleo (o) e gás (g).
λ_α	– a mobilidade da fase, razão entre permeabilidade relativa e a viscosidade;
$\lambda_{\alpha,i}$	– mobilidade da fase α na conectividade i;
μ_α	– viscosidade para fase α;
δ	– diferença do vetor de estado a cada iteração;
ρ_α	– massa específica ou a "densidade" da fase α;
Ω_i	– representação de uma célula
$\Gamma_{i,k}$	– meia-face;

$\Delta x, \Delta y, \Delta z$ – dimensões de cada elemento do bloco nas direções cartesianas;

$A=\left[a_{ij}\right]$ – matriz esparsa proveniente do esquema TPFA.

A – termo de acumulação

Δt – passo de tempo.

a – expoente de Corey para a curva de permeabilidade relativa do óleo;

b – expoente de Corey para a curva de permeabilidade relativa de água.

b_α – inverso do fator volume de formação para fase α ;

B_α – fator volume-formação para fase α ;

E^m – erro médio de cada poço ;

E – erro médio total;

F – termo de fluxo;

g – módulo da aceleração gravitacional;

g – o vetor do resíduo que se procura zerar;

G – estrutura da malha.

h – comprimento do poço.

H_{wi} – a pressão hidrostática na conectividade i ;

i,j,k – ordenamento lógico da malha cartesiana;

J – matriz jacobiana das derivadas parciais;

K – tensor de permeabilidade absoluta da rocha;

kr_α – permeabilidade relativa para fase α ;

$k_{r\alpha}^0$ – ponto terminal da permeabilidade relativa do óleo para fase α ;

K_e – permeabilidade efetiva (para meio anisotrópico);

n_c – número de elementos da malha cartesiana;

n_f – número de faces da malha cartesiana;

n_{fs} – número total de faces com sobreposição;

n_n – número global de nós na malha (sem sobreposição).

n_{ns} – número total de nós com sobreposição;

n , $n+1$ – passo de tempo atual *(timestep*) e próximo passo de tempo;

p_α – pressão da para fase α ;

p_i – a pressão no bloco perfurado;

p_{bh} – a pressão de fundo de poço;

p_c – pressão capilar;

p_{wh} – pressão no topo do reservatório;

q_α – termo fonte volumétrico para fase $\alpha = w,o,g$;

Q – termo fonte/sumidouro

r_s – razão de solubilidade do gás no óleo;

r_v – solubilidade de óleo no gás;

r_e – raio equivalente (para malhas com elementos retangulares, $\Delta x \neq \Delta y$);

r_w – raio do poço;

S – *skin fator;*

s_α – saturação da para fase α ;

S_{or} – saturação de óleo residual;

S_{wc} – saturação de água conata;

T_α – indica transmissibilidade , que relaciona o fluxo na fase α com a diferença de pressão;

$T_{i,k}$ – transmissibilidade unilateral;

u – vetor de controles dos poços;

v_α – velocidade de Darcy para fase α ;

V – volume do bloco.

x – vetor de estado que contém as incógnitas primárias (p_o e s_w).

WI_i – índice de injetividade/produtividade do poço (notação MRST);

z – componente do vetor deslocamento orientado para baixo;

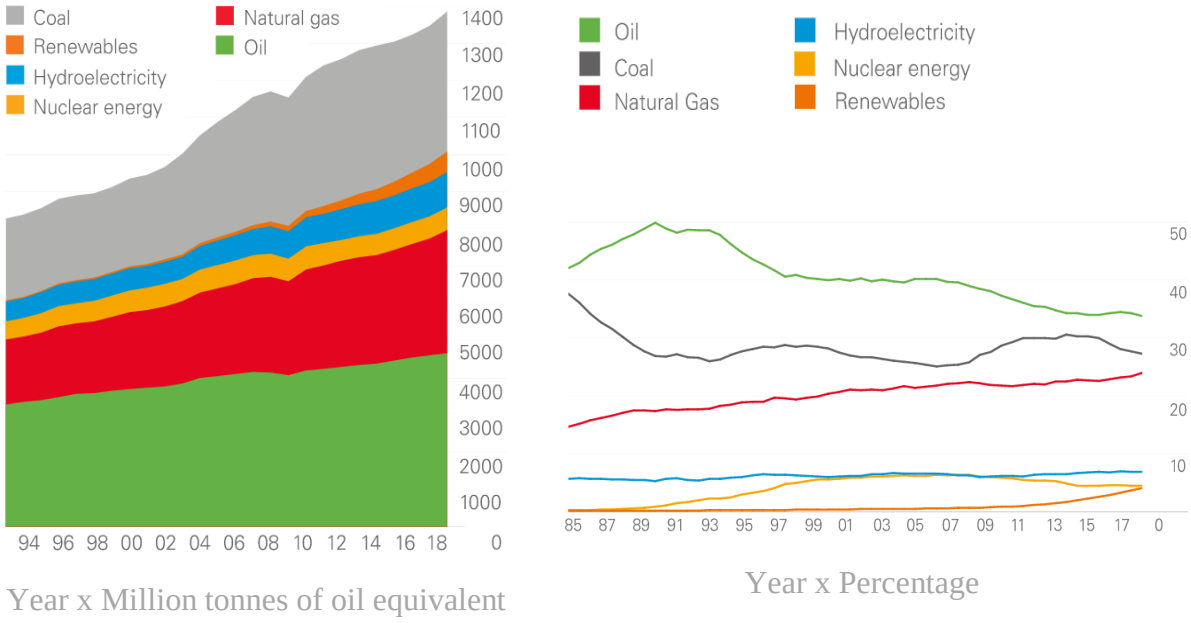
z^i – mapa de saturações do estado x^i

SUMÁRIO

1 INTRODUÇÃO

Com o intuito de consolidar a importância do desenvolvimento do presente estudo, será introduzido o panorama mundial a respeito do petróleo. Sabe-se que atualmente a demanda global por energia é grande e continuará crescendo. De acordo com a Figura 1, é possível perceber que grande parte dessa energia é fornecida por combustíveis fósseis, ou seja, petróleo, carvão e gás. O petróleo continua sendo o combustível mais utilizado. O carvão é o segundo maior combustível, mas perdeu participação em 2018, representando 27%, seu nível mais baixo em 15 anos.

Figura 1 – Consumo mundial de energia primária



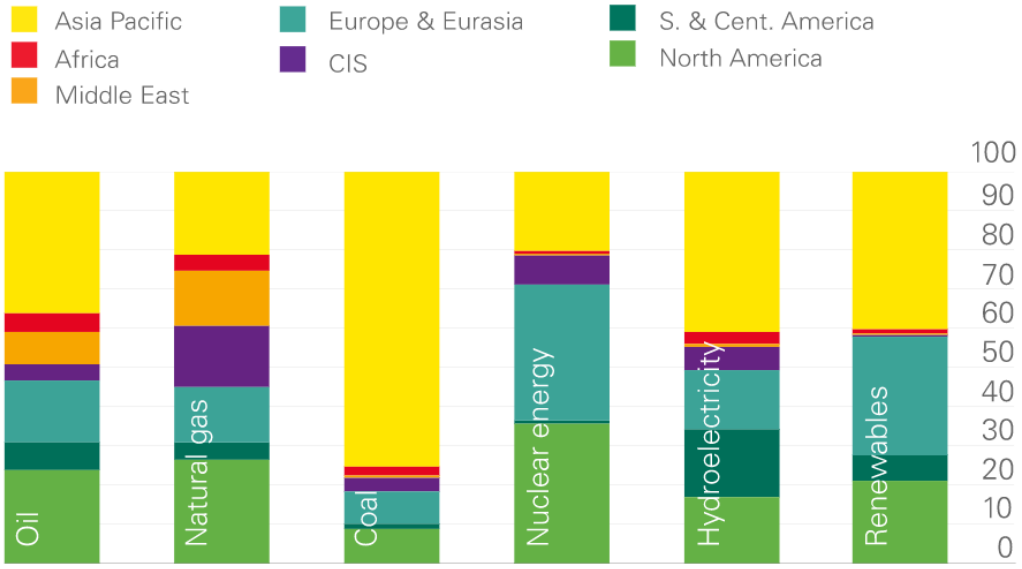
Fonte: BP Statistical review of world energy (2018).

O petróleo é consumido principalmente na Ásia-Pacífico e na América do Norte. Juntos, eles representam 60% do consumo global (ver Figura 2).

Com relação do histórico do preço do petróleo, há uma grande oscilação. De 1999 até meados de 2008, o preço do petróleo subiu significativamente, devido a crescente demanda de petróleo em países como China e Índia. No meio da crise financeira de 2007-2008, o preço do petróleo sofreu uma queda significativa após o pico recorde de US$ 147,27 atingido em julho de 2008. O preço se recuperou após a crise e subiu para US$ 82 o barril em 2009. Em meados de 2014, o preço começou a cair devido ao aumento significativo na produção de petróleo nos EUA e à demanda em declínio nos países emergentes. O excesso de petróleo, causado por

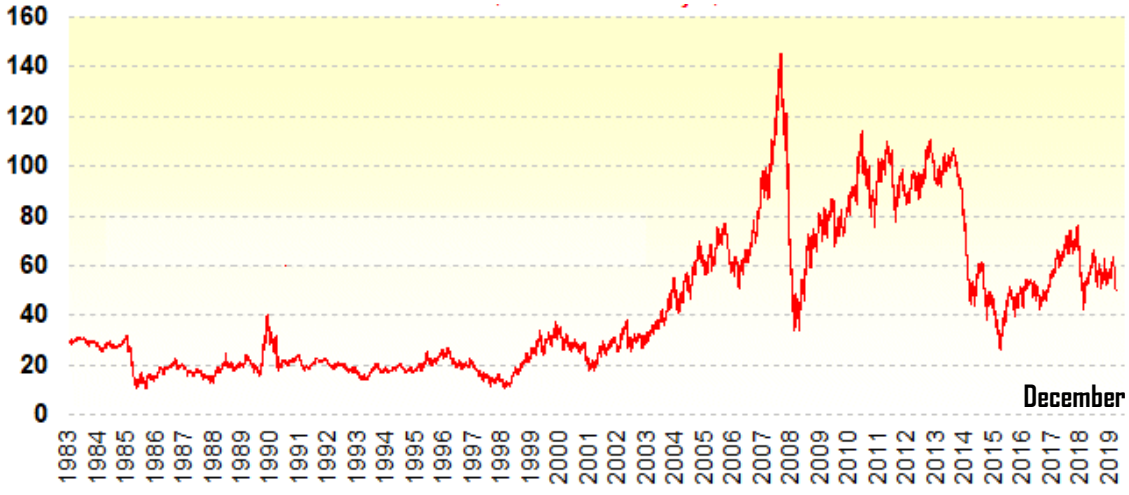
vários fatores, provocou uma queda no preço do petróleo que continuou até fevereiro de 2016. Em 2019, o preço do petróleo Brent, referência internacional, ficou em média US$ 64 por barril, US$ 7/b abaixo da média de 2018. O preço do petróleo bruto West Texas Intermediate (WTI), a referência dos EUA, teve uma média de US$ 57/b em 2019, US$ 7/b menor que em 2018 (EIA,2020).

Figura 2 – Consumo de combustível por região



Fonte: BP Statistical review of world energy (2018).

Figura 3 – Histórico de Preços do Petróleo



Fonte: EIA (2020).

Apesar de toda oscilação em decorrência da economia mundial, é inegável que a demanda por petróleo se mantém imprescindível. Destaca-se também a crescente importância dos petroquímicos no crescimento da demanda de petróleo.

O petróleo tem origem na matéria orgânica depositada junto com sedimentos. A interação dos fatores – matéria orgânica, sedimento e condições termogênicas apropriadas – é fundamental para a formação de petróleo. Uma vez gerado e migrado, o petróleo é acumulado em uma rocha, denominada de rocha reservatório com porosidade e permeabilidade adequadas (THOMAS et al, 2001). Através de modelos geológicos e geofísicos são identificadas as áreas favoráveis à acumulação de petróleo, que atuando em conjunto conseguem indicar o local mais propício para perfuração.

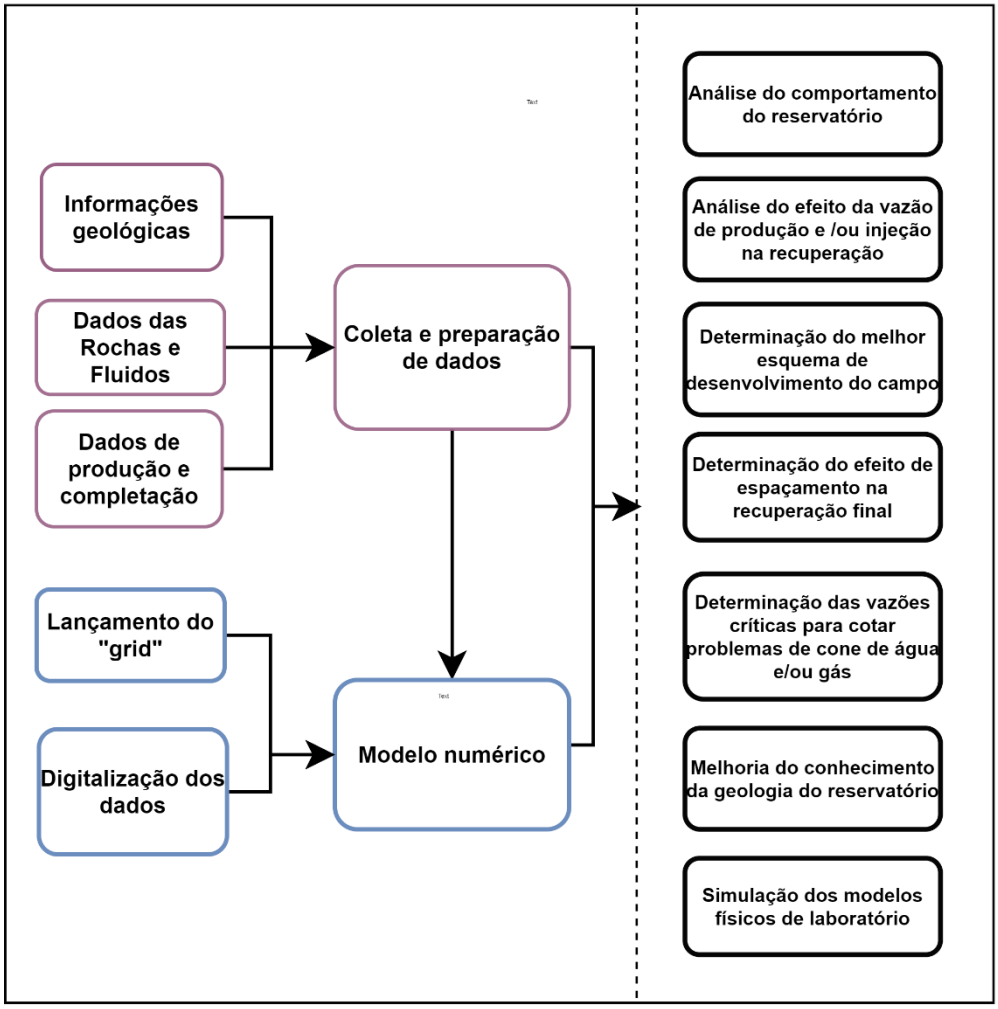
Em virtude da demanda global de petróleo, bem como, da necessidade de obter resultados precisos para previsões de um reservatório sobre diferentes condições de operações, há um crescente interesse na simulação e gerenciamento de reservatórios de petróleo.

Dessa forma, o uso da simulação de reservatório como ferramenta preditiva se tornou um padrão na indústria de petróleo. Modelos numéricos usam computadores para resolver equações que descrevem o comportamento físico do processo em reservatório (ERTEKIN, 2001).

O esquema da Figura 4 fornece uma ideia da importância do uso da simulação numérica no estudo de reservatórios. Verifica-se que para o modelo matemático convergem as informações geológicas, informações sobre as propriedades das rochas e fluidos presentes no meio poroso, informações sobre a produção e outras a respeito dos poços de petróleo, tais como características de completação (ROSA, 2011).

O uso do simulador permite obter informações sobre o desempenho de um campo de acordo com diversos esquemas de produção, de modo que possam ser determinadas as condições ótimas para se produzir nesse campo. Mais especificamente, pode ser analisado o comportamento de um reservatório quando sujeito à injeção de diferentes tipos de fluidos, analisada a influência de diferentes vazões de produção e/ou injeção, ou determinado o efeito da localização dos poços e espaçamento entre eles na recuperação final de óleo e/ou gás (ROSA, 2011).

Figura 4 – Simulador de Reservatórios



Fonte: Adaptado de ROSA (2011).

No presente trabalho, será utilizado como simulador o *MATLAB Reservoir Simulation Toolbox* (MRST). Ele consiste em uma ferramenta de livre acesso ao código fonte, composto por uma variedade de estruturas de dados e métodos computacionais, empregados na solução de diversos problemas relacionados ao escoamento superficial e subterrâneo, e tem se tornado cada vez mais popular em instituições acadêmicas ao redor do mundo, assim como, na indústria de petróleo (LIE, 2016).

Acoplados à simulação, o estudo e utilização de técnicas de otimização auxiliam no gerenciamento de reservatórios, porém requer a utilização do simulador diversas vezes, exigindo demanda computacional excessiva. Dessa forma, são desenvolvidos métodos que se aproximam dos resultados da simulação com execução mais rápida. No presente trabalho, serão descritos, com detalhes, os procedimentos para a implementação de uma técnica de otimização semi-intrusiva, *Trajectory Piecewise Linearization* (TPWL), utilizando o MRST.

1.1 JUSTIFICATIVA E MOTIVAÇÃO

Na área de gerenciamento de reservatórios de petróleo existe uma grande variedade de problemas que podem ser tratados com aplicação de técnicas de otimização. No planejamento de novos poços, questões-chave incluem a determinação da localização ideal do poço, tipo de poço (injetor ou produtor; vertical, horizontal ou multilateral) e cronograma de perfuração, dadas as restrições operacionais e econômicas. Para os poços existentes, um benefício significativo pode ser alcançado através da otimização das configurações do poço (controles do poço), como pressões do fundo de poço (BHPs) ou vazões, em função do tempo. Ao otimizar essas configurações, a produção de petróleo pode ser aumentada e o tempo em que os fluidos injetados aparecem nos poços de produção (referidos como *breakthrough*) pode ser atrasado (JANSEN e DURLOFSKY, 2017).

Na otimização dos controles dos poços são determinadas as configurações ótimas de poço, em geral utilizando como função objetivo o Valor Presente Líquido, ou simplesmente, VPL, que mede o retorno econômico do campo ao fim do tempo de concessão. Dessa forma, são determinados os controles que maximizam a produção e o ganho econômico do poço no reservatório.

Esse processo de otimização em geral requer a execução da simulação de fluxo inúmeras vezes. Dessa forma, em um modelo com um grande número de elementos de malha e/ou física complicada, a demanda computacional no processo de otimização é excessiva.

Portanto, essa foi a motivação principal para o desenvolvimento de modelos de ordem reduzida, mais conhecidos por modelos substitutos. Esses modelos devem fornecer uma aproximação dos resultados da simulação, porém devem possuir uma execução muito mais rápida.

Nesse contexto, conforme mencionado anteriormente, foi desenvolvida a técnica semi-intrusiva de modelo substituto, *Trajectory Piecewise Linearization* (TPWL), para redução da complexidade do problema utilizando o MRST, visto que ele possui o código fonte aberto.

1.2 REFERENCIAL TEÓRICO

A otimização do controle dos poços foi a motivação principal para o desenvolvimento da técnica proposta no presente trabalho, uma vez que é uma questão de grande interesse para a engenharia de reservatórios de petróleo.

As técnicas de otimização podem ser classificadas de acordo com o grau de intrusão no código de programação do simulador de fluxo: (a) altamente intrusivas; (b) semi-intrusivas e (c) não-intrusivas.

As técnicas altamente intrusivas utilizam métodos adjuntos para calcular o gradiente da função objetivo com relação às variáveis de controle (JANSEN, 2011). Tais métodos são em geral os mais eficientes (BROWER e JANSEN, 2004; SARMA *et al.*, 2008; CHEN *et al.*, 2010, 2012), porém necessitam de grande esforço de programação para serem implementados, por isso, não são facilmente encontrados nos simuladores comerciais.

Os métodos semi-intrusivos utilizam de modelos de ordem reduzida como *Trajectory Piecewise Linearization* - TPWL (CARDOSO, 2009; HE, 2010; CARDOSO e DURLOFSKY, 2010; HE *et al.*, 2011; HE e DURLOFSKY, 2014) ou *Discrete Empirical Interpolation* - DEIM (GILDIN *et al.*, 2013; CHATURANTABUT e SORENSEN, 2009). Podem ainda utilizar a simulação de linhas de fluxo a fim de igualar o instante de erupção de água nos poços (ALHUTHALI *et al.*, 2009).

Já as técnicas não intrusivas usam o simulador como uma "caixa preta" e são puramente baseados nos dados das simulações anteriores. As principais técnicas existentes são aquelas que usam algoritmos evolucionários (OLIVEIRA, 2006; ALMEIDA *et al.*, 2010; SOUZA *et al.*,2010) e métodos baseados em modelos substitutos (QUEIPO *et al.*, 2002; CMOST, 2012).

Cabe destacar ainda outros algoritmos que não são baseados em derivadas. Eles aproximam os gradientes da função objetivo através de métodos estocásticos (WANG *et al.*, 2009) e métodos baseados em *ensembles* (*ensemble-based*) (CHEN e OLIVER, 2010) que podem ser corrigidos por cálculos de diferenças finitas (XIA e REYNOLDS, 2013) ou serem incorporados em um modelo de interpolação quadrático (ZHAO *et al.*, 2011). Uma discussão mais completa é encontrada em (CONN *et al.*, 2009).

No presente trabalho, o método considerado é do tipo semi-intrusivo conhecido como *Trajectory Piecewise Linearization* – TPWL. Esse método foi aplicado inicialmente no contexto da microeletrônica (REWIENSKI, 2003). Para a simulação de reservatórios, o TPWL

foi aplicado em Cardoso (2009) e aperfeiçoado por Cardoso e Durlofsky (2010), He (2010), He *et al.* (2011), He e Durlofsky (2014, 2015). O TPWL consiste na linearização das equações de fluxo em torno de estados previamente convergidos e gravados durante uma ou mais simulações de treinamento. Esse método é utilizado como redutor da complexidade numérica na aproximação (MARKOVINOVIC, 2003). Ele é geralmente combinado com a técnica *Proper Orthogonal Decomposition* - POD para reduzir a dimensão do problema. Existem alguns relatos de instabilidade na literatura acerca deste método (HE, 2010; GILDIN *et al.*, 2013). Entretanto, o problema de instabilidade parece ter sido resolvido ao trocar-se a projeção de Bubnov-Galerkin por uma projeção de Petrov-Galerkin (CARLBERG *et al.*, 2009; HE e DURLOFSKY, 2014).

Os desenvolvimentos do TPWL/POD em Cardoso (2009) e He (2010) foram aplicados ao problema de simulação bifásica (óleo e água) com formulação baseada em saturações. Em He e Durlofsky (2014) aplicou-se a técnica de redução a uma formulação composicional baseada em frações molares. Em Machado (2014), a formulação considerada é água-óleo baseada em frações mássicas.

No presente trabalho, a formulação considera a implementação da técnica TWPL para a formulação bifásica baseada em saturações, utilizando como simulador o MRST. O MATLAB Reservoir Simulation Toolbox (LIE, 2016) é uma ferramenta de livre acesso ao código fonte, para a simulação do escoamento sub-superficial e subterrâneo, estruturado em módulos e composto por uma variedade de estruturas de dados e métodos computacionais.

Hewson (2015) desenvolveu a implementação do TPWL-POD no MRST, incorporando ainda o ajuste de histórico. Contudo, destaca-se que a implementação aqui realizada difere da sua, uma vez que, em Hewson (2015) a implementação foi realizada no módulo, denominado *ad-fi*, primeira implementação dos *solvers* de diferenciação automática para modelos *black-oil*, mas que atualmente está desativado.

Portanto, no presente trabalho tem-se uma atualização da implementação do TPWL no MRST, utilizando a família de módulos da diferenciação automática, consolidada a partir da versão MRST2016a. A versão utilizada no presente trabalho foi MRST2018b.

1.3 OBJETIVOS GERAIS E ESPECÍFICOS

Esta dissertação não busca desenvolver uma nova técnica para modelos substitutos, mas sim realizar a implementação de uma técnica de redução de complexidade, consolidada na literatura, em um simulador de reservatórios com livre acesso ao código fonte. Portanto, deverá ser realizada a implementação da técnica *Trajectory Piecewise Linearization* – TPWL no *MATLAB Reservoir Simulation Toolbox* – MRST.

Com essa finalidade, os objetivos específicos desse trabalho envolvem:

a) Estudar em detalhes o *MATLAB Reservoir Simulation Toolbox – MRST* para desenvolver algoritmos semi-intrusivos de otimização.
b) Criar um manual básico do MRST, destacando pontos chaves para a sua utilização e compreensão de sua formulação de fluxo.
c) Compreender o MRST AD-OO (*MATLAB Reservoir Simulation Toolbox - Automatic Differentiation - Object Oriented*), nova estrutura com programação orientada objeto do MRST.
d) Estudar, implementar e analisar a técnica *Trajectory Piecewise Linearization* – TPWL no simulador de reservatórios.

1.4 ORGANIZAÇÃO DA DISSERTAÇÃO

A dissertação está dividida em seis partes principais. O capítulo 2 destina-se a apresentação da formulação matemática e numérica para a equação de fluxo. Esse capítulo permite um leitor leigo conhecer os fundamentos básicos da simulação de reservatórios, introduzindo as equações de fluxo requeridas nos demais capítulos da dissertação.

O capítulo 3 apresenta o simulador escolhido para o desenvolvimento da pesquisa. A escolha se deve à necessidade de acesso ao código do simulador. Serão apresentados as estruturas e os métodos do MRST, permitindo que os leitores saibam utilizá-lo para exemplos básicos, com essa finalidade, serão alguns apresentados trechos de código ao longo do capíttulo. Além disso, a descrição permitirá compreender a estrutura do MRST AD-OO, ou seja, será apresentada a formulação das equações de fluxo com a utilização da diferenciação automática e programação orientado a objeto.

O capítulo 4 dedica-se a estudar a técnica TPWL (*Trajectory Piecewise Linearization*) como redutor da complexidade numérica. Serão apresentados os aspectos teóricos da técnica, bem como, descritos detalhes da sua implementação através da exportação dos estados convergidos e derivadas dos resíduos extraídas do MRST.

Por fim, no capítulo 5 são apresentados alguns resultados comparativos da simulação do MRST (alta fidelidade) com o TPWL (baixa fidelidade). Para tal estudo é utilizado o modelo SPE10, modelo *benchmark* para a simulação de reservatórios. Dessa forma, será possível obter conclusões a respeito da acurácia e ganho computacional do modelo substituto.

Ao final do trabalho, no capítulo 6, são apresentadas algumas conclusões acerca do uso do TPWL no MRST, bem como, desenvolvidas algumas sugestões de trabalhos futuros.

2 CONCEITOS DE SIMULAÇÃO DE RESERVATÓRIOS

Nesta seção, serão apresentados os conceitos e formulações matemáticas para a descrição do escoamento em meio poroso em reservatórios de petróleo. Após a descrição da formulação matemática na primeira seção, será apresentada a formulação numérica, onde as equações diferenciais do fluxo são discretizadas em diferenças finitas.

2.1 FORMULAÇÃO MATEMÁTICA

O modelo a ser estudado é denominado por *black-oil*. Ele consiste em um modelo isotérmico, composto por três componentes e três fases (água, óleo e gás). As fases água e óleo são imiscíveis e sem troca de massa, e o componente gás é solúvel no óleo. Serão descritas a seguir as equações que descrevem o modelo.

2.1.1 Modelo Black-Oil

O modelo *black-oil* apresenta três fases imiscíveis no reservatório: água (w), óleo (o) e gás (g). Nas condições do reservatório, os dois componentes (óleo e gás) podem ser parciais ou completamente dissolvidos um no outro, dependendo da pressão, formando uma fase líquida e uma fase gasosa. Dessa forma, na fase óleo podem existir os componentes óleo e gás dissolvidos, e na fase gasosa existem os componentes denominados gás livre e gás dissolvido. Além disso, existe uma fase aquosa, assumida composta apenas por água. A seguir são apresentadas as equações que descrevem o modelo.

$$\frac{\partial\left(\phi b_w s_w\right)}{\partial t} + \nabla \cdot \left(b_w v_w\right) - b_w q_w = 0 \tag{1}$$

$$\frac{\partial\left[\phi\left(b_o s_o + b_g r_v s_g\right)\right]}{\partial t} + \nabla \cdot \left(b_o v_o + b_g r_v v_g\right) - \left(b_o q_o + b_g r_v q_g\right) = 0 \tag{2}$$

$$\frac{\partial\left[\phi\left(b_g s_g + b_o r_s s_o\right)\right]}{\partial t} + \nabla \cdot \left(b_g v_g + b_o r_s v_o\right) - \left(b_g q_g + b_o r_s q_o\right) = 0 \tag{3}$$

Onde:

ϕ – porosidade, definida como a razão entre o volume de vazios e o volume total de uma rocha;

p_α – pressão da fase $\alpha = w,o,g$;

q_α – termo fonte volumétrico para fase $\alpha = w,o,g$;

b_α – inverso do fator volume de formação ($1/B_\alpha$) para $\alpha = o,g$. O fator volume-formação $\left(B_\alpha\right)$ é a relação entre o volume que ele ocupa em uma determinada condição de temperatura e pressão e o volume ocupado nas condições padrão de superfície.

r_s – razão de solubilidade do gás no óleo, é a relação entre o volume de gás dissolvido, expresso nas condições-padrão, e o volume de óleo, também expresso em condições padrão.

r_v – solubilidade de óleo no gás, definida como a quantidade de óleo condensado na superfície que pode ser vaporizado em gás nas condições de reservatório.

A velocidade superficial (v_α) é dada pela Lei de Darcy, escrita de uma forma generalizada para uma fase α, pela seguinte equação (BEAR, 1972):

$$v_\alpha = -\lambda_\alpha K \left(\nabla p_\alpha - \rho_\alpha g \nabla z\right) \quad \alpha = o,w,g \tag{4}$$

Onde:

K é o tensor de permeabilidade absoluta da rocha;

λ_α é a mobilidade da fase α, dada por $\lambda_\alpha = kr_\alpha / \mu_\alpha$, onde μ_α e kr_α são, respectivamente, a viscosidade e permeabilidade relativa da fase α (ver seção 2.1.3);

g é o módulo da aceleração gravitacional;

z é o componente do vetor deslocamento orientado para baixo;

ρ_α é a massa específica ou a "densidade" da fase α.

O termo q_α é determinado pelo modelo de poço, que busca calcular a taxa de injeção ou produção do reservatório quando a pressão no raio do poço é conhecida. A relação resultante entre a pressão no fundo de poço e a vazão na superfície é conhecida por IPR (*inflow-performance relation*). A mais simples e abrangente IPR é descrita pela lei linear: (LIE, 2016)

$$q = WI\left(p_{\mathrm{Res}} - p_{well}\right) \tag{5}$$

Dessa forma, percebe-se que a vazão no poço é diretamente proporcional ao rebaixamento da pressão na região próxima ao poço. A constante de proporcionalidade, representada na Eq. (5) por WI , é denominada de índice de produtividade e injetividade para o poço produtor e injetor, respectivamente. No MRST não há diferença entre ambos (LIE, 2016). Esse índice depende das propriedades do reservatório e do fluido, bem como, fatores geométricos que afetam o fluxo.

No simulador utilizado, é adotado o modelo de Peaceman, onde o índice de injetividade é dado pela Eq. (6) (PEACEMAN, 1991; ERTEKIN, 2001; LIE, 2016).

$$WI = \frac{2\pi h K_e}{\mu B\left[\ln\left(r_e / r_w\right) + S\right]} \tag{6}$$

Onde:

h – comprimento do poço;

K_e – permeabilidade efetiva (para meio anisotrópico), no caso de um modelo bidimensional discretizado ela é dada pela seguinte expressão:

$$\mathrm{K}_e = \sqrt{K_x K_y} \quad r_e = \frac{\left(\sqrt{K_y / K_x}\,\Delta x^2 + \sqrt{K_x / K_y}\,\Delta y^2\right)^{1/2}}{\left(K_y / K_x\right)^{1/4} + \left(K_x / K_y\right)^{1/4}} \tag{7}$$

r_e – raio equivalente (para malhas com elementos retangulares, $\Delta x \neq \Delta y$);

r_w – raio do poço;

S – *skin fator*.

Incluindo a força gravitacional, assumindo o equilíbrio hidrostático e considerando o modelo *black-oil*, a IPR escrita em termo da taxa de produção volumétrica para cada fase α , considerando cada conectividade i , conforme disposto no MRST, é dada por (LIE, 2016):

$$q_{\alpha,i} = -\lambda_{\alpha,i} WI_i\left(p_i - p_{bh} - H_{wi}\right) \qquad WI_i = \frac{2\pi h K_e}{\left[\ln\left(r_e / r_w\right) + S\right]} \tag{8}$$

Onde:

$\lambda_{\alpha,i}$ é a mobilidade da fase α ;

WI_i é o índice de injetividade/produtividade do poço;

p_i é a pressão no bloco perfurado;

p_{bh} é a pressão de fundo de poço;

H_{wi} é a pressão hidrostática.

Descrevendo a equação constitutiva ou de restrição das saturações como $s_w + s_o + s_g = 1$ e fornecendo funções capilares, dependentes de saturação, que relacionam às pressões das fases, se obtém o modelo fechado (sistema com três equações e três incógnitas primárias).

2.1.2 Escoamento Bifásico

Nesse trabalho será estudado especificamente o escoamento bifásico de óleo e água em meio poroso. O meio é considerado totalmente saturado, portanto, pode-se escrever:

$$s_w + s_o = 1 \tag{9}$$

Em virtude da curvatura e da tensão superficial entre as duas fases, a pressão na fase molhante (água) é menor que na fase não molhante (óleo). A diferença é dada pela pressão capilar (Eq. (10)), que empiricamente é função da saturação.

$$p_c = p_o - p_w \tag{10}$$

Partindo da Eq. (1) a Eq. (4), considerando a presença apenas da fase óleo e água, é possível descrever o modelo pelas seguintes equações:

$$\frac{\partial\left(\phi b_w s_w\right)}{\partial t} - \nabla \cdot \left[b_w \lambda_w K \left(\nabla p_w - \rho_w g \nabla z\right)\right] - b_w q_w = 0 \tag{11}$$

$$\frac{\partial\left(\phi b_o s_o\right)}{\partial t} - \nabla \cdot \left[b_o \lambda_o K \left(\nabla p_o - \rho_o g \nabla z\right)\right] - b_o q_o = 0 \tag{12}$$

Desse modo, a Eq.(11) e Eq.(12), em conjunto com a Eq.(9) e Eq.(10), descrevem por completo o modelo bifásico formado pelo sistema de duas equações e duas incógnitas primárias (em geral p_o e s_w).

2.1.3 Permeabilidade Relativa

Nessa seção será descrito a equação adotada para a principal fonte de não linearidade para os modelos considerados, ou seja, as curvas de permeabilidade relativa do óleo e da água. Tais curvas de permeabilidade relativa podem ser fornecidas através de tabelas contendo alguns pontos que são interpolados ou podem utilizar o modelo de Corey, expresso pelas Eq.(13) e Eq.(14), que representam respetivamente, as permeabilidades relativas do óleo (k_{ro}) e da água (k_{rw}), em função da saturação da água (S_w):

$$k_{ro} = k_{ro}^0 \left(\frac{1 - S_w - S_{or}}{1 - S_{wc} - S_{or}} \right)^a \tag{13}$$

$$k_{rw} = k_{rw}^0 \left(\frac{1 - S_w - S_{or}}{1 - S_{wc} - S_{or}} \right)^a \tag{14}$$

Onde:

S_{or} – saturação de óleo residual que representa a mínima saturação que normalmente pode se obter;

S_{wc} – saturação de água conata que representa a mínima saturação de água que normalmente existia no reservatório antes da injeção de água;

k_{ro}^0 – ponto terminal para a permeabilidade relativa do óleo, medido na saturação S_{wc};

k_{rw}^0 – ponto terminal para a permeabilidade relativa da água, medido na saturação $1 - S_{or}$;

a – expoente de Corey para a curva de permeabilidade relativa do óleo;

b – expoente de Corey para a curva de permeabilidade relativa de água.

2.2 FORMULAÇÃO NUMÉRICA

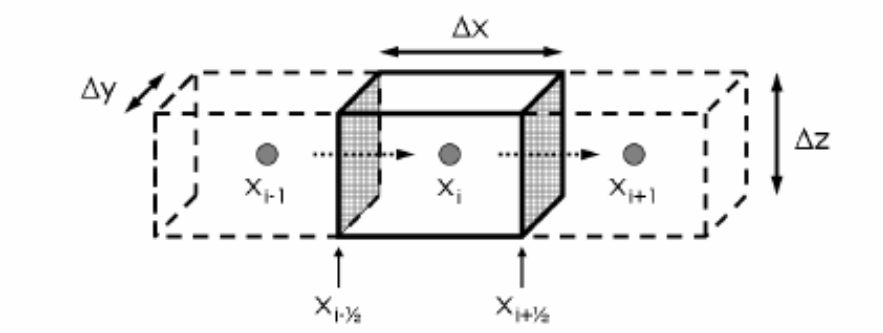
Modelos numéricos usam computadores de alta velocidade para resolver equações matemáticas, que descrevem o comportamento físico de um reservatório, para obter solução numérica desse comportamento no campo. Em virtude da não linearidade das equações que descrevem o processo, não é possível utilizar técnicas analíticas e as soluções são obtidas com

métodos numéricos (aproximados), onde os valores da pressão e saturação são obtidos em pontos discretos do reservatório, aproximando a solução exata. Dessa forma, a discretização é o processo de converter equações diferencias parciais em equações algébricas (ERTEKIN, 2001).

Para a construção do modelo numérico faz-se necessário realizar o lançamento do *grid* (malha). Esta etapa consiste então em dividir o reservatório em várias células, onde cada uma delas é identificada por suas coordenadas *x, y, z* (ROSA et al, 2011). Dessa forma, se obtém uma malha no sistema cartesiano com n_c elementos seguindo o ordenamento lógico *i, j, k*.

Para realizar a discretização das equações apresentadas na seção 2.1.2, considere a representação de um elemento da malha na Figura 5, admitindo que as dimensões de cada elemento do bloco são constantes (Δx, Δy, Δz) e o fluxo é unidimensional (na direção x). Por simplificação, será considerado ainda o caso incompressível, ou seja, o fator volume formação é $b_\alpha = 1$ e a densidade constante, assim como, desprezado os efeitos gravitacionais e a pressão capilar, ou seja, $p_o = p_w$.

Figura 5 – Representação do volume de controle e seus vizinhos



Fonte: CARDOSO (2009).

Será adotada a discretização totalmente implícita, geralmente utilizada em simuladores comerciais. Dessa forma, é possível escrever o primeiro termo das Eq. (11) e (12), referentes ao termo de acumulação, da seguinte forma discreta:

$$\partial_t \left(\phi b_\alpha s_\alpha\right) \approx \frac{1}{\Delta t}\left[\left(\phi b_\alpha s_\alpha\right)^{n+1} - \left(\phi b_\alpha s_\alpha\right)^{n}\right] \tag{15}$$

Onde o subscrito α indica a fase, o subscrito n é o passo de tempo (timestep) e $n+1$ especifica o próximo passo de tempo. Para o caso simplificado em questão, sistema incompressível ϕ é constante e $b_\alpha = 1$, logo a Eq.(15) é reduzida para:

$$\partial_t \left(\phi b_\alpha s_\alpha\right) \approx \frac{\phi}{\Delta t}\left[\left(s_\alpha\right)^{n+1} - \left(s_\alpha\right)^{n}\right] \tag{16}$$

O segundo termo da Eq. (11) e Eq. (12) representa o termo de fluxo, sua forma discretizada é dada por:

$$\nabla \cdot \left[b_\alpha \lambda_\alpha K \left(\nabla p_\alpha \right) \right] = \frac{\partial}{\partial x} \cdot \left[K \lambda_\alpha \left(\frac{\partial p_\alpha}{\partial x} \right) \right]$$
$$\nabla \cdot \left[b_\alpha \lambda_\alpha K \left(\nabla p_\alpha \right) \right] \approx \left\{ \left(T_\alpha \right)_{i-1/2}^{n+1} (p_{i-1}^{n+1} - p_i^{n+1}) + \left(T_\alpha \right)_{i+1/2}^{n+1} (p_{i+1}^{n+1} - p_i^{n+1}) \right\} \frac{1}{V} \quad (17)$$

Onde i se refere a um bloco da malha e $V = \Delta x \Delta y \Delta z$ é o volume do bloco i. O termo $\left(T_\alpha \right)_{i-1/2}^{n+1}$ e $\left(T_\alpha \right)_{i+1/2}^{n+1}$ indicam transmissibilidades, que relacionam o fluxo na fase α com a diferença de pressão entre o bloco i e $i-1$, conforme indicado na Eq. (18) (ERTEKIN, 2001; CHEN et al, 2006).

$$\left(T_\alpha \right)_{i-1/2}^{n+1} = \left(\frac{KA}{\Delta x} \right)_{i-1/2} \left(b_\alpha \lambda_\alpha \right)_{i-1/2}^{n+1} \quad (18)$$

Sendo:

$A = \Delta y \Delta z$, igual à área da face comum entre os blocos $i-1$ e i;

$K_{i-1/2}$ é calculada como a média harmônica de K_{i-1} e K_i;

$\left(b_\alpha \lambda_\alpha \right)_{i-1/2}^{n+1} = \left(b_\alpha \, kr_\alpha / \mu_\alpha \right)_{i-1/2}^{n+1}$, ponderado a montante com relação à direção do escoamento da fase.

O último termo na Eq. (11) e Eq. (12) representa o termo fonte/sumidouro. Na simulação de reservatório ele corresponde ao poço, modelado pela Eq. (19).

$$\left(q_\alpha \right)_i^{n+1} = \left(q_\alpha \right)_i^{n+1} V = - \cdot WI \cdot \left(\lambda_\alpha \right)_i^{n+1} \left(p_{bh} - p_i^{n+1} \right) \quad (19)$$

Onde $\left(q_\alpha \right)_i^{n+1}$ é a taxa de vazão volumétrica da fase α de um bloco i no tempo *n+1*, p_i^{n+1} é a pressão no bloco no tempo *n+1*, p_{bh} é a pressão BHP para o poço *w* no bloco i, e *WI* é índice de injetividade, descrito pelo modelo Peaceman, conforme indicado na Eq. (8).

A partir das discretizações apresentadas da Eq. (15) até a Eq. (19), a versão discretizada totalmente implícita das Eq. (11) e Eq. (12) para um sistema incompressível pode ser representado por (AZIZ e SETTARI, 1986):

$$\mathrm{g} = T^{n+1} x^{n+1} - D^{n+1} \left(x^{n+1} - x^n \right) - Q^{n+1} = 0 \quad (20)$$

Onde g é o vetor do resíduo que se procura zerar, x é o vetor de estado que contém as incógnitas primárias (p_o e s_w). O termo $T^{n+1}x^{n+1}$ representa o transporte (fluxo) obtido pela Eq. (17). T é uma matriz pentadiagonal em bloco para malhas bidimensionais e uma matriz heptadiagonal em bloco para malhas tridimensionais. O termo $D^{n+1}\left(x^{n+1}-x^{n}\right)$ representa a acumulação, conforme obtido na Eq.(15), sendo D uma matriz diagonal. Por fim, Q representa o termo fonte/sumidouro expresso pela Eq. (19). As matrizes T , D e Q dependem de x e devem ser atualizadas para cada iteração todo passo de tempo.

A Eq. (20) representa um conjunto não linear de equações algébricas e é resolvido aplicando o método de Newton (ver **APÊNDICE A – MÉTODO DE NEWTON-RAPHSON**) para conduzir g à zero, conforme indica a Eq. (21):

$$J\,\delta = -\mathrm{g} \tag{21}$$

Onde J é a matriz jacobiana das derivadas parciais e δ é a diferença do vetor de estado a cada iteração.

3 MATLAB RESERVOIR SIMULATION TOOLBOX - MRST

Para o desenvolvimento do presente trabalho foi utilizado o *Matlab Reservoir Simulation Toolbox* (MRST) como simulador. Dentre outras razões, o MRST foi escolhido principalmente por permitir o completo acesso ao código fonte.

Nesse capítulo, serão apresentados os principais conceitos relacionados à simulação numérica nessa ferramenta. Serão dispostos diversos trechos de código com o intuito de permitir que o leitor obtenha as principais noções de programação nessa ferramenta.

Na primeira seção é realizada uma apresentação do MRST e dos seus comandos iniciais. Em seguida, é apresentada a estrutura e comandos para a geração de malhas cartesianas no MRST (seção 3.2), bem como para a definição das propriedades das rochas (seção 3.3).

Posteriormente, apresenta-se como se utilizar as equações de modelagem de escoamento, indicadas no Capítulo 2, no MRST. Inicialmente, por fins didáticos, na seção 3.5, discute-se como realizar a simulação utilizando a programação procedural, com o *solver* incompressível e monofásico.

Logo após, são introduzidos os conceitos de operadores diferencias discretos (seção 3.6), para escrever as equações de fluxo na forma residual; prototipagem rápida (seção 3.7), associada à vetorização da programação; programação orientado objeto (seção 0), apresentando suas principais características e elementos; e por fim, a diferenciação automática no contexto geral e no MRST (seção 3.9).

Finalmente, demonstra-se na seção 3.10 como realizar a simulação do escoamento bifásico com o uso dos operadores discretos combinados com diferenciação automática para prototipagem de novos *solvers.*

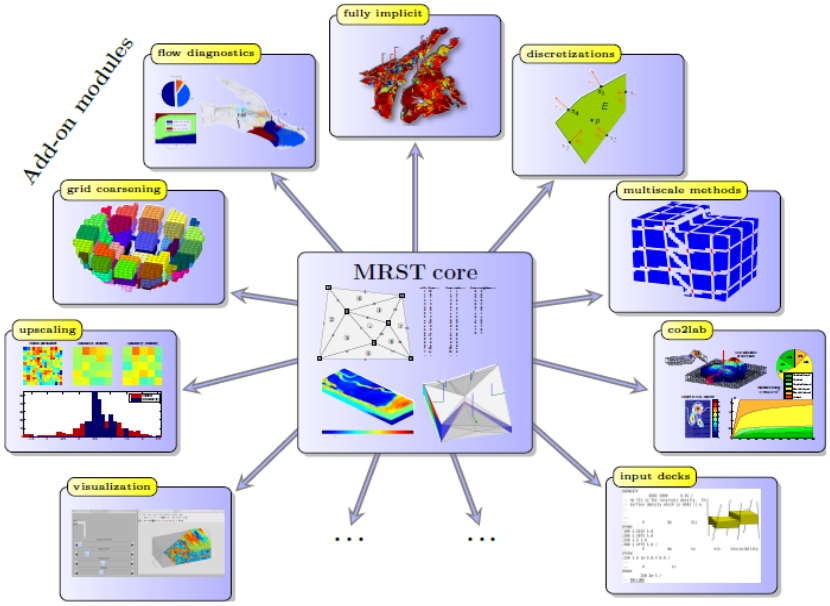
3.1 VISÃO GERAL DO MRST

O *MATLAB Reservoir Simulation Toolbox (MRST)* é uma ferramenta de livre acesso ao código fonte, composto por uma variedade de estruturas de dados e métodos computacionais, empregados na solução de diversos problemas relacionados ao escoamento superficial e subterrâneo. Ele tem se tornado cada vez mais popular em instituições acadêmicas ao redor do mundo, assim como, na indústria de petróleo, atraindo interesse de diversos pesquisadores, profissionais e estudantes para a simulação de reservatórios de petróleo (LIE, 2016).

Foi desenvolvido inicialmente pelo grupo de Geociências Computacionais no Departamento de Matemática e Cibernética do SINTEF Digital. Além de incluir módulos desenvolvidos por pesquisadores de Heriot-Watt University, NTNU, University of Bergen, TNO, e TU Delft.

O MRST é basicamente composto por duas partes, conforme indica a Figura 6.

Figura 6 – Organização do Matlab Reservoir Simulation Toolbox - MRST



Fonte: KROGSTAD (2015).

A essência do MRST é um módulo central, chamado *MRST-core*, que inclui rotinas e estruturas de dados relacionadas às funcionalidades básicas, tais como, criação e manipulação de malhas, determinação de propriedades petrofísicas, condições de contorno, termos fontes sumidouros e poços. Além disso, estão incluídas as funcionalidades primordiais da diferenciação automática (*AD – Automatic Differentiation*), detalhada na seção 3.9.1. Essas rotinas presentes no núcleo do MRST estão consolidadas e documentadas desde as primeiras versões, dessa forma, não há expectativas de mudanças em sua estruturação.

A segunda e maior parte do software consiste em um conjunto de módulos complementares (*add-on modules)* ao *MRST-core*. Esses conjuntos de módulos complementares variam entre si no que diz respeito às diferentes funcionalidades que agregam.

Essa estrutura de módulos visa organizar o desenvolvimento do *software*, separando o código genérico do específico, permitindo assim a ativação ou desativação de funcionalidades

específicas, conforme a utilização do usuário. Um módulo no MRST nada mais é do que uma coleção de funções, objetos e exemplos agrupados em uma pasta.

Podem ser destacadas oito principais categorias de módulos complementares.

Primeiramente, pode ser destacada a categoria que agrega outras funcionalidades, além das apresentadas no núcleo do MRST, no que diz respeito à **Geração e Processamento de Malhas**. Alguns módulos que podem ser aqui mencionados são: *agglom e coarsegrid*, relacionadas com a criação de partições para a geração de malha grossa; *libgeometry*, que calcula a geometria da malha (centroides, volumes, áreas), utilizando a linguagem C para reduzir os custos computacionais; *opm_gridprocessing*, construção de malhas *Corner-point* grids a partir da entrada de dados do ECLIPSE, utilizando a linguagem C; *triangle*, oferece um gerador bidimensional e triangulação de Delaunay para malhas do MRST; e *upr*, apresenta funcionalidades para gerar malhas poliédricas não estruturadas que se alinham a elementos geométricos pré-descritos.

Em seguida, tem-se a categoria denominada por **Discretizações do Modelo Incompressível e *Solvers***. Ela inclui o módulo *incomp* que apresenta a implementação do método *two point flux approximation (TPFA)* para o problema de fluxo, e formulação implícita ou explícita com ponderação a montante para o problema de transporte, associados ao escoamento monofásico ou bifásico de fluidos incompressíveis e imiscíveis. Em virtude da inconsistência e erros de orientação da malha, para a busca da convergência, foram desenvolvidos módulos com outros *solvers: mimetic, mpfa* (*multi point flux approximation)* e *vem (virtual element methods).* Essa família apresenta seus *solvers* implementados com programação procedural e está bem consolidada e documentada desde as primeiras versões do MRST, sem perspectiva de grandes mudanças para as demais versões. Nessa família ainda pode-se destacar o módulo *adjoint*, que implementa estratégias de otimização baseada na formulação adjunta para escoamento bifásicos e incompressíveis.

A terceira família de módulos envolve **Solvers Implícitos baseados em diferenciação automática**. Adiante serão trazidos detalhes a respeito da funcionalidade da diferenciação automática. De antemão cabe adiantar que ela busca simplificar o processo de implementação dos simuladores uma vez que as derivadas e jacobianas são calculadas automaticamente. Cabe mencionar que, inicialmente, ela foi utilizada em conjunto com a programação procedural, porém, isso se tornou um fator limitante. Portanto, foi desenvolvida uma nova estrutura com programação orientada objeto, denominada por MRST AD-OO. Após dois anos de mudanças

no código, a partir da versão 2016a, a funcionalidade básica da diferenciação automática foi consolidada, necessitando a partir de então de correções de erros e melhorias de desempenho (Lie, 2016). Contudo, cabe mencionar que, durante o desenvolvimento do presente trabalho, foram identificadas alterações significativas em algumas classes constituintes do MRST AD-OO ao comparar duas versões diferentes. A versão adotada para estudo foi MRST 2018b. Na seção 3.1.1 apresenta-se a estrutura geral do MRST AD-OO. Destaca-se que partir da programação orientada a objeto, busca-se realizar simulação de modelos compressíveis multifásicos similares aos presentes na indústria de petróleo.

A quarta família de módulos é denominada de ***Workflow Tools*** e pode ser utilizada antes ou depois da simulação do reservatório como parte da modelagem. Pode ser destacado o módulo denominado *diagnostics,* que proporciona ferramentas para a determinação de conexões volumétricas em reservatório, cálculo de fatores de alocação de poço, medida de heterogeneidade dinâmica e fornecimento de estimativas simplificadas de recuperação, dentre outros. Há também os módulos *upscaling* e *steady-state* que permitem a realização da simulação de modelos reduzidos com menos graus de liberdade e menor custo computacional. Cabe destacar que *steady-state* traz o *upscaling* da permeabilidade relativa com base na suposição do estado estacionário.

Além disso, há uma categoria adicional referente aos **Métodos Multiescala.** Esses métodos podem ser usados como um sofisticado método de *upscaling* para produzir informações de fluxo e pressão na malha grossa ou como um *solver* na escala fina de forma aproximada. Os módulos pertencentes a essa categoria são: *msmfe - multiscale mixed finite-element, msfv - multiscale finite-volume* e *mrsb - multiscale restricted-smoothing basis (MsRSB) method.*

A sexta família traz ferramentas de **Simulação Especializadas para Problemas Específicos**, como estudos da injeção e migração de CO^2 em grandes aquíferos subterrâneos, geomecânica e simulação de escoamento em meio fraturado. Pode-se citar *co2lab, dfm, dual-porosity, fvbiot, geochemistry, hfm* e *vemmech*.

A sétima categoria destacada consiste em uma variedade de módulos associado à **Interface Gráfica e Visualização Avançada**, além de rotinas para leitura e processamento de modelos de simulação e outros dados de entrada. Dessa forma, ela fornece funções utilidades para outros módulos. Pode-se citar o *mrst-gui*, para criar interface interativa de visualização; *octave*, que permite a compatibilidade com GNU Octave; *matlab_bgl*, para download do

MATLAB *Boost Graph Library; book*, com todos os códigos usados em exemplos do livro; *streamlines*, método para traçar *streamline* em malhas cartesianas ou curvilíneas; *wellpaths*, para definir poços seguindo trajetórias curvilíneas e *spe10*, para fazer download, converter e carregar informação do SPE10 para o MRST.

Por fim, há ainda um grupo de **módulos que ainda não são parte da versão oficial**, mas estão desenvolvidos, como *enkf* (Ensemble Kalman filter); *remso*, otimização baseada em múltiplas escolhas e *mrst-cap*, implementa um escoamento composicional com pressão capilar (LIE, 2016)

Todos módulos apresentados, exceto a última categoria, estão disponíveis publicamente na página do *software* (MRST, 2020). Todos como regra geral, devem possuir um conjunto de tutoriais, apresentando exemplos de suas funcionalidades chaves.

É importante ressaltar que este trabalho se concentrará principalmente nas funcionalidades básicas do núcleo e na família de módulos AD-OO para prototipagem rápida de simuladores totalmente implícitos, que será detalhada a seguir.

3.1.1 Estrutura do MRST AD - OO

Conforme mencionado anteriormente, a nova estrutura desenvolvida no MRST é baseada na programação orientada a objetos (POO) e utiliza *solvers* baseados em Diferenciação Automática (AD). A seguir são apresentadados os principais módulos que o compõem.

Um dos principais módulos do MRST AD-OO, é o *ad-core*. Esse módulo não contém nenhum simulador, mas implementa toda estrutura comum dos módulos MRST AD-OO, incluindo classes abstratas para modelos do reservatório, *solvers* não linear e linear, dentre outros.

Ainda podem-se destacar os seguintes módulos:

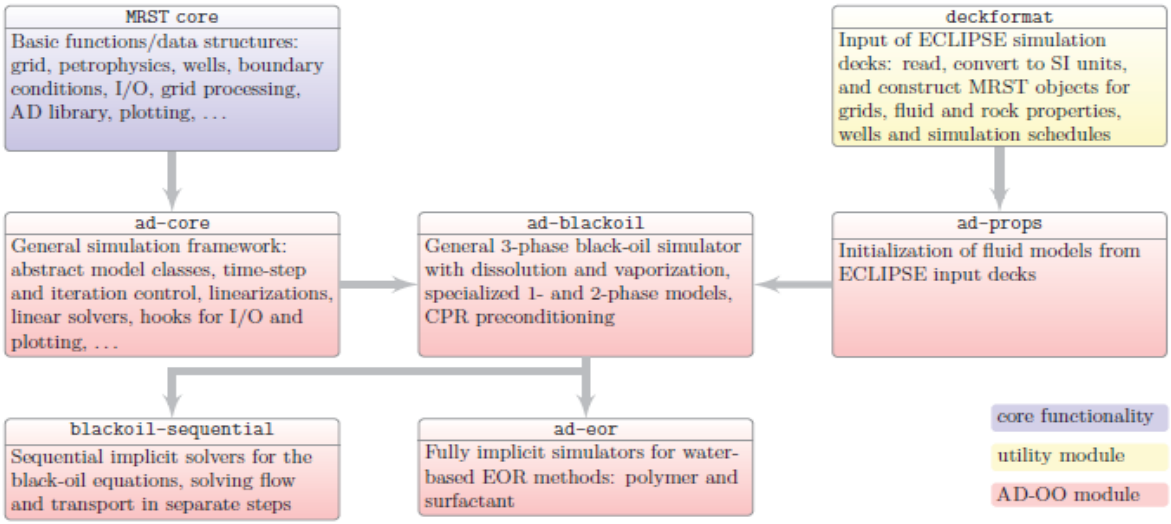
- *deckformat*, que possibilita converter a entrada de dados no formato do ECLIPSE para os objetos do MRST;
- *ad-props*, relacionado ao cálculo das propriedades como as relacionadas aos fluidos para o *ad-core*;
- *ad-blackoil*, estende a estrutura encontrada em *ad-core* para problemas *black-oil;*

- *blackoil-sequential*, destinado a resolver as mesmas equações do modelo *blackoil* utilizadas no módulo *ad-blackoil* porém de forma sequencial, ou seja, as equações de pressão e saturação são resolvidas separadamente;
- *ad-eor* estende as funcionalidades do *ad-core* e *ad-blackoil* para a simulação da recuperação aprimorada de petróleo (*Enhanced Oil Recovery - EOR)*, também de recuperação terciária, caracterizada pela adição de diferentes químicos (polímeros e surfactante);
- *compositional*, oferece *solver*s para problemas composicionais;
- *solvents*, extensão para modelos miscíveis sem utilizar o custoso módulo composicional;
- *ad-mechanics*, inclui a modelagem mecânica para elasticidade linear que pode ser acoplada ao modelo padrão de escoamento de fluido;
- *optimization*, inclui rotinas para solução de controles ótimos, utilizam rotinas quase Newton com atualização das hessianas por BFGS, mas pode ser facilmente adaptado para outros otimizadores.

Cabe destacar ainda que o antigo módulo *ad-fi*, correspondente a primeira implementação dos *solvers* AD para modelos *blackoil*, foi atualmente desativado.

Os módulos relacionados ao modelo *black-oil* estão destacados na Figura 7. No presente trabalho, serão utilizados principalmente as funcionalidades dos módulos *ad-core*, *ad-blackoil* e *ad-props*.

Figura 7 – Módulos da estrutura AD-OO



Fonte: Lie (2016).

3.2 INTRODUÇÃO À UTILIZAÇÃO DO MRST

Após a compreensão do que é e como está estruturado o MRST, é importante que seja aqui apresentado como se procede a utilização dessa ferramenta. Portanto, além das instruções para *download* do *software*, nas seções posteriores, serão descritas as estruturas internas e comuns para a simulação, bem como, exemplos básicos para a simulação de um modelo, destacando os principais passos para a compreensão de seu funcionamento.

3.2.1 Download e instalação do MRST

A primeira etapa para a utilização do MRST consiste na realização de seu download. Ele dever ser realizado na seguinte webpage: http ://www.sintef.no/MRST/, conforme ilustra a Figura 8.

Essa página disponibiliza todas as versões do MRST. A cada semestre é disponibilizada uma nova atualização. Para o desenvolvimento desse trabalho, foi utilizada a versão 2018b, visto que essa foi a versão mais recente durante o período do trabalho em questão.

Figura 8 – Download do MATLAB Reservoir Simulation Toolbox

Fonte: MRST (2020).

Realizado o download da versão desejada, será obtido um arquivo zipado que deverá ser extraído no diretório de trabalho de sua preferência. Em seguida, deve aberta a interface do

MATLAB ou do GNU Octave nesse mesmo diretório. Cabe destacar que o MRSR é compatível com ambos os softwares. Dessa forma, uma vez estando na pasta raiz do MRST, deve-se ativar o software utilizando o seguinte comando:

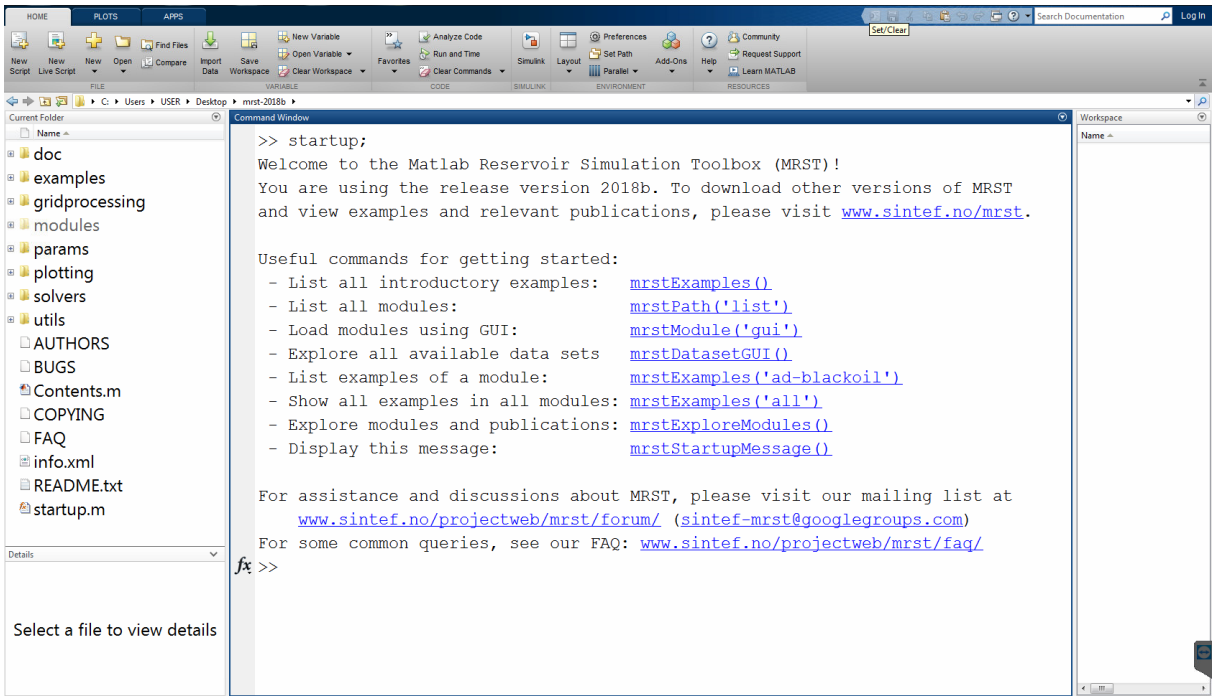
Algoritmo 1 – Trecho de código para Ativação do MRST

```
1.    startup; % or  run C:\MyPath\mrst-2018b\startup.
2.
```

Observe que uma segunda possibilidade, caso não queira mudar o diretório até o local da raiz é utilizar o comando comentado no *snippet* (trecho de código) acima.

A Figura 9 apresenta a pasta núcleo do MRST e o comando de ativação do software. Note que irá aparecer uma mensagem inicial, mostrando que o software está pronto para uso.

Figura 9 – Interface da pasta raiz do MRST e comando de ativação



Fonte: Autora (2020).

3.2.2 Comandos usuais do MRST

Primeiramente, cabe destacar que um dos comandos mais recorrentes, para compreender o funcionamento das diversas funções presentes no MRST é o comando `help`, utilizado conforme exemplificado a seguir.

Algoritmo 2 – Comando Help

```
%Apresenta a documentação da função computeTrans.
help computeTrans;
```

Em seguida, destacam-se alguns comandos presentes na mensagem inicial, apresentada na Figura 9. O Algoritmo 3 exemplifica a utilização desses comandos:

Algoritmo 3 – Comandos usuais

```
% Lista exemplos do MRST core
mrstExamples();

%Lista exemplos do módulo especificado
mrstExamples('ad-blackoil');

% Lista todos os exemplos
mrstExamples('all')

%Interface gráfica com a listagem, descrição, e exemplos da cada
% módulo
mrstExploreModules()

% Carrega o módulo através interface gráfica
mrstModule('gui') OU
mrstModule gui OU
moduleGUI

% Ativa ou desativa o módulo da lista
mrstModule <add|clear|reset> [module list]
% Ex: mrstModule add mimetic mpfa

% Identifica as pastas que são módulos
 mrstPath

% Inclui um módulo ao MRST
mrstPath register[name][path]
Ex: mrstPath register AGMG C :\mrst\modules\agmg

% Interface gráfica para ter acesso a conjuntos de dados de campos
referências
mrstDatasetGUI()
```

Os três primeiros comandos, apresentados no Algoritmo 3, são variedades da utilização do `mrstExamples`, sendo responsável por fornecer acesso aos exemplos presentes tanto no *MRST core*, como nos demais módulos adicionais.

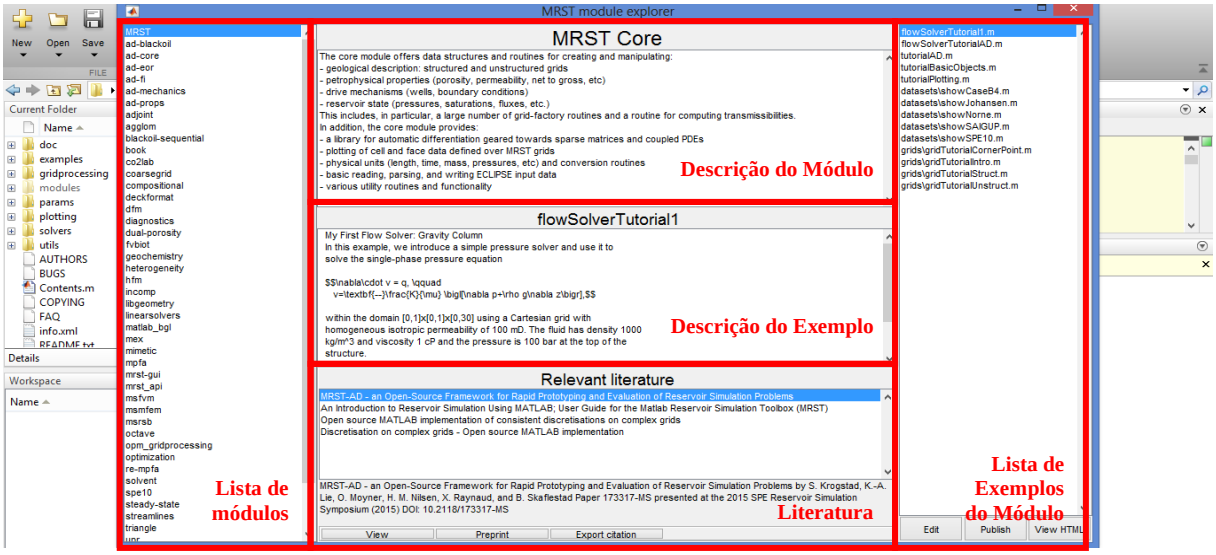
No que diz respeito aos módulos, existem dois comandos usuais: `mrstExploreModules` e `mrstModule`. O primeiro traz uma interface gráfica, dispondo da lista, descrição e exemplos de todos os módulos, bem como o acesso à versão online e à importantes publicações na área. A Figura 10 ilustra essa interface.

O segundo comando, `mrstModule`, é imprescindível para a utilização do software. Ele é responsável pela ativação ou desativação dos módulos presentes no MRST. Dessa forma, o módulo é adicionado ou removido do caminho de pesquisa do Matlab. Para todo exemplo em que se utilizar algum comando de um módulo específico, deve-se ativar esse módulo para que o comando funcione. Ele deve ser usado conforme exemplificado na linha 19 do Algoritmo 3, seguido dos comandos `add`, `clear`, `reset` e dos nomes dos módulos de interesse.

Para mapear todos os módulos reconhecidos pelo MRST e, portanto, possíveis de ser ativados ou desativados conforme explicado acima, deve-se utilizar o comando `mrstPath`. Esse comando também pode ser utilizado para acrescentar um módulo adicional, ou seja, para que o software reconheça uma pasta como módulo, conforme exemplificado na linha 26 do Algoritmo 3.

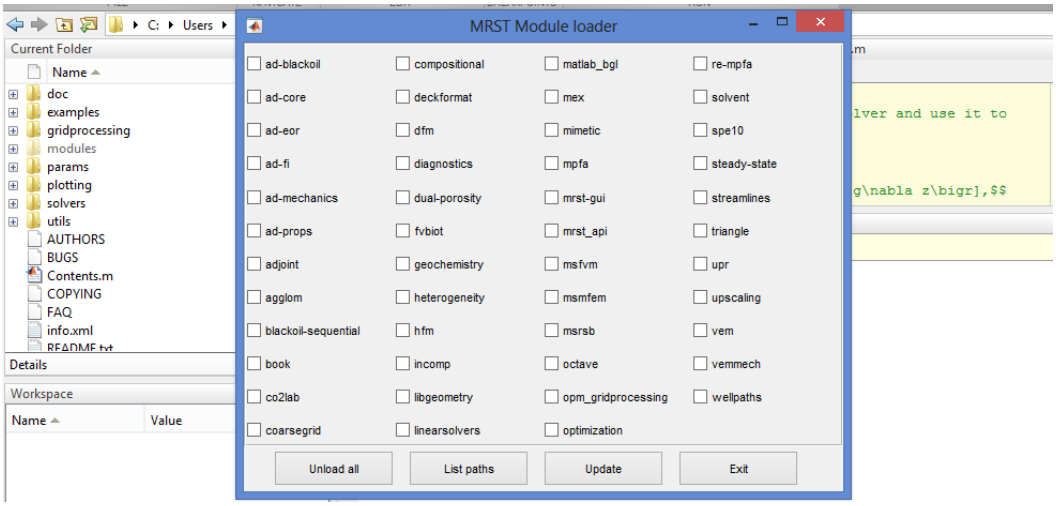
Além disso, também é possível fazer essa ativação ou desativação através de interface gráfica, através do comando `mrstModule gui` ou `moduleGUI`, conforme indica a Figura 11.

Figura 10 – Interface Gráfica gerada pelo comando mrstExploreModules



Fonte: A autora (2020).

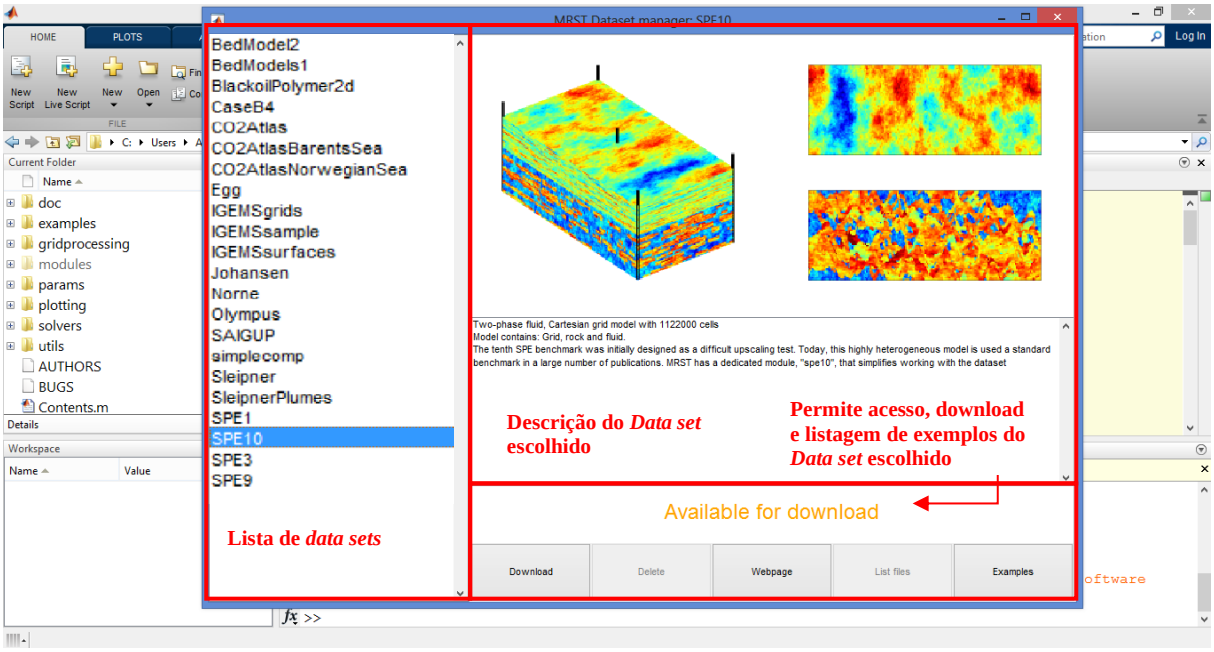
Figura 11 – Interface Gráfica gerada pelo comando moduleGUI



Fonte: A autora (2020).

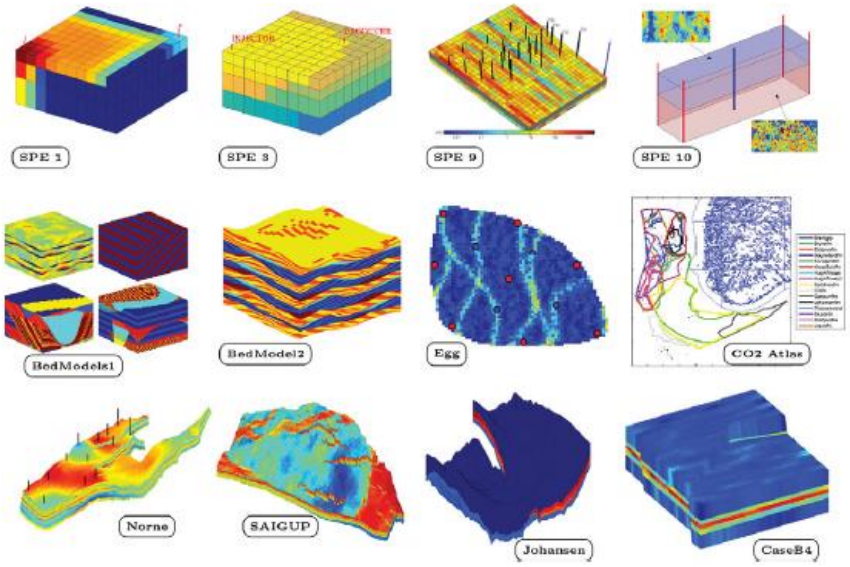
O MRST também oferece uma interface gráfica para facilitar o acesso ao conjunto de dados públicos de campos referências, comumente utilizados para estudos na área de petróleo. Esse comando é denominado por `mrstDatasetGUI`. Ele cria uma interface gráfica que permite o acesso à lista de conjuntos de dados, a descrição e download rápido dos dados, conforme ilustra a Figura 12. Dessa forma, é possível testar os novos métodos computacionais em campos relevantes, realísticos e referências na área de petróleo com um acesso simples e prático. A Figura 13 ilustra a interface de alguns módulos

Figura 12 – Interface Gráfica gerada pelo comando mrstDatasetGUI.



Fonte: A autora (2020).

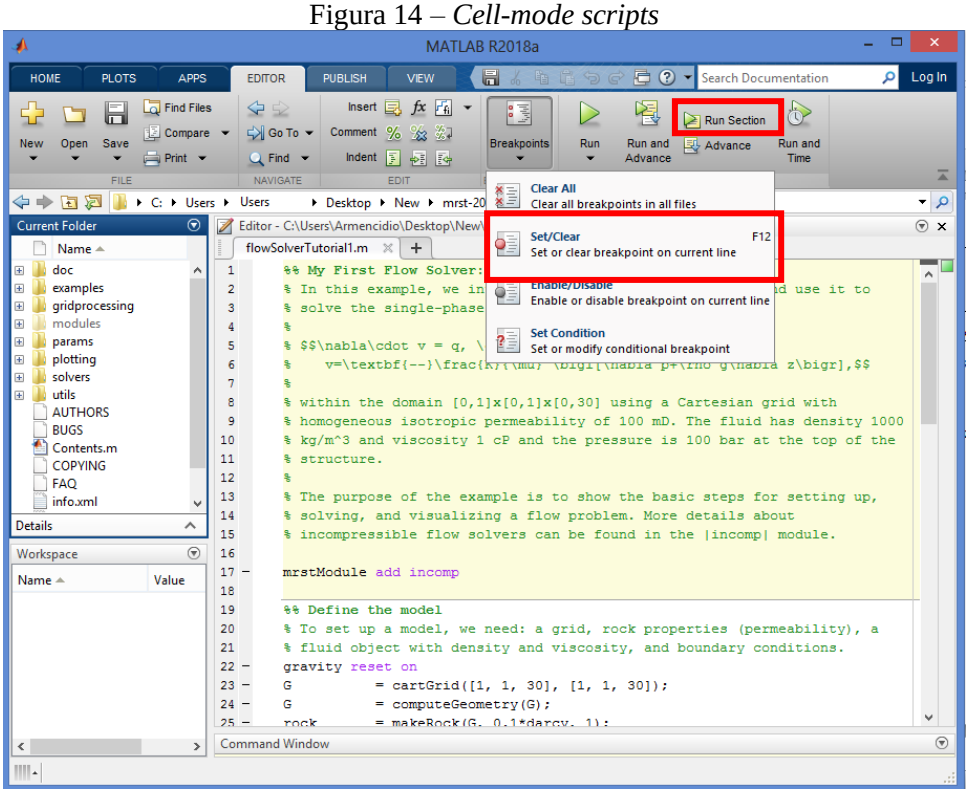
Figura 13 – Ilustração de alguns *datasets* disponíveis



Fonte: MRST (2020).

Cabe destacar que a escrita dos códigos utiliza *cell-mode scripts*, ou seja, o código é dividido em pequenas seções que podem ser executadas individualmente, tanto utilizando o comando *Run Section* ou *Set Breakpoints* no início de cada seção, conforme indica a Figura 14.

Figura 14 – *Cell-mode scripts*



Fonte: A autora (2020).

3.3 GERAÇÃO E ESTRUTURA DE MALHAS

A primeira seção importante para a simulação computacional de um modelo é a geração de malhas, com a descrição de sua geometria e discretização do modelo.

Uma malha nada mais é que um conjunto de células que se distinguem umas das outras pela sua geometria (forma das células) e topologia (conexão das células).

Essa é das partes mais importantes a ser compreendida no MRST, uma vez que praticamente todos os *solvers*, *workflow tools* e rotinas de visualização requerem uma malha como dado de entrada.

No MRST, por convenção, a estrutura da malha é denominada por G. Destaca-se que ela foi construída prezando pela generalidade, os desenvolvedores do software optaram por utilizar uma estrutura geral, capaz de armazenar informações no formato de malha não estruturada, onde os dados sobre as células, faces, vértices e conexões entre elementos são explicitamente representados. Dessa forma, a eficiência foi sacrificada, focando na flexibilidade para a representação da malha. (LIE, 2016)

Acrescenta-se que no MRST algumas informações requeridas para esquemas de discretização em simulação de fluxo, tais como, volumes, centroides, normal da face, dentre outros, podem ser facilmente incluídas explicitamente na representação de malhas, em virtude da sua constância de uso.

Inicialmente será explicado como gerar uma malha cartesiana simples no tamanho de 3x3x3, utilizada como exemplo ao decorrer das explicações, para facilidade de compreensão das estruturas e objetos do MRST.

Porém, também serão apresentados casos mais complexos, permitindo ao leitor uma visão completa das funcionalidades do software e as possibilidades de extensão e trabalhos futuros no mesmo.

3.3.1 Criação Malha Estruturada Cartesiana

Malhas cartesianas estruturadas discretizam o domínio em elementos possuindo conectividade implícita, ou seja, apenas com as coordenadas dos nós destes obtêm-se todas as relações de conectividade existentes, o que acelera o processo de solução (BATISTA, 2005).

Para gerar a malha cartesiana, deve-se fornecer o número de elementos em cada direção n_i e a extensão L_i de cada dimensão, com i = x,y,z, através do comando `cartGrid`, conforme apresentado no Quadro 1. Destaca-se que o retorno dessa estrutura será mais bem detalhado no tópico seguinte.

Quadro 1 – Malha cartesiana - cartGrid

FUNÇÃO SINTAXE:	
`G = cartGrid([nx,ny,nz],[Lx,Ly,Lz]); para malha cartesiana 2D` `G = cartGrid([nx,ny],[Lx,Ly]); para malha cartesiana 3D`	
	PARÂMETROS DE ENTRADA:
nx,ny,nz –	Número de elementos em cada direção.
Lx,Ly,Lz –	Extensão L em cada direção.
	RETORNO:
G-	Estrutura da malha, com campos: ***cells; faces; nodes; cartDims; type; griddim**

Fonte: Adaptado do MRST (2018).

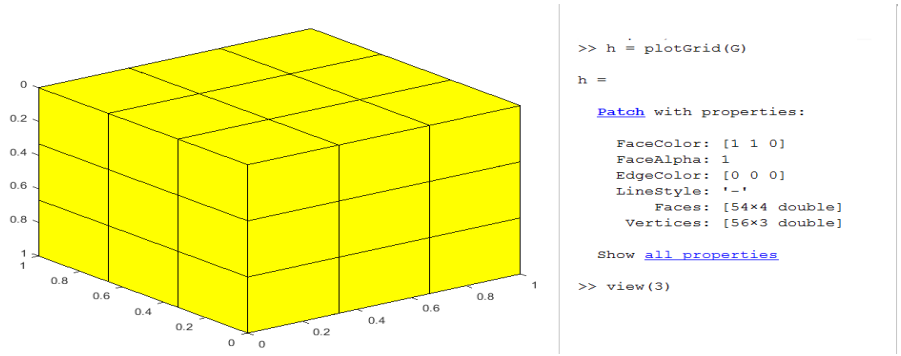
Dessa forma, para gerar uma malha cartesiana com 3 elementos em cada direção (n_x=n_y=n_z=3) com dimensões de 1 metro (L_x=L_y=L_z=1), deve-se escrever:

Algoritmo 4 – Exemplo para gerar Malha cartesiana

```
1.  % Cria a estrutura G para uma malha Cartesiana 3D
2.  G = cartGrid([3,3,3],[1,1,1]);
3.
4.  % Plotar a malha
5.   plotGrid(G)
6.
7.  % Permite a visualização 3D
8.   view(3)
```

Note que, se apresenta acima, o comando para visualização da malha, `plotGrid`, gerando a visualização apresentada na Figura 15.

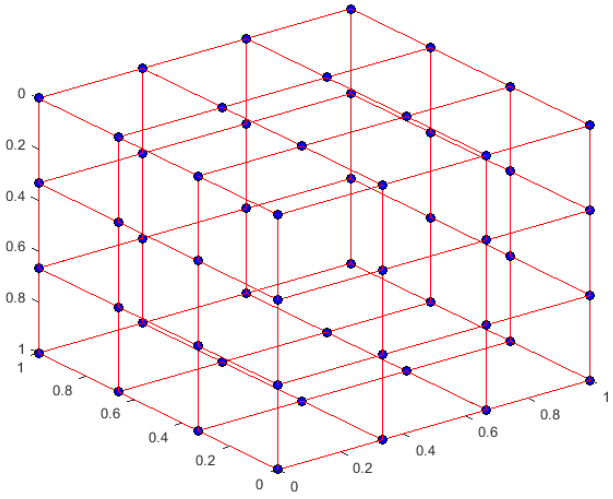
Figura 15 – Uso do comando plotGrid no exemplo base



Fonte: A autora (2020).

Observe que as propriedades dessa plotagem podem ser alteradas, de acordo com a intenção do usuário. Na Figura 16 é ilustrado como essa alteração pode ser realizada.

Figura 16 – Alterações de propriedade do plotGrid




```
>> h = plotGrid(G,'FaceColor', 'none',...
    'EdgeColor', 'red','LineWidth',0.8,...
    'Marker','o','MarkerFaceColor','blue',....
    'MarkerEdgeColor','black');
>> view(3)
```

Fonte: A autora (2020).

3.3.2 Estrutura Interna da Geometria

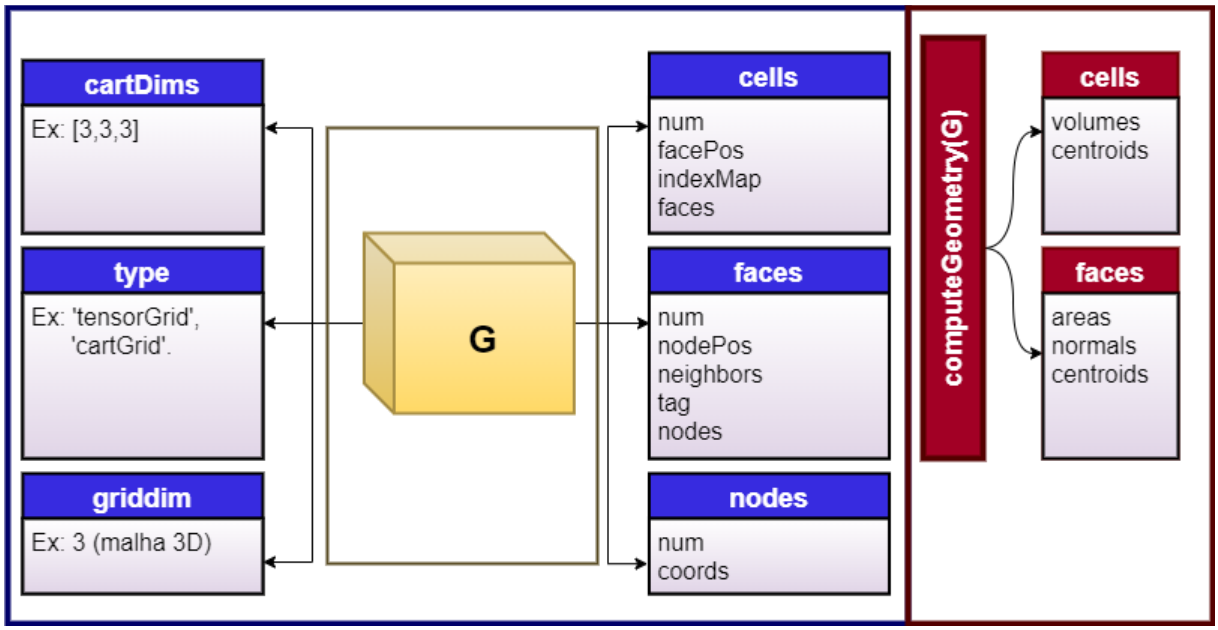
Nessa seção, será apresentada a estrutura interna gerada no MRST para a representação das malhas. Cabe ressaltar que esse tópico é essencial para quem pretende realizar alguma nova implementação no MRST.

Conforme mencionado anteriormente, essa estrutura é composta de modo similar para todos os tipos de malhas, ou seja, estruturadas e não estruturadas. A estrutura principal, denominada de G, contém seis campos. O três primeiros, `cells`, `faces` e `nodes`, especificam propriedades para cada célula, face e nó da malha, respectivamente. O campo `griddim` distingue uma malha 2D (células são polígonos, `griddim=2`) e 3D (células são entidades volumétricas poliédricas, `griddim=3`). Há ainda um campo, denominado `type`, composto por um conjunto de *strings* que descreve o histórico do construtor das malhas e as funções modificadoras aplicadas para criá-las. Por fim, no caso de malhas com estrutura cartesiana, existe um campo chamado de `cartDims`, que especifica o tamanho da malha.

É importante lembrar que mais informações, podem ser encontradas em sua documentação, através do comando `help grid_structure` ou `doc grid_structure`.

Cada um dos campos `cells`, `faces` e `nodes`, são compostos por outros campos, ilustrados na Figura 17. Adiante será detalhado cada um desses campos.

Figura 17 – Estrutura interna de G



Fonte: A autora (2020).

A estrutura `G.cells` contém obrigatoriamente os campos `num`, `facePos`, `indexMap` e `faces`, detalhados a seguir.

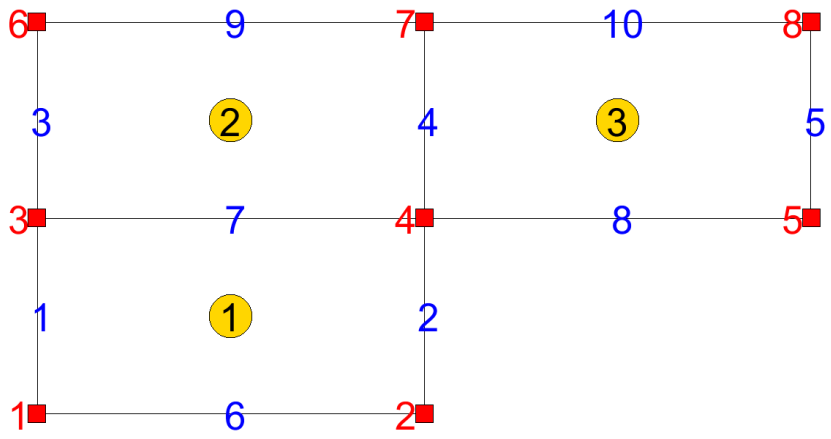
- `num`: número de células na malha (n_c). Para a malha 3x3x3 gerada na Figura 15, esse número é igual a 27.

- `facePos`: mapeamento indireto das faces em um vetor, com tamanho [n_c+1,1], onde n_c = número de células.

  Nesse ponto, cabe explicar o que consiste um mapeamento indireto, bastante utilizado no MRST. Para exemplificar, observe a Figura 18 e a Figura 19. Sendo *a* o conjunto de numeração das faces e *b* o agrupamento das faces pertencentes a cada elemento, tem-se *a* = [1,6,2,7,3,7,4,9,4,8,5,10] e b = [1,1,1,1,2,2,2,2,3,3,3,3]. Realizando um mapeamento de indireto para encontrar rapidamente o valor de *a* pertencente a um particular b, ou seja, a que célula cada face pertence, obtém-se *m* =[1,5,9,13]. O vetor *m* é obtido adicionando o número de elementos a partir de 1. Note que poderia existir uma coluna extra em *a*, com os valores de b, no entanto, isso seria uma informação redundante que pode ser guardada mais resumidamente da seguinte forma: vetor b = [1,2,3] e a repetição n = [4,4,4]. Essa técnica de comprimir e descomprimir a informação é muito utilizada no MRST para evitar armazenamento de informação redundante.

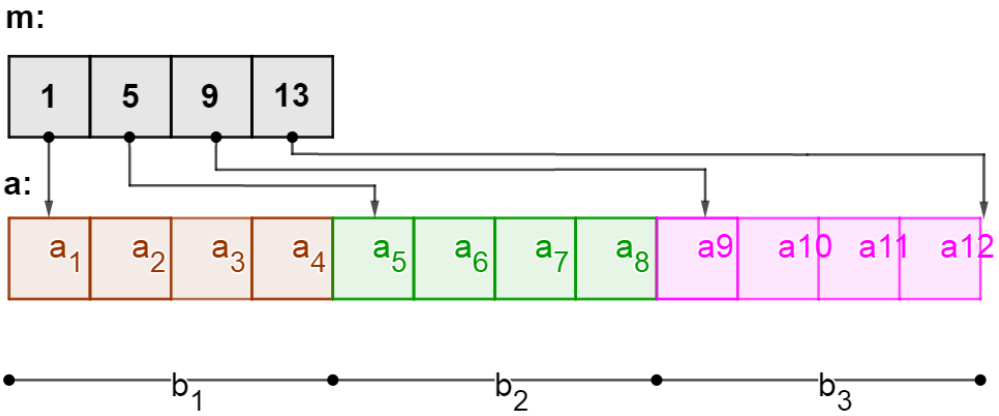
Nesse contexto, dada uma malha qualquer, o campo `facePos` será um vetor onde: e_1=1; e_2= e_1+$n_f(e_1)$; e_3= e_2+$n_f(e_2)$, e assim sucessivamente, onde e_1=elemento 1 e $n_f(e_1)$ = número de faces do elemento 1. Dessa forma, para a malha da Figura 15, como o número de face de cada elemento é 6 e o número de células é 27, tem-se o vetor, `G.cells.facePos = [1; 7; 13; 19; 25; 31; 37; 43; 49; 55; 61; 67; 73; 79; 85; 91; 97; 103; 109; 115; 121; 127; 133; 139; 145; 151; 157; 163]`$_{28x1}$.

Figura 18 – Mapeamento 2D



Fonte: A autora (2020).

Figura 19 – Mapeamento indireto



Fonte: A autora (2020).

Existem alguns comandos recorrentes com a utilização de `facePos`, conforme disposto abaixo.

Algoritmo 5 – Comandos Recorrentes com facePos

```
1.
2.  %% Informação das faces para determinada célula:
3.  %Obtém a informação das faces da célula i, ou seja, os valores de a
4.  % para um determinado b, conforme nomenclatura da Figura 19.
5.  i = 1;
6.
7.  G.cells.faces(G.cells.facePos(i): G.cells.facePos(i+1)-1,:);
8.  %Para o exemplo base:
9.  %G = cartGrid([3,3,3],[1,1,1]), com i=1,
10. %tem-se: [1,1; 2, 2; 37, 3; 40, 4;  73, 5;  82, 6]. Vide Figura 20.
11.
12. %A segunda coluna indica as direções das faces, conforme apresentado
13. %adiante
14. %% Número de faces de cada célula:
15.
16. diff(G.cells.facePos)
17. %Para o exemplo base –
18. %G = cartGrid([3,3,3],[1,1,1]), tem-se: [6; 6;...;6]27x1
19. %% Número total das faces, incluindo sobreposições:
20.
21. nfs=G.cells.facePos(end)-1;
22. %Para o exemplo base:
23. %G = cartGrid([3,3,3],[1,1,1]), nfs = 162
```

- `indexMap`: Consiste em um vetor que mapeia os índices internos das células para os índices externos. Para modelos sem células inativas, como no exemplo base, o índice é igual ao número de células, porém o mesmo não ocorre quando elas existem. Para o exemplo da Figura 18, onde a célula na posição 2 foi suprimida, `G.cells.indexMap = [1;3;4]`.

  Para malhas cartesianas, utilizando `indexMap`, pode ser construído um mapeamento com índices lógicos, conforme, apresentado abaixo.

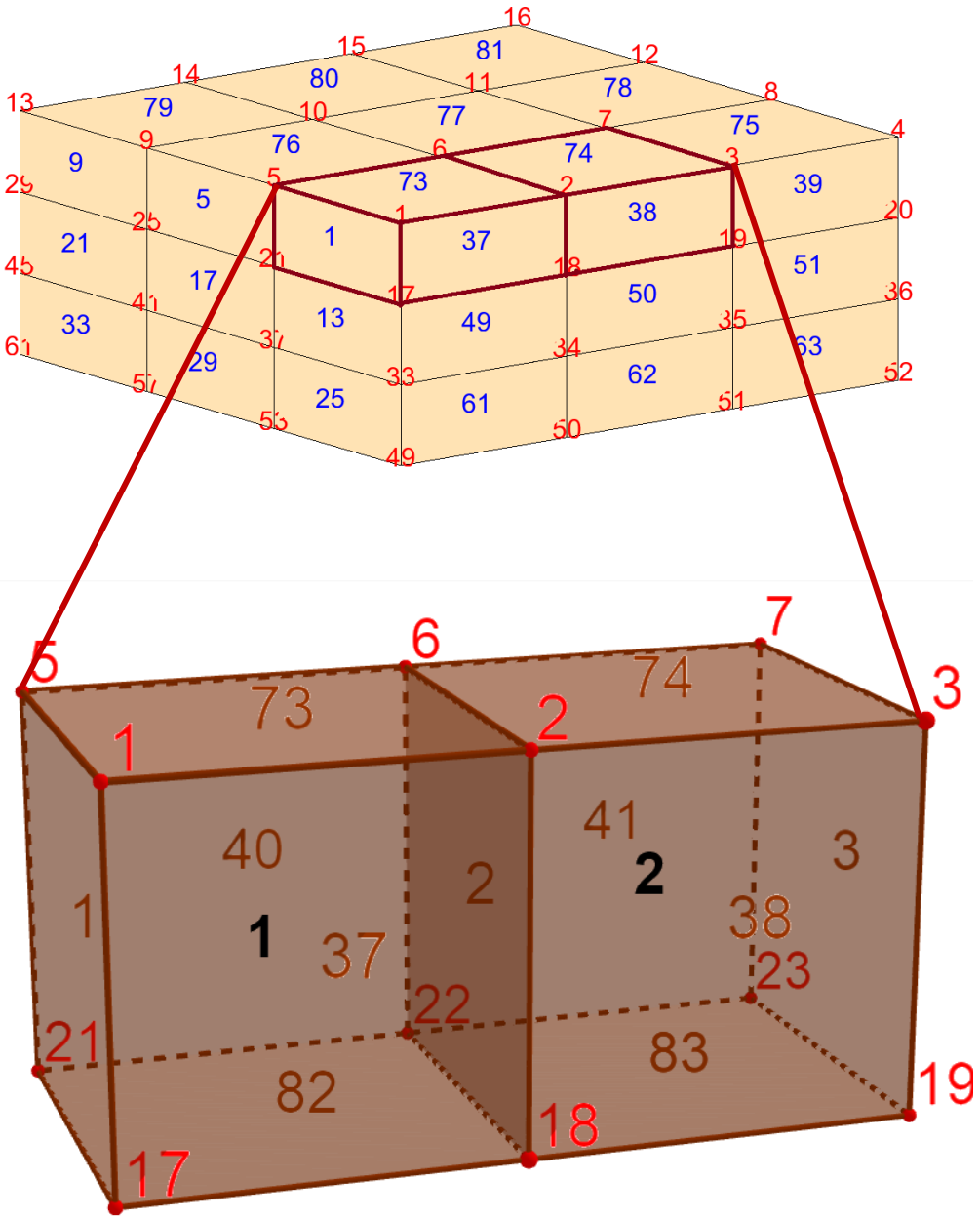
Algoritmo 6 – Mapeamento para construção de índices lógicos

```
1.  %% Índices lógicos para malhas 2D
2.  %ind2sub é usado para determinar os valores subscritos (i,j)
3.  %equivalentes a um determinado índice.recebe como argumento o tamanho
4.  %e a matriz de índices.
5.
6.  [ij{1:2}] = ind2sub(G.cartDims, G.cells.indexMap(:)); ij = [ij{:}];
7.  % Em 2D, exemplo da  Figura 18:[1 1; 1 2, 2 2]
8.
9.  %% Índices lógicos para malhas 3D
10.
11. [ijk{1:3}] = ind2sub(G.cartDims, G.cells.indexMap(:));ijk = [ijk{:}];
```

- `faces`: um vetor com dimensão n_{fs} (número total de faces com sobreposições) x 2, especificando as faces pertencentes a uma determinada célula. A primeira coluna corresponde ao índice global da face, pertencente à determinada célula. A segunda coluna

distingue as direções das faces: 1 - Oeste, 2 - Leste, 3 - Sul, 4 - Norte, 5 - Acima, 6 - Abaixo. Observando a Figura 20 tem-se `G.cells.faces = [1 1;2 2;37 3;40 4;73 5;82 6;2 1;3 2;38 3;41 4;74 5;83 6; ...]`$_{162x2}$. Poderia ainda haver uma coluna com a informação das células que abrangem cada face do vetor. Porém, para evitar redundância essa coluna é suprimida podendo ser reconstruída pelo Algoritmo 7.

Figura 20 – Numeração do cubo MRST – Exemplo Base



Fonte: A autora (2020).

Algoritmo 7 – Reconstrução da coluna inicial de G.cells.faces

```
1.  %rldecode - Descomprimir a codificação do comprimento de execução da
2.  %matriz 'A' ao longo da dimensão 'dim'. B = rldecode(A, n, dim).
3.  %% Cria coluna indicando a que célula cada face pertence:
4.  f2cn = rldecode(1:G.cells.num, diff(G.cells.facePos), 2)';
5.
6.  %% OU
7.
8.  f2cn = gridCellNo (G);
9.  % Para Figura 20 (exemplo base), tem-se:
10. % [1,1,1,1,1,1,2,2,2,2,2,2...27,27, 27,27,27,27].
```

Além dos campos obrigatórios, após o comando `computeGeometry` são incorporados os seguintes campos em `G.cells`:

- `volumes`: com dimensão de n_c x 1 constando do volume de cada célula. Para o exemplo base, esse valor será dado por $(1/3)^3 = 0.0370$, para todas as células.

- `centroids`: um vetor n_c x d (dimensão da malha) constando do centroide de cada célula. No exemplo base, tem-se:

$$G.cells.centroids = \begin{bmatrix} 0.1667 & 0.1667 & 0.1667 \\ 0.5000 & 0.1667 & 0.1667 \\ 0.8333 & 0.1667 & 0.1667 \\ & \cdots & \end{bmatrix}_{27x3} \quad (1)$$

No que diz respeito às faces, tem-se a estrutura `G.faces` contendo obrigatoriamente os campos `num`, `nodePos`, `neighbors`, `tag` e `nodes`, detalhados a seguir.

- `num`: número global de faces (n_f) na malha (sem sobreposição). Para o exemplo base, `G.faces.num=108`.

- `nodePos`: mapeamento indireto dos nós em um vetor, com tamanho [n_f+1,1], onde n_f = número global de faces. Para o Exemplo Base (Figura 20), visto que cada face possui 4 nós, `G.faces.nodePos=[1;5;9;13;...;429;433]`$_{109x1}$. No Algoritmo 8 estão apresentados os principais comandos relacionados a `nodePos`.

Algoritmo 8 – Comandos Recorrentes com nodePos

```
1.   %% Informação dos nós para determinada face:
2.   %Obtém a informação dos nós da face i, ou seja, os valores de a
3.   %para um determinado b, conforme nomenclatura da Figura 19.
4.
5.   i = 1; G.faces.nodes(G.faces.nodePos(i): G.faces.nodePos (i+1)-1,:);
6.   %Para o exemplo base:
7.   %G = cartGrid([3,3,3],[1,1,1]), com i=1, tem-se: [1; 5; 21; 17].
8.   %Vide Figura 20.
9.
10.
11.  %% Número de nós de cada face:
12.  diff(G.faces.nodePos)
13.  %Para o exemplo base:
14.  %G = cartGrid([3,3,3],[1,1,1])
15.  %tem-se:[4; 4;...;4]108x1
16.
17.
18.  %% Número total de nós, incluindo sobreposições:
19.  nns=G.faces.nodePos(end)-1;
20.  %Para o exemplo base:
21.  %G = cartGrid([3,3,3],[1,1,1]), nns = 432.
22.
```

- `neighbors`: como o próprio nome diz, trata-se de um vetor que armazena a informação da vizinhança. Sua dimensão é dada por n_f x 2. Para uma determina face global i, sua vizinhança possui duas células: `G.faces.neighbors(i,1)` e `G.faces.neighbors(i,2)`. Se algum desses valores for zero, isso significa que a face é externa e pertence somente a uma célula. Observando a Figura 20, tem-se, por exemplo: `G.faces.neighbors(1,:) = [0,1]; G.faces.neighbors(2,:) = [1,2].`

- `tag`: vetor inteiro n_f x 1, para definição do usuário de indicadores da face, por exemplo, face que pertence a uma determinada célula.

- `nodes`: vetor com dimensão dada por: n_{ns} (número total de nós com sobreposição) x 1. Armazena o índice global do nó pertencente a cada face. Para a Figura 20 tem-se, `[1;5;21;17;2;6;22;18;3;7;23;19 ...]`$_{432x1}$.

  Poderia haver uma coluna anterior, expressando o número da face que cada nó pertence. Porém, a fim de evitar redundância, essa informação é suprimida, podendo ser facilmente obtida, conforme expressa o Algoritmo 9.

Algoritmo 9 – Reconstrução da primeira coluna de G.faces.nodes

```
1.
2.  %% Coluna indicando a que face cada nó
3.  % presente em G.faces.nodes pertence:
4.  n2fc = rldecode(1:G.faces.num, diff(G.faces.nodePos), 2)';
5.
6.  % Para Figura 20 (exemplo base), tem-se:
7.  % [1,1,1,1,2,2,2,2...108,108, 108,108]432x1.
8.
```

Após o comando `computeGeometry` são incorporados os seguintes campos em `G.faces`:

- `áreas`: com dimensão de n_f x 1 constando da área de cada face. Para o exemplo base, esse valor será dado por $(1/3)^2 = 0.1111$, para todas as faces.
- `normals`: matriz com dimensão de n_f x d (dimensão da malha) constando das áreas das faces, direcionadas em P^d. A direção da normal de uma face i, aponta da célula `G.faces.neighbors(i,1)` para `G.faces.neighbors(i,2)`.

  Para o exemplo base, tem-se a seguinte matriz:

$$G.faces.normals = \begin{bmatrix} 0.1111 & 0 & 0 \\ 0.1111 & 0 & 0 \\ 0.1111 & 0 & 0 \\ \cdots & & \end{bmatrix}_{108x3} \tag{2}$$

- `centroids`: matriz com dimensão de n_f x d (dimensão da malha) constando dos centroides das faces em P^d.

$$G.faces.centroids = \begin{bmatrix} 0 & 0.1667 & 0.1667 \\ 0.3333 & 0.1667 & 0.1667 \\ 0.6667 & 0.1667 & 0.1667 \\ & \cdots & \end{bmatrix}_{108x3} \tag{3}$$

Por fim, em relação aos nós, tem-se a estrutura `G.nodes`, contendo obrigatoriamente os seguintes campos:

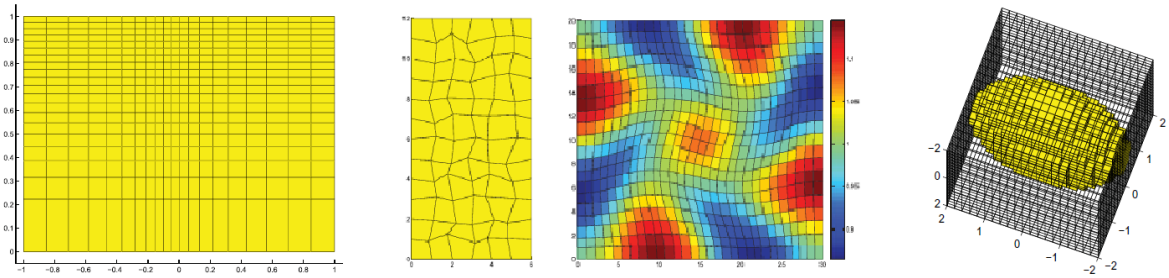
- `num`: número global de nós (n_n) na malha (sem sobreposição). Para o exemplo base, `G.nodes.num=64`.
- `Coords`: matriz com dimensão de n_n x d (dimensão da malha) constando das coordenadas de cada nó em P^d. Para o exemplo base, tem-se:

$$G.nodes.coords = \begin{bmatrix} 0 & 0 & 0 \\ 0.3333 & 0 & 0 \\ 0.6667 & 0 & 0 \\ \cdots & & \end{bmatrix}_{64x3} \quad (4)$$

3.3.3 Outros comandos para Geração da Malha

Além do comando apresentado para malhas cartesianas, apresentado no Quadro 1, existem outras formas de gerar malhas no MRST. As malhas estruturadas modificadas no MRST têm os formatos apresentados na Figura 21.

Figura 21 – Malhas estruturadas - MRST



Fonte: Lie (2016).

Para se gerar malhas retilíneas, ou seja, formada por retângulos ou paralelepípedos, não necessariamente congruente entre eles, com espaçamento dos vértices variando entre si, pode-se utilizar o comando apresentado no Quadro 2 e exemplificado no Algoritmo 10.

Algoritmo 10 – Malha estruturadas retilíneas

```
1.  % Ex. no domínio [-1,1]x[0,1]
2.  dx = 1-0.5*cos((-1:0.1:1)*pi);
3.  x = -1.15+0.1*cumsum(dx);
4.  y = 0:0.05:1;
5.  G = tensorGrid(x, sqrt(y));
6.  plotGrid(G); axis([-1.05 1.05 -0.05 1.05]);
7.
```

Quadro 2 – Malha cartesiana - tensorGrid

FUNÇÃO SINTAXE:	
`G = tensorGrid(x,y);  para malha cartesiana 2D` `G = tensorGrid(x,y,z);para malha cartesiana 3D`	
	PARÂMETROS DE ENTRADA:
(x,y,z) –	vetores com os vértices ao longo das direções coordenadas.
	RETORNO:
G-	Estrutura da malha, com campos: ***cells; faces; nodes; cartDims; type; griddim**

Fonte: Adaptado do MRST (2018).

Para construir malha curvilíneas, o MRST não possui um comando próprio. Deve-se criar uma malha cartesiana regular e então modificar seus vértices internos. Essas malhas possuem a mesma topologia que as malhas cartesianas, porém, ao invés de retângulos e paralelepípedos, as células são constituídas por quadriláteros e cuboides. O Algoritmo 11 exemplifica essa alteração.

Algoritmo 11 – Malha estruturadas curvilíneas

```
1.  % determinação da malha
2.  nx = 6; ny=12;G = cartGrid([nx, ny]);
3.
4.  %elementos fora do contorno
5.  c = G.nodes.coords;
6.  I = any(c==0,2) | any(c( :,1)==nx,2) | any(c( ,2)==ny,2);
7.
8.
9.  %modificação da coordenada desses elementos
10. G.nodes.coords(~I,:) = c(~I,:) + 0.6*rand(sum(~I),2)-0.3;
11. plotGrid(G);
12.
13.
14. %% Modo 2
15. %causar perturbação é com twister:
16. G = cartGrid([30, 20]);
17. G.nodes.coords = twister(G.nodes.coords);
18. G = computeGeometry(G);
19. plotGrid(G);
20.
```

Por fim, uma opção para criar geometrias mais complexas com malhas estruturadas, é utilizar o método chamado de domínio fictício. No MRST é utilizada a função removeCells, conforme indica o Algoritmo 12.

Algoritmo 12 – Malha com domínio fictício

```
1.
2.   %criação de malhas
3.   x = linspace(-2,2,21); G = tensorGrid(x,x,x);
4.   G = computeGeometry(G);
5.   c = G.cells.centroids;
6.   r = c(:,1).^2 + 0.25*c(:,2).^2+0.25*c(:,3).^2;
7.
8.   %remoção de células
9.   G = removeCells(G, r>1);
10.
11.  plotGrid(G); view(-70,70); axis equal;
12.
13.
```

Outros comandos para definição de geometrias não estruturadas podem ser encontrados em Lie (2016).

3.4 MODELAGEM DAS ROCHAS RESERVATÓRIO

Aquíferos e reservatórios naturais de petróleo consistem em um subvolume de rochas sedimentares que possuem porosidade e permeabilidade para armazenar e transmitir os fluidos, respectivamente (Lie, 2016). Cabe destacar que a porosidade está relacionada com o volume de poros (V_{vazios}/V_{total}), enquanto a permeabilidade está relacionada com a interconexão entre eles.

Nesse tópico, será apresentado como incluir as rochas como parte do modelo de simulação no MRST.

Todos os *solvers* de transporte e escoamento assumem que os parâmetros das rochas são representados por uma estrutura, denominada `rock`. Essa estrutura contém dois campos: `poro` e `perm`. O primeiro (`rock.poro`) expressa, em um vetor, o valor da porosidade para cada célula ativa, associada à malha. O segundo (`rock.perm`) pode conter uma coluna para permeabilidade isotrópica, 2 ou 3 colunas para permeabilidade diagonal (2D e 3D, repectivamente), ou seis colunas para o tensor completo simétrico de permeabilidade, como apresentado na Eq. (5).

$$K(i) = \begin{bmatrix} K_1(i) & K_2(i) & K_3(i) \\ K_2(i) & K_4(i) & K_5(i) \\ K_3(i) & K_5(i) & K_6(i) \end{bmatrix} \xrightarrow{\text{para linha } i} K_1 \quad K_2 \quad K_3 \quad K_4 \quad K_5 \quad K_6 \tag{5}$$

Pode ainda, eventualmente, existir um campo denominado, `ntg`, para representar a razão *net-to-gross*, que consiste de um escalar ou uma coluna com um valor para cada célula ativa.

3.4.1 Exemplos de geração dos parâmetros das rochas

A seguir serão apresentados os algoritmos utilizados no MRST para a geração de diferentes modelos de reservatótios.

3.4.1.1 Modelo Homogêneo

Para geração de modelos homogêneos, utiliza-se o comando `makeRock` conforme apresentado no Algoritmo 13.

Algoritmo 13 – Modelo homogêneo para exemplo base

```
1.  %geração de malha do exemplo base.
2.  G = cartGrid([3 3 3]);
3.
```

Algoritmo 13 – Modelo homogêneo para exemplo base

```
4.   %porosidade uniforme de 0.2 e permeabilidade isotrópica de 200 mD.
5.   %Observe as formas de conversão da unidade.
6.   rock = makeRock(G,200*milli*darcy,0.2);
7.   rock = makeRock(G,convertFrom(200,milli*darcy),0.2);
8.   %modelo homogêneo e permeabilidade anisotrópica.
9.   rock = makeRock(G,[100 100 10].*milli*darcy,0.2);
10.
11.  % plotar Kx
12.  plotCellData(G,convertTo(rock.perm(:,1),milli*darcy));
13.  colorbar('vert'); axis equal tight; view(3);
14.
```

É importante destacar que o MRST utiliza unidades no SI, dessa forma, para outras unidades, utiliza-se as funções com fatores de conversão, presentes no *MRST-core* (especificamente em mrst-2018b\utils\units). No caso apresentado no Algoritmo 13, são utilizadas duas funções de conversão: `milli` e `darcy`, para converter a unidade "Darcy" para "metros^2". Para a conversão, pode-se ainda utilizar as funções `convertTo` e `convertFrom`, para colocar um determinado valor na unidade pretendida ou mudar de uma unidade para o SI, respectivamente.

Para plotar as informações de porosidade, permeabilidade ou qualquer outra de interesse, deve-se utilizar a função `plotCellData`, a qual recebe como argumentos a estrutura da malha e os dados de interesse.

3.4.1.2 Modelos Randômicos e Lognormal

O MRST contém dois métodos simplificados para geração de realizações geoestatísticas (Lie, 2016).

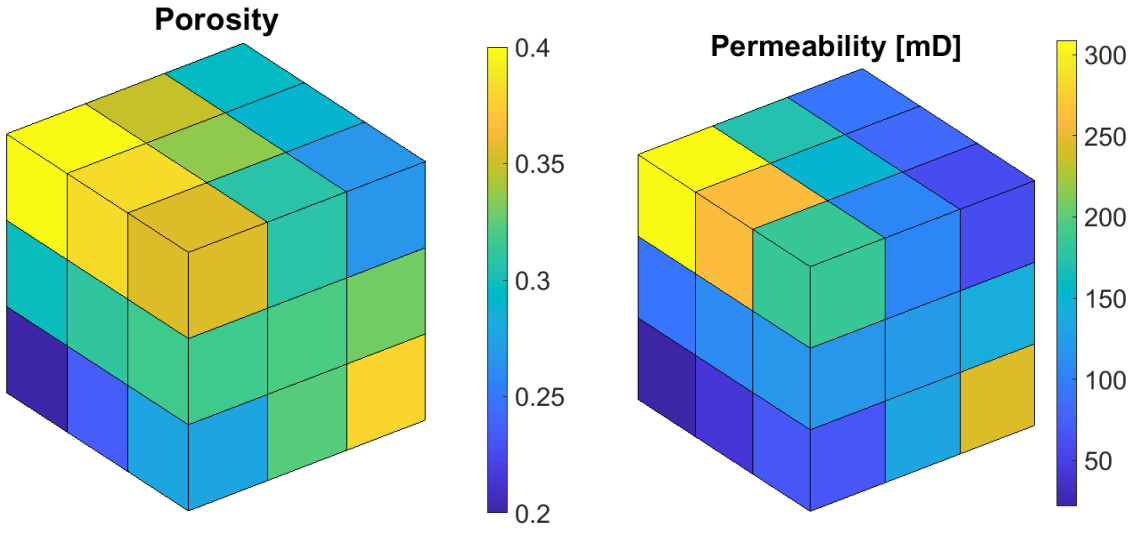
O primeiro método é a geração da porosidade ϕ através da função `gaussianField`, que cria um campo gaussiano randômico. Essa função recebe como parâmetros as dimensões, intervalos para valores do campo randômico, tamanho da convolução kernel e desvio padrão. Para a permeabilidade, utiliza-se a relação Carman–Kozeny, que relaciona esta com a porosidade bem como com a textura da rocha local, ou seja, com a tortuosidade τ (admitida como $\tau = 0.81$, nesse exemplo) e a área específica da superfície A_v (dada por $A_v = 6/d_p$ para o meio formado por grãos esféricos, adotando aqui o diâmetro $d_p = 10\mu m$). Essa relação é expressa na Eq. (6). A utilização do método é apresentada no Algoritmo 14 e ilustrada na Figura 22.

$$k = \frac{1}{8\tau A_v^2}\frac{\phi^3}{\left(1-\phi\right)^2} \text{ onde } A_v = 6/d_p \longrightarrow k = \frac{1}{72\tau}\frac{\phi^3 d_p^2}{\left(1-\phi\right)^2} \quad (6)$$

Algoritmo 14 – Modelo randômico e lognormal – caso 1

```
1.  %geração de malha do exemplo base.
2.  G = cartGrid([3 3 3]);
3.  %porosidade com distribuição randômica lognormal
4.  p = gaussianField(G.cartDims, [0.2 0.4], [11 3 3], 2.5);
5.
6.  % permeabilidade pela relação Carman-Kozeny
7.  K = p.^3.*(1e-5)^2./(0.81*72*(1-p).^2);
8.
9.  %plotar porosidade
10. figure
11. plotCellData(G,rock.poro);
12. colorbar('vert'); axis off equal tight; view(3);
13. title('Porosity')
14. set(gca,'FontSize',24)
15. % plotar permeabilidade
16. figure
17. plotCellData(G,convertTo(rock.perm(:,1),milli*darcy));
18. colorbar('vert'); axis off equal tight; view(3);
19. title('Permeability [mD]')
20. set(gca,'FontSize',24)
```

Figura 22 – Modelos Randômicos e Lognormal



Fonte: A autora (2020).

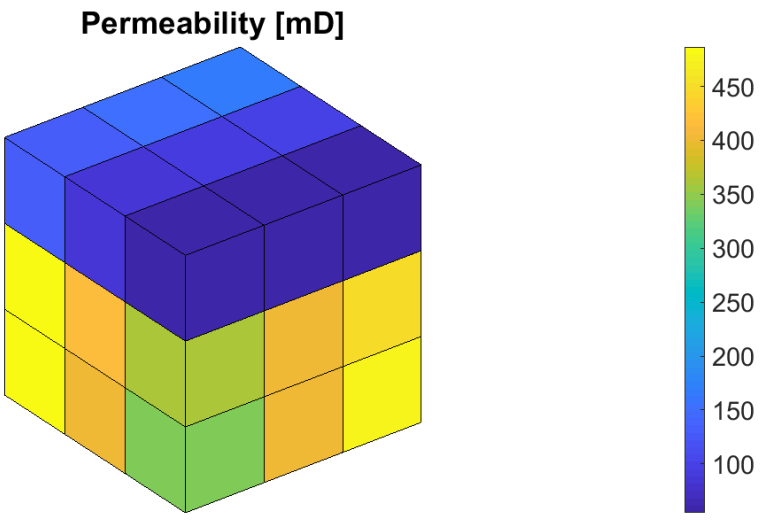
O segundo método é semelhante ao anterior, porém a permeabilidade com distribuição lognormal é dada em cada camada geológica, independente das demais. Para isso é utilizada a

função `logNormLayers`. Essa função recebe como parâmetro a dimensão da malha, as permeabilidades médias em cada camada, e alternativamente podem ser incluídos como parâmetros, os índices, especificando o início e o término de cada camada. Dessa forma, se L(i) é a primeira camada, onde a permeabilidade é dada por Km(i), ela terminará no índice L(i+1) - 1, onde se inicia a segunda camada e assim sucessivamente, na direção do topo a base. Isso é exemplificado na Figura 23 e Algoritmo 15.

Algoritmo 15 – Modelo randômico e lognormal – caso 2

```
1.  %geração de malha do exemplo base.
2.  G = cartGrid([3 3 3]);
3.  %determinação da permeabilidade da em cada camada. A primeira possui
4.  média de 100 mD e as duas últimas de 400 mD
5.  K=logNormLayers(G.cartDims, [100 400]*milli*darcy, 'indices', ...
6.  [1 2 4]);
7.
8.  % plotar permeabilidade
9.  figure
10. plotCellData(G,convertTo(K(:),milli*darcy));
11. colorbar('vert'); axis off equal tight; view(3);
12. title('Permeability [mD]')
13. set(gca,'FontSize',24)
14.
```

Figura 23 – Modelo randômico e lognormal – caso 2



Fonte: A autora (2020).

3.4.1.3 The 10th SPE Comparative Solution Project – SPE10

A Sociedade de Engenheiros de Petróleo (*Society of Petroleum Engineers – SPE)* tem desenvolvido uma série de *benchmarks* para comparação de simuladores e métodos computacionais. O SPE 10 é um dos modelos que se tornou bastante popular na comunidade acadêmica, devido à alta heterogeneidade do seu meio. (LIE, 2016)

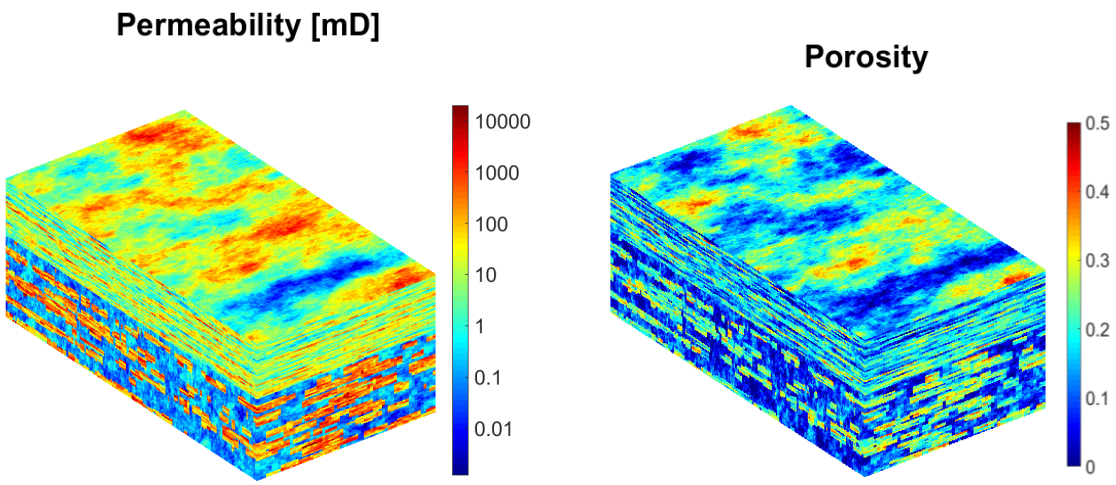
Ele possui uma topologia simples, descrito por uma malha cartesiana regular de 60x220x85 células, com dimensões de 1200x220x170(ft). As 35 camadas superiores possuem 70 ft e apresentam a formação Tarbert consistindo na representação de um ambiente próximo à costa. Enquanto as 50 camadas inferiores possuem 100 ft e é a representação de um ambiente fluvial, Upper Ness.

O SPE 10 está disponível no site https://www.spe.org/web/csp/datasets/set02.htm ou mesmo através do próprio MRST, utilizando o comando apresentado na Figura 12. O Algoritmo 16 apresenta como carregar esse módulo.

Algoritmo 16 – Carregar o SPE 10

```
%carregar o módulo spe10
mrstModule add spe10;

%carregar a malha.
%A geometria em z foi multiplicada por 5 para melhor visualização
G =cartGrid([60 220 85],[60 220 85].*[20 10 2*5].*ft);

%obter os dados das rochas
rock = getSPE10rock(); p=rock.poro; K=rock.perm;

% plotar porosidade plotCellData(G,rock.poro,'EdgeColor','none');
colorbar('vert'); axis off equal; view(3);
shading flat; grid off; title('Porosity');
set(gca,'FontSize',24,'zdir','reverse');
colormap('jet')

% plotar permeabilidade
figure
plotCellData(G,log10(K(:,1)), 'EdgeColor','none');
axis off equal tight off; view(3);
shading flat; grid off; title('Permeability [mD]');
set(gca,'FontSize',24,'zdir','reverse'); colormap('jet');
h=colorbar('vert');
set(h,'XTickLabel',10.^[get(h,'XTick')]);

```

Figura 24 – Propriedades das rochas para o modelo SPE10



Fonte: A autora (2020).

Além do SPE10, existem outros campos de referência, cuja permeabilidade pode ser facilmente obtida e utilizada no MRST.

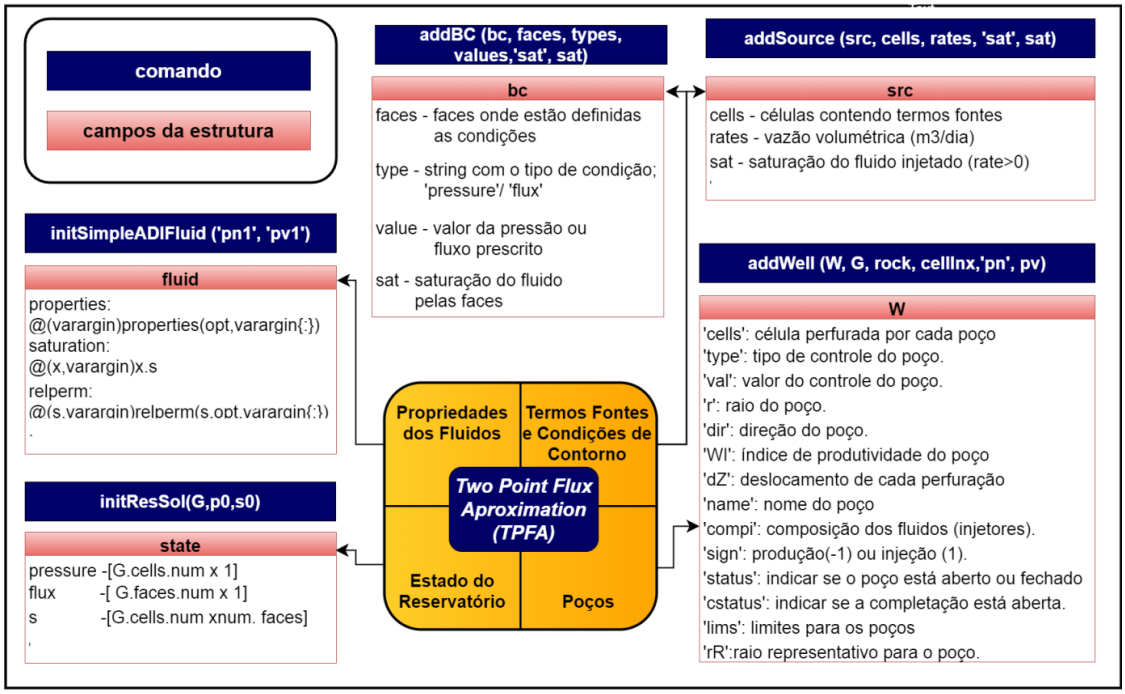
3.5 ESCOAMENTO MONOFÁSICO E INCOMPRESSÍVEL

Nessa seção, será apresentado como representar o comportamento do fluido em termo de seus objetos, constando de suas propriedades básicas como: densidade, viscosidade, compressibilidade. Além disso, de forma geral, serão introduzidos comportamentos mais complexos para o caso multifásico, incluindo propriedades como permeabilidade relativa e pressão capilar que descrevem a interação de múltiplos fluidos e a porosidade da rocha.

Em seguida, serão discutidas as estruturas necessárias para representar as condições de contorno, o termo fonte e modelos de injeção e produção dos poços.

Por fim, será apresentado ao esquema TPFA para a simulação do escoamento monofásico e incompressível, implementada em um dos módulos adicionais do MRST, denominado *incomp*. Um resumo das estruturas a serem discutidas é apresentado na Figura 25.

Figura 25 – Estruturas para a simulação do escoamento incompressível com TPFA



Fonte: A autora (2020).

Antes de introduzir o conteúdo referente aos fluidos, será abordado como tópico inicial duas funções presentes no próprio MATLAB e com ampla utilização no MRST, especialmente empregadas também na estrutura dos fluidos.

3.5.1 Function Handle and Anonymous Functions

Nesse ponto, cabe destacar que *function handle* é um objeto padrão do MATLAB, que permite uma variável guardar uma referência a uma função. De forma muito simplificada, é equivalente ao nome da função (REGISTER, 2007). Com *function handle* se pode passar uma função para outra função.

Para criar uma *function handle* deve preceder o nome da função por @. Observe o exemplo no Algoritmo 17.

Algoritmo 17 – Function Handle

```
1.  %existindo uma função denominada computeSquare.
2.  function y = computeSquare(x)
3.  y = x.^2;
4.  end
5.  % cria-se a handle
6.  f = @computeSquare;
7.  % Realiza chamada da função. Resulta em 16.
8.  a = 4;
9.  b = f(a)
```

Um *function handle* pode ser usada para criar funções anônimas (*Anonymous Functions*). Essas funções são expressas em uma linha, e não necessitam de um programa específico. A sintaxe de uma função anônima é: @(lista_de_argumentos)função_anônima, conforme expressa o Algoritmo 18.

Algoritmo 18 – Função Anônima

```
%declara função anônima
sqr = @(n) n.^2;

% Realiza chamada da função. Resulta em 16.
x = sqr(4)
```

Function handle e *Anonymous Functions* são amplamente utilizadas no MRST como será visto na seção seguinte.

3.5.2 Propriedades dos Fluidos

Para o escoamento monofásico e modelos incompressíveis, as propriedades dos fluidos necessárias são viscosidade e densidade. Para inicializar essa estrutura básica do objeto fluido deve - se utilizar o comando `initSingleFluid`, apresentado no Quadro 3 e exemplificado no Algoritmo 19.

Quadro 3 – Inicialização do objeto fluido – Caso monofásico e Incompressível

FUNÇÃO SINTAXE:	
`fluid = initSingleFluid('pn1', pv1, ...)`	
	PARÂMETROS DE ENTRADA:
'pn'/pv –	Lista de parâmetros e valores definindo características do fluido. Ex: **mu** – Viscosidade do Fluido. Unidade: Pa*s. **rho** – Densidade do fuido.Unidde: kg/m^3
	RETORNO:
fluid –	Estrutura com campos: ***properties:** `@(varargin)properties(opt,varargin{:})` ***saturation:** `@(x,varargin)x.s` ***relperm:** `@(s,varargin)relperm(s,opt,varargin{:})`

Fonte: Adaptado do MRST (2018).

Algoritmo 19 – Inicialização com initSingleFluid

```
1.  %adiciona módulo incompressível.
2.  mrstModule add incomp
3.
4.
5.  % inicialização do objeto fluido.
6.  %Deve-se fornecer como argumento:
7.  %('parametro1', valor1,'parametro2', valor2)
8.  fluid = initSingleFluid('mu', 1*centi*poise,...
9.   'rho', 1014*kilogram/meter^3);
```

Essa função recebe uma lista de parâmetros e seus respectivos valores como argumento, retornando uma estrutura. Essa estrutura gerada contém campos pré-definidos de *function handles*, com as quais se obtém propriedades (ex: densidade e viscosidade), saturação das fases e as curvas de permeabilidade relativa, conforme indicado no Quadro 3.

A função `properties` é a única relevante para o escoamento monofásico, em sua primeira chamada fornece a viscosidade, em seguida, adiciona a densidade e então, outros parâmetros adicionais. A função `saturation` aceita como argumento a estrutura do estado do reservatório e retorna a saturação correspondente. Por fim, a função `relperm` possui a saturação do fluido como argumento e retorna a permeabilidade relativa.

Essa representação é genérica para o módulo adicional `incomp`, com intuito de garantir a compatibilidade com *solvers* escritos para fluidos mais avançados.

No caso do escoamento bifásico, para o caso incompressível, utiliza-se a função `initSimpleFluid`, conforme apresentado no quadro e algoritmos seguintes. Note que, a estrutura retornada possui a mesma forma genérica do caso monofásico.

Quadro 4 – Inicialização do objeto fluido – Caso bifásico e incompressível

FUNÇÃO SINTAXE:	
`fluid = initSimpleFluid('pn1', pv1, ...)`	
	PARÂMETROS DE ENTRADA:
'pn'/pv –	Lista de parâmetros e valores definindo características do fluido. Ex: **mu:** Viscosidade do Fluido. Unidade: Pa*s. **rho:** Densidade do fuido.Unidde: kg/m^3 **n**: Expoentes da curva de permeabilidade relativa
	RETORNO:
fluid –	Estrutura com campos: ***properties:** `@(varargin)properties(opt,varargin{:})` ***saturation:** `@(x,varargin)x.s` ***relperm:** `@(s,varargin)relperm(s,opt,varargin{:})`

Fonte: Adaptado do MRST (2018).

Algoritmo 20 – Inicialização com initSimpleFluid - Caso bifásico

```
1.  %adiciona módulo incompressível.
2.  mrstModule add incomp
3.
4.  %inicialização do objeto fluido  parâmetros e valores.
5.  fluid = initSimpleFluid('mu', [1,10]*centi*poise,...
6.  'rho', [1014, 859]*kilogram/meter^3, ...
7.  'n'  , [2,2]);
8.
```

3.5.3 Estado do Reservatório

Para armazenar as informações do estado do reservatório, o MRST utiliza uma estrutura especial, geralmente denominada por `state`. Para o caso incompressível, essa estrutura é inicializada pelo comando apresentado no Quadro 5.

Quadro 5 – Inicialização do estado do reservatório

FUNÇÃO SINTAXE:	
`state = initResSol(G, p0)` `state = initResSol(G, p0,s0)`	
	PARÂMETROS DE ENTRADA:
G -	Estrutura da malha.
p0 –	Pressão inicial do reservatório. Pode ser um escalar ou um vetor com um valor para cada célula [G.cells.num x 1].
s0-	Saturação inicial do reservatório. Pode ser um vetor [1 x número de fases] ou uma matriz [G.cells.num x número de fases]. O padrão é s0 = 0 para uma única fase
	RETORNO:
state-	Estrutura com campos: ***pressure:** [G.cells.num x 1] ***flux:** [G.faces.num x 1] ***s:** [G.cells.num x número de fases] No caso de 3 fases: (1) Água,(2)Líquido, (3) Vapor.

Fonte: Adaptado do MRST (2018).

Note que a estrutura retornada possui três campos: pressão, com um valor para cada célula; fluxo, com o valor do fluxo de Darcy para cada face da malha; e saturação, com a saturação de cada fase para todas as células. É importante destacar que essa função não inicializa a pressão do fluido para estar em equilíbrio hidrostático, caso isso seja necessário, o próprio usuário deverá fazer.

No caso de poços no modelo, para essa inicialização do reservatório, pode-se usar a função expressa no Quadro 6. Nesse caso, é acrescentada aos parâmetros de entrada a estrutura W, contendo as informações dos poços e ao retorno é acrescentado uma estrutura denominada `wellSol`.

Quadro 6 – Inicialização do estado do reservatório – modelo com poços

FUNÇÃO SINTAXE:	
`state = initState(G, W, p0, s0);`	
	PARÂMETROS DE ENTRADA:
G –	Estrutura da malha.
W –	Estrutura dos poços
p0 –	Pressão inicial do reservatório.
s0-	Saturação inicial do reservatório.
	RETORNO:
state-	Estrutura com campos: ***pressure:** [G.cells.num x 1] ***flux:** [G.faces.num x 1] ***s:** [G.cells.num x número de fases] ***wellSol:** estrutura [1 x 2] com campos: ***wellSol.flux:** fluxo para cada completação de cada poço ***wellSol.pressure:** pressão BHP para cada poço

Fonte: Adaptado do MRST (2018).

3.5.4 Termos fontes/ sumidouros

Para adicionar explicitamente termos fonte/sumidouro, deve-se proceder conforme apresentado no Quadro 7 e exemplificado no Algoritmo 21. Essa é a forma mais simples de adicionar escoamento ao modelo.

Quadro 7 – Criar termos Fonte/Sumidouros

FUNÇÃO SINTAXE:	
`src = addSource(src, cells, values, 'pn1', pv1)`	
	PARÂMETROS DE ENTRADA:
src–	Estrutura de um anterior comando 'addSource' para ser atualizado, ou `src==[]`, para criar um novo.
cells –	células contendo termos fontes.
values –	valor da vazão volumétrica (m3/dia). Um escalar para cada células presente em ‘cells’.
	RETORNO:
src-	Estrutura com campos: ***cells:** células contendo termos fontes ***rates:** valor da vazão volumétrica (m3/dia) ***sat:** composição do fluido injetado (rate>0) (1)Água; (2) Líquido; (3) Vapor.

Fonte: Adaptado do MRST (2018).

Algoritmo 21 – Criar termos Fonte/Sumidouros

```
1.  pv  = sum(poreVolume(G,rock));
2.  src = addSource([], 1, pv);
3.
4.  src = addSource(src, G.cells.num, -pv);
```

Para o modelo estar bem posto é importante que a quantidade de fluido injetado seja igual ao fluido extraído. Dessa forma, a soma dos termos fontes deve ser zero.

3.5.5 Condições de Contorno

Caso não seja especificado, as condições de contorno consideram fluxo nulo na fronteira externa. Para especificar as condições de contorno de Dirichlet ou Neumann, deve-se utilizar o comando apresentado no Quadro 8.

Quadro 8 – Gerar condições de Contorno

FUNÇÃO SINTAXE:	
	`bc = addBc(bc, faces, type, values);` `bc = addBC(bc, faces, type, values, 'sat', sat);`
PARÂMETROS DE ENTRADA:	
bc–	Estrutura de um anterior comando 'addBc' para ser atualizado, ou `bc==[]`, para criar um novo.
faces –	Faces globais do modelo para as quais a condição de contorno será aplicada.
type-	Tipo de condição de contorno: `'pressure'` e `'flux'`
values –	Valor da vazão as condição de contorno. Para `type = 'pressure'`, unidade Pa; para `type ='flux',` unidade m3/s. Um escalar para cada face presente em 'faces'.
RETORNO:	
bc-	Estrutura com campos: ***faces –** faces externas para quais são definidas condições explícitas. ***type** – string com o tipo de condição: 'pressure' e 'flux'. ***value –** pressão [Pa] e/ou valor do fluxo prescrito [m3/s] para cada face em faces. ***sat-** (opcional)composição do fluido pelas faces (1)Água; (2) Líquido; (3) Vapor.

Fonte: Adaptado do MRST (2018).

Por conveniência, o MRST também oferece duas rotinas adicionais para malhas estruturadas com o intuito de estabelecer as condições de Neumann e Dirichlet nas faces

externas, conforme apresentado no Quadro 9, no Quadro 10 e exemplificado no Algoritmo 22. O retorno dessas funções possui a mesma estrutura apresentada no Quadro 8.

Quadro 9 – Condições de Contorno – Malhas Estruturadas – pside

FUNÇÃO SINTAXE:	
`bc = pside(bc, G, side, p);`	
	PARÂMETROS DE ENTRADA:
bc–	Estrutura de um anterior para ser atualizada, ou `bc==[]`, para criar um nova.
G –	Estrutura da Malha.
side-	`side` é string com valores: 1.`'West', 'XMin', 'Left'` 2.`'East', 'XMax', 'Right'` 3.`'South', 'YMin', 'Back'` 4.`'North', 'YMax', 'Front'` 5.`'Upper', 'ZMin', 'Top'` 6.`'Lower', 'ZMax', 'Bottom'`
p –	Valor da pressão em Pa. Valor para a face de cada elemento da lateral.

Fonte: Adaptado do MRST (2018).

Quadro 10 – Condições de Contorno – Malhas Estruturadas – fluxside

FUNÇÃO SINTAXE:	
`bc = fluxside(bc, G, side, flux);`	
	PARÂMETROS DE ENTRADA:
bc–	Estrutura de um anterior para ser atualizada, ou `bc==[]`, para criar um nova.
G –	Estrutura da Malha.
side-	`side` é string com valores: 1.`'West', 'XMin', 'Left'` 2.`'East', 'XMax', 'Right'` 3.`'South', 'YMin', 'Back'` 4.`'North', 'YMax', 'Front'` 5.`'Upper', 'ZMin', 'Top'` 6.`'Lower', 'ZMax', 'Bottom'`
flux –	Fluxo total em m^3/s. Valor total produzido pela soma de todas as faces. positivo = injeção, negativo = extração.

Fonte: Adaptado do MRST (2018).

Algoritmo 22 – Gerar condições de Contorno – Malhas Estruturadas

```
1.  bc = fluxside([], G, 'EAST', 5e3*meter^3/day);
2.  bc = pside(bc, G, 'WEST', 50*barsa);
3.  %Note que ao Leste a soma do fluxo é 5000 m³/dia.
4.  %No oeste cada face tem a pressão de 50 barsa
```

É importante notar a diferença da declaração de `addBC` e `fluxside`. Na primeira, quando se especifica um valor escalar, ele é replicado para todas as faces, enquanto na segunda ele corresponde ao valor cumulativo do fluxo para todas as faces pertencente ao lado global.

3.5.6 Poços

No MRST, o modelo que descreve a vazão dos poços é dado pela diferença entre a pressão média na célula que contém o poço e a pressão dentro do poço, conforme apresenta a Eq. (7).

$$v_p = WI\left(p_i - p_f\right) \tag{7}$$

Onde:

v_p - taxa para cada perfuração
WI - índice do poço
p_i - pressão na célula perfurada
$p_f = p_{wh} + \rho \Delta z_f$ - pressão em cada completação em equilíbrio hidrostático
p_{wh} = pressão no topo
Δz_f = distância vertical até ponto da perfuração

Para representar os poços, utiliza-se a estrutura denominada por W, criada pelo comando apresentado no Quadro 11.

Quadro 11 – Criar poços - addWell

FUNÇÃO SINTAXE:		
`W = addWell(W,G,rock, cellInx,'pn',pv)`		
	PARÂMETROS DE ENTRADA:	
W–	Estrutura do poço. Quando ainda não existe de ser usado W = [].	
G-	Estrutura da malha .	
rock–	Estrutura referente às propriedades do reservatório (permeabilidade, porosidade, *net-to-gross*)	
cellInx –	Vetor com o índice das células perfuradas	
'pn',pv -	Uma ou mais keywords pode ser usada para especificar parâmetros opcionais no poço:	
	*'Type':	Controle do poço -'bhp' ou 'rate'.
	*'Val':	Valor em [Pa] ou [m^3/s].(Padrão – 0)
	*'Radius':	Raio do poço [m]. (Padrão – 0.1)
	*'Dir':	Direção do poço.
	*'InnerProduct':	Para discretizações consistentes, define a matriz de massa. Suporta – 'ip_tpf' (padrão), 'ip_simple', 'ip_quasitpf', 'ip_rt'.

FUNÇÃO SINTAXE:	
`W = addWell(W,G,rock, cellInx,'pn',pv)`	
*'WI':	Índice de produtividade do poço. Vetor com comprimento igual ao número de perfurações no poço. (Padrão – WI = repmat(-1, [numel(cellInx),1])
*'Kh':	Permeabilidade x espessura. (Padrão – WI = repmat(-1, [numel(cellInx),1]).
*'Skin':	Fator de película calculando a efetividade do poço. Valor escalar de comprimento = numel(cellInx). Padrão = 0.
*'Comp_i':	Composição do fluido para poços injetores. (Padrão – Comp_i = [1,0,0], %injeção de água).
*'Sign':	Define se o poço é produtor (-1) ou injetor (1).
*'Name':	*String* com o nome do poço.
*'refDeph':	Profundidade de referência do modelo de controle, onde é definido o BHP.
RETORNO:	
W -	Estrutura os seguintes campos:
	***cell–** Célula perfurada por cada poço
	***type–** Tipo de controle do poço.
	***val–** Valor do controle do poço.
	***r–** Raio do poço.
	***dir–** Direção do poço.
	***WI–** índice de produtividade do poço.
	***dZ–** Deslocamento de cada perfuração medida a partir do mais alto contato horizontal
	***name–** Nome do poço.
	***compi–** Composição dos fluidos (injetores).
	***sign–** Produção(-1) ou injeção (1).
	***status–** Variável booleana, indicando se o poço está aberto ou fechado.
	***cstatus–** Uma entrada por perfuração, indicando se a completação está aberta.
	***lims–** Limites para os poços. Em geral, injetores tem limite superior e produtores possuem limite inferior. Contém campos por tipos (bhp, rate, orat,..)
	***rR–** Raio representativo para o poço.

Fonte: Adaptado do MRST (2018).

A seguir se apresenta um exemplo de como utilizar essa função em um campo de malha 3x3x3, estabelecendo dois poços, um injetor e outro produtor. O injetor é controlado por vazão

de injeção 1.0 m^3/dia e está localizado no canto inferior esquerdo da malha, com completações nas duas últimas camadas. O produtor é controlado por BHP no valor de 10^5 Pa e localizado no canto superior direito, com completações nas duas primeiras camadas. O Algoritmo 23 apresenta o código a ser utilizado.

Algoritmo 23 – Uso do comando addWell

```
%Define malha
nx = 3; ny = 3; nz = 3;
G = cartGrid([nx, ny, nz]);
G = computeGeometry(G);
rock = makeRock(G, 200*milli*darcy, 0.2);

%Define Poço 1
cellsWell1 =  (1+nx*ny)  - nx*ny  - nx*ny*nz;
W = addWell([], G, rock, cellsWell1, 'Type', 'rate', ...
'InnerProduct', 'ip_tpf', ...
'Val', 1.0/day(), 'Radius', 0.1, 'name', 'I');'

%Define Poço 2
cellsWell2 = ny*nx : nx*ny : nx*ny*2;
W = addWell(W, G, rock, cellsWell2, 'Type', 'bhp', ...
'InnerProduct', 'ip_tpf', ...
'Val', 1.0e5, 'Radius', 0.1, 'name', 'P');
```

O MRST também possui uma função para especificar poços verticais em malhas cartesianas ou extrudadas, conforme apresentado no Quadro 12.

Quadro 12 – Criar poços verticais – malhas cartesianas

FUNÇÃO SINTAXE:	
`W = verticalWell(W, G, rock, I, J, K, 'pn',pv)`	
	PARÂMETROS DE ENTRADA:
W–	Estrutura do poço. Quando ainda não existe de ser usado W = [].
G–	Estrutura da malha .
rock–	Estrutura referente às propriedades do reservatório
I,J –	Localização horizontal do poço. Há dois modos: Modo 1: I é o índice ao longo da primeira direção lógica; J é o índice ao longo da segunda direção lógica. Modo 2: I é o índice da célula (1<= I <= `G.cells.num`) no topo da coluna em que o poço vertical será completado;

FUNÇÃO SINTAXE:	
`W = verticalWell(W, G, rock, I, J, K, 'pn',pv)`	
	J não existe. Essa declaração só é possível apenas quando existe camadas.
K –	Vetor de camadas nas quais o poço será completado. Padrão: `1 : num_layers`, onde `num_layers` representa o número de camadas.
`'pn',pv` –	Uma ou mais *keywords* pode ser usada para especificar parâmetros opcionais no poço, como indicado no Quadro 11.
	RETORNO:
W –	Estrutura W conforme indicado no Quadro 11.

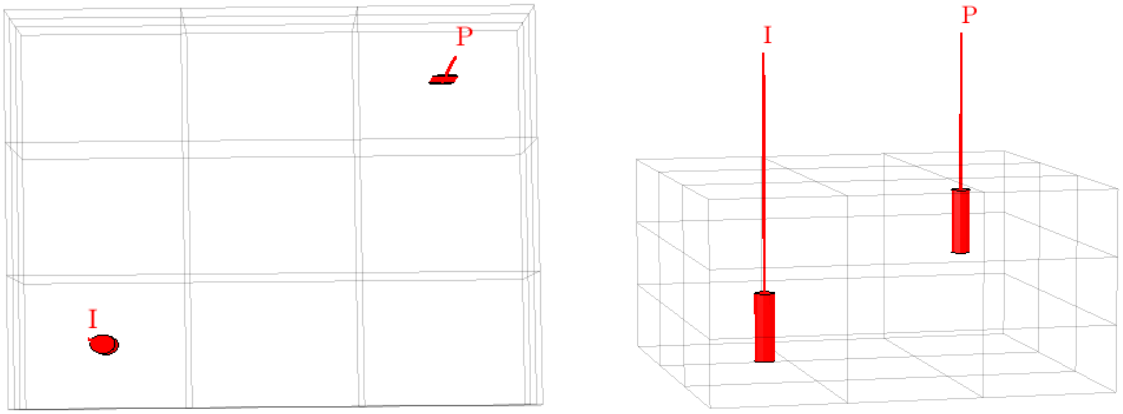
Fonte: Adaptado do MRST (2018).

Implementando o mesmo exemplo anterior, utilizando, porém o comando `verticalWell`, obtém-se o código expresso no Algoritmo 24. Ambas as implementações resultam nos poços representados na Figura 26.

Algoritmo 24 – Uso do comando verticalWell

```
1.  %Define malha
2.  nx = 3; ny = 3; nz = 3;
3.  G = cartGrid([nx, ny, nz]);
4.  G = computeGeometry(G);
5.  rock = makeRock(G, 200*milli*darcy, 0.2);
6.
7.  %Define Poço 1
8.  W = verticalWell([], G, rock, 1, 1, 2:3, ...
9.  'Type', 'rate' , 'Val', 1.0/day(),...
10. 'Radius', 0.1, 'InnerProduct', 'ip_tpf', ...
11. 'Comp_i', [1 0], 'name', 'I',...
12.             'Sign', 1,'refDepth', 0);
13. %Define Poço 2
14. W = verticalWell(W, G, rock, 3, 3, 1:2, ...
15. 'Type', 'bhp' , 'Val', 1.0e5, ...
16. 'Radius', 0.1, 'InnerProduct', 'ip_tpf', ...
17. 'Comp_i', [0 1], 'name', 'P',...
18.             'Sign', -1,'refDepth', 0);
19.
20.
```

Figura 26 – Definição de poços



Fonte: A autora (2020).

3.5.7 Two Point Flux Approximation (TPFA)

Inicialmente, será apresentada, a obtenção da equação para o esquema *Two Point Flux Approximation (TPFA)* através da discretização da malha, introduzindo os conceitos como meia face e meia-transmissibilidade. Em seguida, será apresentada a implementação do método no MRST.

3.5.7.1 Discretização de Volumes Finitos Básica - TPFA

Sem perda da generalidade, será considerado o caso simplificado da equação monofásica de escoamento (Eq. (8)).

$$\nabla \cdot \vec{v} = q, \quad \vec{v} = -K \nabla P, \quad in\ \Omega \subset \Re^d \tag{8}$$

Para discretizar essa equação por volumes finitos, deve-se reescrevê-la na forma integral usando uma célula Ω_i da malha como volume de controle (ver Figura 27). Dessa forma, se obtém a Eq. (9), apresentada a seguir:

$$\int_{\partial\Omega_i} \vec{v} \cdot \vec{n}\, ds = \int_{\Omega_i} q\, dx, \tag{9}$$

Aproximando a integral sobre a face da célula, pelo teorema do ponto médio, pode utilizar a lei de Darcy, para escrever a o fluxo pela Eq. (10).

$$\upsilon_{i,k} = \int_{\partial\Gamma_{i,k}} \vec{\upsilon}\cdot\vec{n}\,ds \approx \mathrm{A}_{i,k}\,\vec{\upsilon}\left(\vec{x}_{i,k}\right)\cdot\vec{n}_{i,k} \qquad \Gamma_{i,k} = \partial\Omega_i \cap \partial\Omega_k$$
$$= -\mathrm{A}_{i,k}\left(K\nabla p\right)\left(\vec{x}_{i,k}\right)\cdot\vec{n}_{i,k} \tag{10}$$

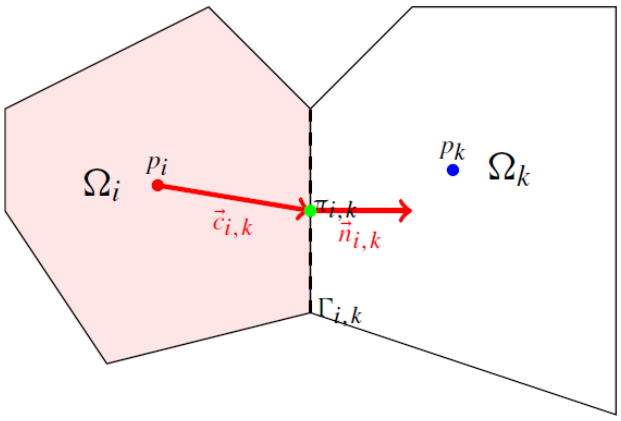
Onde $\Gamma_{i,k}$ consiste na fronteira entre as células Ω_i e Ω_k, e $\vec{x}_{i,k}$ é o centroide de $\Gamma_{i,k}$. De acordo com Lie (2016), $\Gamma_{i,k}$ é denominado como meia-face, uma vez que está associada a uma célula particular Ω_i e a certo vetor $\vec{n}_{i,k}$. Cabe destacar, que nessa fronteira, existe ainda outra meia-face, denominada por $\Gamma_{k,i}$, com área idêntica a $A_{k,i} = A_{i,k}$, mas com vetor normal oposto $\vec{n}_{k,i} = -\vec{n}_{i,k}$.

O gradiente de pressão constante na Eq. (10) deve ser descrito, por diferença finita unilateral, como a diferença entre a pressão da face $\pi_{i,k}$ e a pressão em algum ponto da célula. Como se conhece apenas o valor médio da pressão dentro da célula deve-se assumir que a pressão é linear (ou constante) dentro de cada célula, de forma que, o valor da pressão no centro da célula é idêntico a sua pressão média p_i. Dessa forma, chega-se a:

$$\upsilon_{i,k} \approx \mathrm{A}_{i,k}\,\mathrm{K}_i\,\frac{\left(p_i - \pi_{i,k}\right)\vec{c}_{i,k}}{\left|\vec{c}_{i,k}\right|^2}\cdot\vec{n}_{i,k} = \mathrm{T}_{i,k}\left(p_i - \pi_{i,k}\right) \tag{11}$$

Onde $T_{i,k}$ é a transmissibilidade unilateral, ou meia transmissibilidade, que está associada a uma única célula e fornece a relação de dois pontos entre o fluxo através da face da célula e a diferença de pressão entre o centroide da célula e da face.

Figura 27 – Duas células para definir Two – Point Finite – Volume



Fonte: Lie (2016).

Para obter a equação final, acrescenta-se a equação da continuidade do fluxo através das faces $v_{i,k} = -v_{k,i} = v_{i,k}$ e a continuidade da pressão das faces $\pi_{i,k} = -\pi_{k,i} = \pi_{i,k}$, resultando nas seguintes equações:

$$v_{i,k} = T_{i,k}\left(p_i - \pi_{i,k}\right) \longrightarrow T_{i,k}^{-1} v_{i,k} = p_i - \pi_{i,k} \tag{12}$$

$$v_{k,i} = T_{k,i}\left(p_k - \pi_{k,i}\right) \longrightarrow -v_{i,k} = T_{k,i}\left(p_k - \pi_{i,k}\right) \longrightarrow -T_{k,i}^{-1} v_{i,k} = p_k - \pi_{i,k} \tag{13}$$

Pela Eq. (12) e Eq. (13), e eliminando $\pi_{i,k}$:

$$\begin{aligned} p_i - T_{i,k}^{-1} v_{i,k} = p_k + T_{k,i}^{-1} v_{i,k} &\rightarrow p_i - p_k = T_{i,k}^{-1} v_{i,k} + T_{k,i}^{-1} v_{i,k} \\ &\rightarrow \left[T_{i,k}^{-1} + T_{k,i}^{-1}\right]^{-1}\left(p_i - p_k\right) = v_{i,k} \end{aligned} \tag{14}$$

$$\left[T_{i,k}^{-1} + T_{k,i}^{-1}\right]^{-1}\left(p_i - p_k\right) = v_{i,k} \rightarrow T_{ik}\left(p_i - p_k\right) = v_{i,k} \tag{15}$$

Onde T_{ik} é a transmissibilidade associada com a conexão de duas células.

Por fim, o esquema TPFA para a Eq. (8) é escrito pela Eq. (16), utilizando dois pontos, as pressões médias das células p_i e p_k para aproximar o fluxo através da interface Γ_{ik} entre as células Ω_i e Ω_k.

$$\sum_k T_{ik}\left(p_i - p_k\right) = q_i \quad \forall \Omega_i \subset \Omega \tag{16}$$

No MRST é criada uma matriz esparsa A=$\left[a_{ij}\right]$, definida conforme a Eq. (17). O sistema é simétrico, e a solução é somente definida para uma constante arbitrária. No MRST, $p_1 = 0$, adicionando uma constante para a primeira diagonal da matriz A.

$$a_{ij} = \begin{cases} \sum_k T_{ik} & se\ j = i \\ -T_{ik} & se\ j \neq i \end{cases} \tag{17}$$

Essa matriz é esparsa e irá apresentar uma estrutura em faixas para malhas estruturadas: tridiagonal para malhas e penta ou heptadiagonal para malhas cartesianas em 2D e 3D, respectivamente.

3.5.7.2 Implementação no MRST

O esquema Two Point Flux Approximation (TPFA) é implementado por duas diferentes rotinas: computeTrans e incompTPFA. A primeira, computeTrans, é responsável pelo cálculo da meia transmissibilidade, apresentada acima e não depende do modelo do fluido, estado do reservatório ou mecanismos de condução. Esses parâmetros, juntamente com a malha e meia transmissibilidade, serão utilizados na segunda rotina, incompTPFA, que solucionará o esquema TPFA. O comando computeTrans depende apenas da estrutura na malha e das propriedades da rocha reservatório, que deve esta nas unidades do sistema internacional – SI. Essas rotinas são apresentadas no Quadro 13 e Quadro 14. Cabe destacar que essas funções são geralmente aplicadas para malhas ortogonais, caso contrário, recomenda-se, por exemplo, o uso da função incompMPFA.

Quadro 13 – Comando computeTrans

FUNÇÃO SINTAXE:	
`hT = computeTrans(G, rock)`	
	PARÂMETROS DE ENTRADA:
G–	Estrutura da malha
rock –	Estrutura das propriedades da rocha reservatório.
	RETORNO:
hT–	Meia transmissibilidade para cada face local de cada elemento da malha. Vetor de dimensão `[G.cells.faces x 1]`

Fonte: Adaptado do MRST (2018).

Quadro 14 – Comando incompTPFA

FUNÇÃO SINTAXE:	
`state = incompTPFA(state, G, hT, fluid, ...'mech', obj, ...)`	
	PARÂMETROS DE ENTRADA:
state –	Estrutura do estado inicial do reservatório.
G–	Estrutura da malha.
hT –	Vetor da meia transmissibilidade
fluid –	Objeto fluido.
`'mech'/obj–`	`'mech' = 'bc', 'scr', e ou 'wells'`
	RETORNO:
State –	Estrutura com campos:
	***pressure –** pressão para cada elemento da malha; ***facePressure –** pressão para cada face da malha; ***flux–** fluxo através das faces; ***A–** matriz do sistema, se especificado 'MatrixOutput'; ***wellSol–** estrutura [1 x 2] com campos:

FUNÇÃO SINTAXE:
`state = incompTPFA(state, G, hT, fluid, ...'mech', obj, ...)`
***wellSol.flux**– fluxo para cada completação de cada poço* ***wellSol.pressure** – pressão BHP para cada poço*

Fonte: Adaptado do MRST (2018).

A seguir serão apresentados alguns detalhes de como funciona o cálculo da transmissibilidade, bem como, a montagem e solução do sistema usando TPFA. Para isso, serão apresentados alguns trechos dos códigos das rotinas *simpleComputeTrans* e *simpleIncompTPFA*, presentes no módulo adicional do MRST denominado *book*.

A transmissibilidade, conforme indicado na seção anterior é dada por:

$$\mathrm{T}_{i,k} = \mathrm{A}_{i,k}\,\mathrm{K}_i\,\frac{\overset{\mathrm{r}}{c}_{i,k}}{\left|\overset{\mathrm{r}}{c}_{i,k}\right|^2}\cdot\overset{\mathrm{r}}{n}_{i,k} \tag{18}$$

$$T_{ik} = \left[T_{i,k}^{-1} + T_{k,i}^{-1}\right]^{-1} \tag{19}$$

Desse modo, inicialmente, deve-se determinar o vetor $\overset{\mathrm{r}}{c}_{i,k}$ com extremidades no centro da célula e da face, conforme representado na Figura 27, resultando no item 1 do Algoritmo 25.

Em seguida, deve-se obter a normal das faces. No MRST, é admitido que elas tem comprimento correspondente a área da face. A direção do vetor é dada pela estrutura `G.faces.neighbors,` se a célula está na primeira coluna desse vetor, recebe o sinal positivo, caso contrário, recebe o sinal negativo. O item 2 do Algoritmo 25 apresenta a determinação de $\mathrm{A}_{i,k}\cdot\overset{\mathrm{r}}{n}_{i,k}$. Por fim, obtém a permeabilidade no formato completo (item 3 –Algoritmo 25) e obtém a meia transmissibilidade expressa na Eq. (18).

Algoritmo 25 – Cálculo da meia transmissibilidade

```
1.  %% Item 1
2.  %vetor com as faces da malha
3.  hf = G.cells.faces(:,1);
4.  %células correspondente a cada face
5.  hf2cn = gridCellNo(G);
6.  % Vetor c_{i,k}
7.  C = G.faces.centroids(hf,:)-G.cells.centroids(hf2cn,:);
8.
9.
10. %% Item 2
11. %determinação do sinal
12. sgn = 2*(hf2cn == G.faces.neighbors(hf, 1)) - 1;
```

```
13.  %Vetor A_{i,k}.n_{i,k}
14.  N = bsxfun(@times, sgn, G.faces.normals(hf,:));(1)
15.
16.
17.  %% Item 3
18.  %expande a permeabilidade no formato completo [nc x dim^2]
19.  [K, i, j] = permTensor(rock, G.griddim);
20.
21.  %% Item 4
22.  %cálculo da meio trasmissibilidade
23.  hT = zeros(size(hf2cn));
24.  for k=1:size(i,2),
25.  hT = hT + C(:,i(k)) .* K(hf2cn, k) .* N(:,j(k));
26.  end
27.  hT = hT./ sum(C.*C,2);
28.
```

(1)Função bsxfun explicada na seção 3.7.

Após a obtenção da meia transmissibilidade, essa informação é passada para a rotina *simpleIncompTPFA*. O primeiro passo dessa função é ajustar a meia transmissibilidade à viscosidade do fluido, uma vez que ela foi obtida para viscosidade unitária (item 1 - Algoritmo 26). Em seguida, é realizado o cálculo da transmissibilidade da face, como a média harmônica da meia-transmissibilidades (item 2 - Algoritmo 26). O próximo passo consiste em somar as transmissibilidades das faces de cada célula, para realizar a montagem da diagonal da matriz A, conforme apresentado na Eq. (17), (item 3 - Algoritmo 26). De posse, dos elementos da matriz, ela é criada através da função *sparse* do MATLAB (item 3 - Algoritmo 26). Finalmente, verifica-se se a condição de contorno de Dirichlet está imposta, caso contrário, modifica-se arbitrariamente o primeiro elemento da matriz do sistema para corrigir a pressão na primeira célula para zero, antes de resolver o sistema (item 5 - Algoritmo 26) e então, é possível resolver o sistema (item 6 - Algoritmo 26). Obtida as pressões nas células, podem ser calculados os valores das pressões nos centroides das faces e reconstruído o fluxo através da célula (item 7 - Algoritmo 26).

Algoritmo 26 – Esquema TPFA

```
1.   %% Item 1
2.   %ajuste da meia transmissibilidade aos parâmetros do fluido
3.   mob = 1./fluid.properties(state);
4.   hT = hT .* mob;
5.
```

Algoritmo 26 – Esquema TPFA

```
6.  %% Item 2
7.  % Cálculo da transmissibilidade.
8.  %hf = G.cell.faces(:,1)
9.  %fornece o num. global da face para cada meia-face.
10. %Logo, o vetor 1/hT de cada meia-face será acumulado.
11. T = 1./ accumarray(hf, 1./ hT,[G.faces.num,1]);(1)
12.
13.
14. %% Item 3
15. nc = G.cells.num;
16. %exclui faces externas
17. i = all(G.faces.neighbors ~= 0, 2);
18. % obtém G.faces.neighbors para as faces internas
19. n1 = G.faces.neighbors(i,1);
20. n2 = G.faces.neighbors(i,2);
21. %concatena os dois vetores de vizinhos
22. %replica a transmissibilidade.
23. %acumula a transmissibilidade da face para cada célula.
24. d = accumarray([n1; n2], repmat(T(i),[2,1]),[nc, 1]);
25.
26. %% Item 4
27. % Índice dos elementos em I
28. I = [n1; n2; (1:nc)'];
29. % Índice dos elementos em J
30. J = [n2; n1; (1:nc)'];
31. % Elementos da matriz
32. V = [-T(i); -T(i); d]; clear d;
33. % cria matriz esparsa, fornecendo como argumento
34. %os índices e o valor a ser preenchido (I, J, V).
35. A = sparse(double(I), double(J), V, nc, nc);
36.
37. %% Item 5
38. A(1) = 2*A(1)
39.
40. %% Item 6
41. %Resolver o sistema: mldivide - avalia o sistema e escolhe
42. %o método mais adequado para ele
43. p = mldivide(A, rhs);
44.
45. %% Item 7
46. fp   =   accumarray(G.cells.faces(:,1),   p(hf2cn).*hT,...
47. [G.faces.num,1])./accumarray(G.cells.faces(:,1),  hT,  ...
48. [G.faces.num,1]);
49.
50. nf = G.faces.num;
51. ni = G.faces.neighbors(i,:);
52.
53. flux = -accumarray(find(i), ...
54. T(i).*(p(ni(:,2))-p(ni(:,1))),[nf, 1]);
55.
```

(1)Função accumarray explicada na seção 3.7.

3.5.8 Exemplo

Após a compreensão de toda estrutura requerida para a simulação do escoamento monofásico e incompressível utilizando o MRST, é apresentado, no Algoritmo 27, um exemplo básico para consolidação desse conjunto de informações.

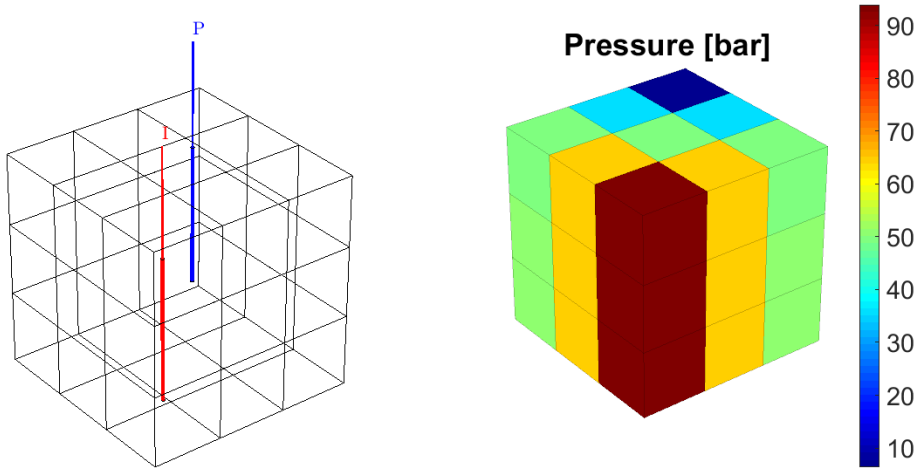
Algoritmo 27 –Exemplo incompTPFA

```
%% Item 1
clear all
close all
clc
%adiciona módulos.
mrstModule add incomp;
%% Item 2
%cria a malha.
G = cartGrid([3 3 3]);
G = computeGeometry(G);
%% Item 3
% determina as propriedades da rocha reservatório.
rock= makeRock(G, 100*milli*darcy, 0.2);
%% Item 4
%estabelece as propriedades do fluido.
fluid = initSingleFluid('mu', 1*centi*poise,...
 'rho', 1000*kilogram/meter^3);

%% Item 5
%determina o estado inicial do reservatório
state = initResSol(G, 100*barsa, 1.0);

%% Item 6
% determina condições de contorno, termos fontes/sumidouros
e/ou poços
 W = addWell([],G, rock, 1: 9: 27,'Type','bhp', ...
 'Val', 100*barsa, 'name', 'I', ...
    'radius',0.1, 'Comp_i', [1 0],'Sign',1);

W = addWell(W,G, rock, 9: 9: 27,'Type','bhp', ...
 'Val', 0, 'name', 'P', 'radius',0.1, ...
    'Comp_i', [0 1],'Sign',-1);


%% Item 7
%monta e resolve o sistema
hT = computeTrans(G,rock);
gravity reset on;
state = incompTPFA(state, G, hT, fluid, 'wells', W);

```

Algoritmo 27 –Exemplo incompTPFA

```
41.  %% Item 8
42.  %Plotar os resultados
43.  subplot(1,2,1), pos = get(gca,'Position'); clf
44.  plotGrid(G, 'FaceColor', 'none');
45.  view(3), camproj perspective, axis equal off,
46.  plotWell(G, W(1), 'radius',  .1, 'height',...
47.   1, 'color', 'r');
48.  plotWell(G, W(2), 'radius', .1, 'height', ...
49.   1, 'color', 'b');
50.  hold on
51.
52.  set(gca, 'Position', pos);
53.  subplot(1,2,2)
54.  plotCellData(G, convertTo(state.pressure...
55.   (1:G.cells.num), barsa),'EdgeAlpha',.1);
56.  title('Pressure [bar]')
57.  view(3), camproj perspective, axis equal off
58.  colorbar
59.  colormap('jet')
60.  set(gca, 'FontSize', 24);
61.
```

Figura 28 – Resultado exemplo incompTPFA



Fonte: A autora (2020).

3.6 OPERADORES DISCRETOS

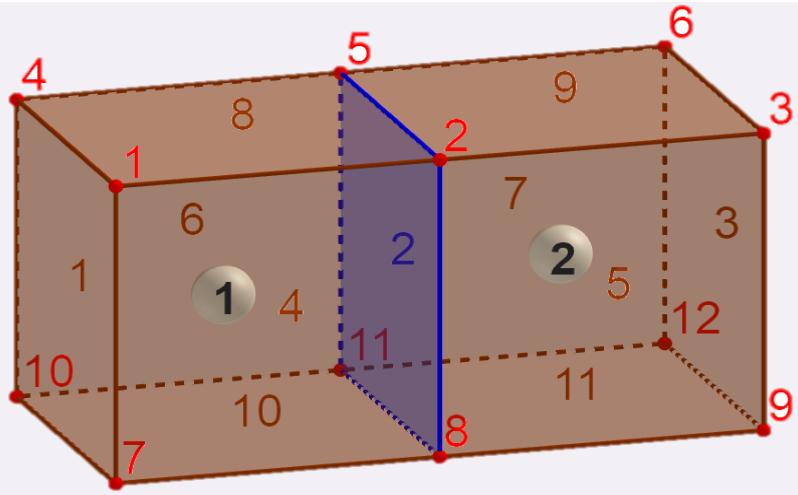
Na seção 3.5.7 referente ao TPFA foi utilizada a notação $v_{i,k}$ e v_{ik} para descrever o fluxo entre uma única interface de duas células vizinhas. No entanto, o MRST dispõe de uma forma mais abstrata de escrever o esquema TPFA que pode ser aplicado em modelos mais complexos.

Conforme Lie (2016), a ideia é introduzir operadores discretos de divergência e gradiente similares ao apresentado na equação contínua.

Para a definição dos operadores, é necessário incorporar informações sobre a topologia da malha. No entanto, uma vez que tenham sido criados, podem ser aplicados a quaisquer células e faces, sem maiores detalhes da topologia.

A definição da topologia da malha é feita por meio de dois mapeamentos. O primeiro mapeia a célula para o número de faces que definem essa célula, e o segundo consiste no mapeamento das células vizinhas correspondentes a uma dada face. Considerando n_c o número de células e n_f o número de faces, matematicamente, esses mapeamentos são descritos, respectivamente, por: $F : \{1,\mathrm{K},n_c\} \to \{0,1\}^{n_f}$, representado no MRST por `G.cell.faces`, conforme indicado na seção 3.3.2; e $C_1,C_2 : \{1,\mathrm{K},n_f\} \to \{1,\mathrm{K},n_c\}$, representado no MRST por `G.faces.neighbors`. A Figura 29 ilustra como são construídos os referidos mapeamentos.

Figura 29 – Mapeamentos da topologia da malha para definir operadores discretos



$$F:\{1,\mathrm{K},n_c\}\to\{0,1\}^{n_f}$$

`G.cell.faces`

c	F(c)
1	1
1	2
1	4
1	6
1	8
1	10
2	2
2	3
2	5
2	7
2	9
2	11

$$C_1,C_2:\{1,\mathrm{K},n_f\}\to\{1,\mathrm{K},n_c\}$$

`G.faces.neighbors`

f	C_1(f)	C_2(f)
1	0	1
2	1	2
3	2	0
4	0	1
5	0	2
6	1	0
7	2	0
8	0	1
9	0	2
10	1	0
11	2	0

Fonte: A autora (2020).

Compreendido os mapeamentos, é possível construir o operador divergente e gradiente. O operador divergente é o mapeamento das faces para as células. Se $\mathrm{v}[f]$ descreve o fluxo discreto através da face f com orientação da célula $C_1(f)$ para a $C_2(f)$, então o operador divergente desse fluxo restrito para a célula c é dado pela Eq. (20) (BAO et al, 2017).

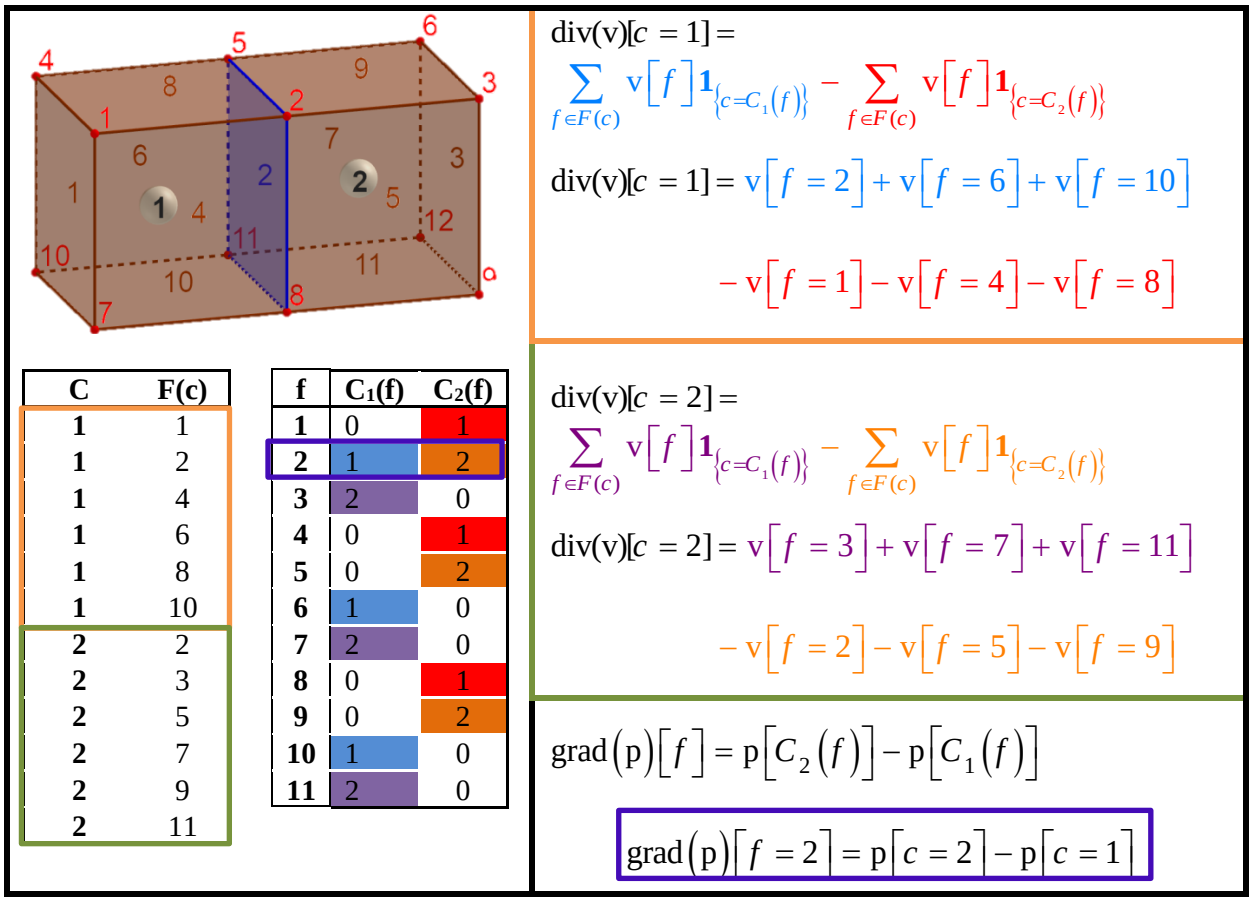
$$\mathrm{div(v)}[c]=\sum_{f\in F(c)}\mathrm{sgn}(f)\mathrm{v}[f],\quad \mathrm{sgn}(f)=\begin{cases}1,\ \mathrm{se}\ c=C_1(f),\\ -1,\ \mathrm{se}\ c=C_2(f).\end{cases} \tag{20}$$

O operador gradiente mapeia dos pares de células para faces. Se, por exemplo, p denota a matriz de pressões das células, o gradiente dessa pressão em relação à face f é definido da seguinte como (BAO et al, 2017):

$$\text{grad}(\text{p})[f] = \text{p}[C_2(f)] - \text{p}[C_1(f)] \tag{21}$$

Observe que esse gradiente discreto não se divide pela distância entre os pontos. A Figura 30 exemplifica como funciona essas definições dos operadores na malha gerada pelo MRST. A notação $\mathbf{1}_{\{\text{exp}\,r\}}$ indica 1 se $\{\text{exp}\,r\}$ for verdadeiro e 0, caso contrário.

Figura 30 – Operadores divergentes e gradientes



C	F(c)
1	1
1	2
1	4
1	6
1	8
1	10
2	2
2	3
2	5
2	7
2	9
2	11

f	C1(f)	C2(f)
1	0	1
2	1	2
3	2	0
4	0	1
5	0	2
6	1	0
7	2	0
8	0	1
9	0	2
10	1	0
11	2	0

Fonte: A autora (2020).

Conforme LIE (2016), no caso contínuo, o operador gradiente é adjunto do operador divergente a menos de um sinal para as condições de fluxo nulo na face externa (Eq. (22)).

$$\int_{\Omega} p\nabla\cdot\vec{v}d\vec{x} + \int_{\Omega} \vec{v}\cdot p\nabla d\vec{x} = 0 \tag{22}$$

No caso dos operadores discretos essa propriedade se mantém. Considerando $S_C = \{1,\text{K},n_c\}$ e $S_f = \{1,\text{K},n_f\}$, pode-se escrever:

$$\sum_{c\in S_C}\text{div}(v)[c]\mathbf{p}[c]=\sum_{c\in S_C}\mathbf{p}[c]\left(\sum_{f\in F(c)}v\left[f\right]\mathbf{1}_{\{c=C_1(f)\}}-\sum_{f\in F(c)}v\left[f\right]\mathbf{1}_{\{c=C_2(f)\}}\right)$$
$$=\sum_{c\in S_C}\sum_{f\in F(c)}v\left[f\right]\mathbf{p}[c]\mathbf{1}_{\{c=C_1(f)\}}\mathbf{1}_{\{f\in F(c)\}}-\sum_{c\in S_C}\sum_{f\in F(c)}v\left[f\right]\mathbf{p}[c]\mathbf{1}_{\{c=C_2(f)\}}\mathbf{1}_{\{f\in F(c)\}} \quad (23)$$
$$=\sum_{f\in F(c)}\sum_{c\in S_C}v\left[f\right]\mathbf{p}[c]\mathbf{1}_{\{c=C_1(f)\}}\mathbf{1}_{\{f\in F(c)\}}-\sum_{f\in F(c)}\sum_{c\in S_C}v\left[f\right]\mathbf{p}[c]\mathbf{1}_{\{c=C_2(f)\}}\mathbf{1}_{\{f\in F(c)\}}$$

Temos que $\mathbf{1}_{\{c=C_1(f)\}}\mathbf{1}_{\{f\in F(c)\}}\neq 0 se c=C_1(f)$ e $\mathbf{1}_{\{c=C_2(f)\}}\mathbf{1}_{\{f\in F(c)\}}\neq 0 se c=C_2(f)$, logo, pode-se escrever a Eq. (23) como:

$$\sum_{c\in S_C}\text{div}(v)[c]\mathbf{p}[c]=\sum_{f\in S_f}v\left[f\right]\mathbf{p}\left[C_1(f)\right]-\sum_{f\in S_f}v\left[f\right]\mathbf{p}\left[C_2(f)\right]$$
$$=-\sum_{f\in S_f}\left(\mathbf{p}\left[C_2(f)\right]-\mathbf{p}\left[C_1(f)\right]\right)v\left[f\right] \quad (24)$$
$$\sum_{c\in S_C}\text{div}(v)[c]\mathbf{p}[c]=-\sum_{f\in S_f}\text{grad}(p)\left[f\right]v\left[f\right]$$

Portanto, é comprovado que a propriedade apresentada na Eq. (22) e também válida para os operadores discretos (Eq. (25))

$$\sum_{c\in S_C}\text{div}(v)[c]\mathbf{p}[c]+\sum_{f\in S_f}\text{grad}(p)\left[f\right]v\left[f\right]=0 \quad (25)$$

Uma vez que `div` e `grad` são operadores lineares, eles podem ser representados por uma matriz esparsa $\mathbf{D}$ onde $\texttt{grad}(\texttt{x})=\mathbf{D}\texttt{x}$ e $\texttt{div}(\texttt{x})=-\mathbf{D}^T\texttt{x}$ (BAO, 2017).

No caso de condições de contorno não nulas é necessário incluir o número da célula c = 0 para se referir ao exterior. Incluindo o operador gradiente para as faces externas na Eq. (25), obtém-se:

$$\sum_{c\in S_C}\text{div}(v)[c]\mathbf{p}[c]+\sum_{f\in S_f}\text{grad}(p)\left[f\right]v\left[f\right]=$$
$$\sum_{f\in\bar{S}_f/S_f}\left(\mathbf{p}\left[C_1(f)\right]\mathbf{1}_{\{C_2(f)=0\}}-\mathbf{p}\left[C_2(f)\right]\mathbf{1}_{\{C_1(f)=0\}}\right)v\left[f\right] \quad (26)$$

onde $\bar{S}_f$ inclui faces internas e externas

A Eq. (26) representa a equação discreta para a Eq. (27).

$$\int_\Omega p\nabla\cdot\vec{v}d\vec{x}+\int_\Omega\vec{v}\cdot p\nabla d\vec{x}=\int_{\partial\Omega}p\vec{v}\cdot\vec{n}ds \quad (27)$$

Adicionalmente, para escrever o esquema TPFA, conforme apresentado na seção 3.5.7, é necessário definir a transmissibilidade $T\left[f\right]$ que descreve o fluxo através da face f dada a

diferença de uma unidade de pressão entre as células vizinhas $C_1(f)$ e $C_2(f)$, conforme apresentado na Eq. (18) e Eq. (19).

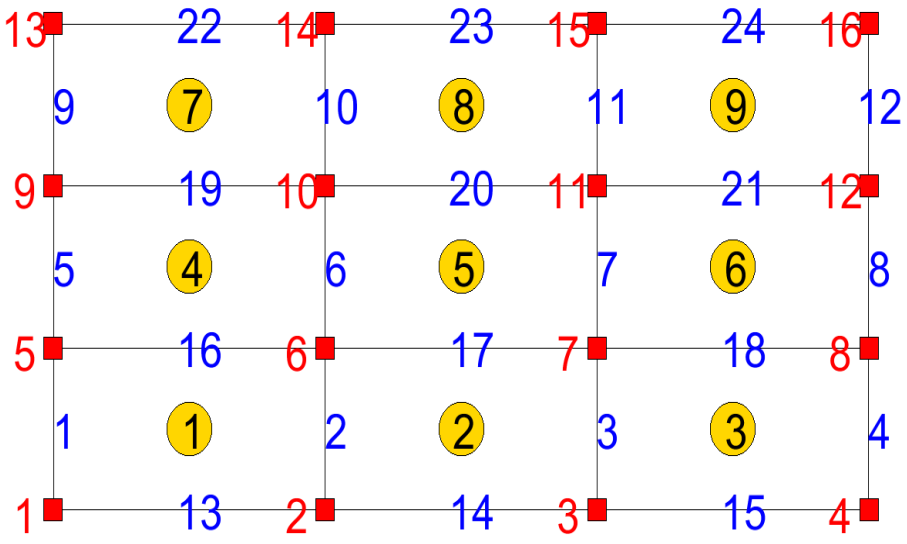
Portanto a discretização da Eq. (8), utilizando os operadores abstratos, é simplesmente escrita como:

$$\mathtt{div}(v) = q \qquad v = -T\,\mathtt{grad}(p) \tag{28}$$

Para exemplificar como são criadas as matrizes dos operadores discretos, considere uma malha cartesiana 2D gerada no MRST e representada na Figura 31. As matrizes esparsas que representarão os operadores gradientes e divergente, algebricamente, são obtidas conforme indicado no Quadro 15 e Quadro 16. Numericamente, no MRST, ele é construído pelo código apresentado no Algoritmo 28.

Nas seções posteriores, referentes ao módulo de diferenciação automática, será possível perceber que a utilização dos operadores discretos permite escrever códigos para solução de problemas de fluxo, de uma forma compacta e objetiva.

Figura 31 – Malha Cartesiana 2D [3 x 3]



Fonte: A autora (2020).

Quadro 15 – Operador gradiente

F	$C_1(f)$	$C_2(f)$
1	0	1
2	1	2
3	2	3
4	3	0
5	0	4
6	4	5
7	5	6
8	6	0
9	0	7
10	7	8
11	8	9
12	9	0
13	0	1
14	0	2
15	0	3
16	1	4
17	2	5
18	3	6
19	4	7
20	5	8
21	6	9
22	7	0
23	8	0
24	9	0

Somente faces internas (12 faces):

$$\text{grad}(\text{p})[f] = \text{p}[C_2(f)] - \text{p}[C_1(f)]$$

1. $\text{grad}(\text{p})[f = 2] = \text{p}[c = 2] - \text{p}[c = 1]$
2. $\text{grad}(\text{p})[f = 3] = \text{p}[c = 3] - \text{p}[c = 2]$
3. $\text{grad}(\text{p})[f = 6] = \text{p}[c = 5] - \text{p}[c = 4]$
4. $\text{grad}(\text{p})[f = 7] = \text{p}[c = 6] - \text{p}[c = 5]$
5. $\text{grad}(\text{p})[f = 10] = \text{p}[c = 8] - \text{p}[c = 7]$

M

12. $\text{grad}(\text{p})[f = 21] = \text{p}[c = 9] - \text{p}[c = 6]$

Dispondo na matriz de dimensão $n_f \times n_c$, ou seja, 12x9:

$$grad(x) = \begin{bmatrix} -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 \\ -1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 1 \end{bmatrix} \begin{matrix} \frac{\partial}{\partial x} \\ \\ \\ \\ \\ \\ \frac{\partial}{\partial y} \\ \\ \\ \\ \\ \\ \end{matrix}$$

Fonte: A autora (2020).

Quadro 16 – Operador divergente

c	F(c)
1	1
1	13
1	2
1	16
2	2
2	14
2	3
2	17
3	3
3	15
3	4
3	18
4	5
4	16
4	6
4	19
5	6
5	17
5	7
5	20
6	7
6	18
6	8
6	21
7	9
7	19
7	10
7	22
8	10
8	20
8	11
8	23
9	11
9	21
9	12
9	24

$$\text{div(v)}[c=1] = \sum_{f \in F(c)} \text{v}\left[f\right]\mathbf{1}_{\{c=C_1(f)\}} - \sum_{f \in F(c)} \text{v}\left[f\right]\mathbf{1}_{\{c=C_2(f)\}}$$

$$\begin{aligned}
\text{div}\left(v\right)\left[1\right] &= v\left[2\right] + v\left[16\right] \\
\text{div}\left(v\right)\left[2\right] &= v\left[3\right] + v\left[17\right] - v[2] \\
\text{div}\left(v\right)\left[3\right] &= v\left[18\right] - v\left[3\right] \\
\text{div}\left(v\right)\left[4\right] &= v\left[6\right] + v\left[19\right] - v[16] \\
\text{div}\left(v\right)\left[5\right] &= v\left[7\right] + v\left[20\right] - v[6] - v[17] \\
\text{div}\left(v\right)\left[6\right] &= v\left[21\right] - v\left[7\right] - v[18] \\
\text{div}\left(v\right)\left[7\right] &= v\left[10\right] + v\left[19\right] \\
\text{div}\left(v\right)\left[8\right] &= v\left[11\right] - v\left[10\right] - v[20] \\
\text{div}\left(v\right)\left[9\right] &= -v\left[11\right] - v\left[21\right]
\end{aligned}$$

Dispondo na matriz de dimensão $n_c \times n_f$, ou seja, 9x12:

$$div(x) = \begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
-1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & -1 & 1 & 0 & 0 & 0 & -1 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\
0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & -1 & 0 \\
0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & -1
\end{bmatrix}$$

Note que: $\text{div}(v) = -\left[\text{grad}(x)\right]^T$

Fonte: A autora (2020).

Algoritmo 28 – Comando para criar operadores no MRST

```
%Seleciona o vetor [nf x 1] constando das células vizinhas
%de cada face. No caso de pertencer à fronteira, essas
%células são retiradas
C = G.faces.neighbors;
C = C(all(C~= 0, 2), : );

% número de faces e número de células
nf = size(C,1);
nc = G.cells.num;


% cria a matriz esparsa:
D = sparse([(1:nf)'; (1:nf)'], C,ones(nf,1)*[-1 1], nf, nc);

%S = sparse(i,j,s,m,n,nzmax)
%S(i(k),j(k)) = s(k)
%nzmax = length(s)
%i = [(1:nf)'; (1:nf)'], repete 2 vezes o índice da célula;
%j = C, índices das células em duas colunas;
%s = ones(nf,1)*[-1 1], matriz de [ nf x 2], com primeira
%coluna composta por -1 e segunda por 1.
%[m x n] = [nf x nc] ->
%[número de linhas x número de colunas] da matriz.

% Cria os operadores gradiente e divergente
grad = @(x) D*x;
div = @(x) -D'*x;
```

3.7 RÁPIDA PROTOTIPAGEM

Em linguagens de programação compiladas é comum que se invista muito tempo escrevendo o código, sem as abstrações matemáticas para as equações e utilização de loops para realização de operações que atuam individualmente no conjunto das matrizes. Linguagens orientado-objeto, como C++, oferecem funcionalidades que podem ser usadas para fazer abstrações que sejam flexíveis e computacionalmente eficientes, possibilitando desenvolver algoritmos com construções matemáticas de alto nível (LIE, 2016).

O MATLAB foi escolhido para o MRST, pois apresenta uma linguagem simples, intuitiva e muitos mecanismos que aumentam a produtividade, por exemplo, fornecem abstrações para vetores e matrizes, sem processo complicado de construção, e permite uma análise interativa para a escrita de códigos compactos e legíveis.

No entanto, é importante destacar que se faz necessário o uso de mecanismos eficientes, como vetorização, mapeamento indireto e funções avançadas do MATLAB para que se tenha

um código realmente eficiente. Aqui serão destacadas duas funções do MATLAB, bastante utilizadas nos códigos presentes no MRST: `bsxfun` e `accumarray`.

A função `bsxfun` aplica a operação binária elemento a elemento, conforme indica o Algoritmo 29. Destaca-se, no entanto, que desde MATLAB® R2016b, podem-se utilizar diretamente operadores ao invés dessa função.

Algoritmo 29 – Comando bsxfun

```
1.          % C = bsxfun(fun,A,B);
2.          % Exemplo desvio dos elementos da matriz com
3.          %relação a média da coluna
4.      A = [1 2 10; 3 4 20; 9 6 15];
5.      C = bsxfun(@minus, A, mean(A));
6.      D = bsxfun(@rdivide, C, std(A))
7.
```

A função `accumarray` constrói uma matriz a partir da acumulação, ou seja, `accumarray(subs,val)` retorna a acumulação dos elementos no vetor `val` para os índices descrito em `subs`. O Algoritmo 30 apresenta como utilizar esse comando.

Algoritmo 30 – Comando accumarray

```
1.      val = 101:105';
2.      subs = [1; 3; 4; 3; 4]
3.      A = accumarray(subs,val)
4.      % Soma os valores em val que possuem o mesmo índice em subs.
5.      %A(3) =102+103 = 206.
6.      % Resulta: [101;0;206;208]
7.
8.      val = 101:106';
9.      subs = [1 1; 2 2; 3 2; 1 1; 2 2; 4 1]
10.     A = accumarray(subs,val)
11.     % Resulta: A=[205 0;0 207;0 103; 106 0]
12.     B = accumarray(subs,val)
13.     % Resulta: B=[205 0 0 0;0 207 0 0;0 103 0 0; 106 0 0 0]
14.
```

Como exemplo, Lie (2016) apresenta a contagem do número de pontos em cada octante, para um vetor gerado randomicamente, conforme expresso no Algoritmo 31.

Algoritmo 31 – Exemplo bsxfun e accumarry

```
1.
2.      n = 5000000;
3.      pt = randn(n,3);
4.      I = sum(bsxfun(@times, pt>0, [1 2 4]),2)+1;
5.      num = accumarray(I,1);
6.
```

3.8 PROGRAMAÇÃO ORIENTADO OBJETO

Até o presente momento, foi apresentada a realização da simulação com programação procedural. No entanto, nos últimos anos, o MRST tem trabalhado em uma nova estrutura baseada na programação orientado objeto.

Como será apresentado a seguir, a programação orientada objeto possui uma fácil aderência à diferenciação automática, estando esta disponibilizada no MRST a partir da versão 2016a.

Nessa seção serão apresentados alguns conceitos introdutórios sobre programação orientada objeto, para a familiarização da sua utilização no MRST.

3.8.1 Objeto e Classe

A Programação Orientada Objeto – POO (*Object Oriented Programming*) é um paradigma de programação, assim como, a Programação Procedural. Essa última está baseada no conceito de chamadas a procedimentos, que simplesmente possuem passos computacionais a serem executados, enquanto a primeira é baseada em objetos que se intercomunicam.

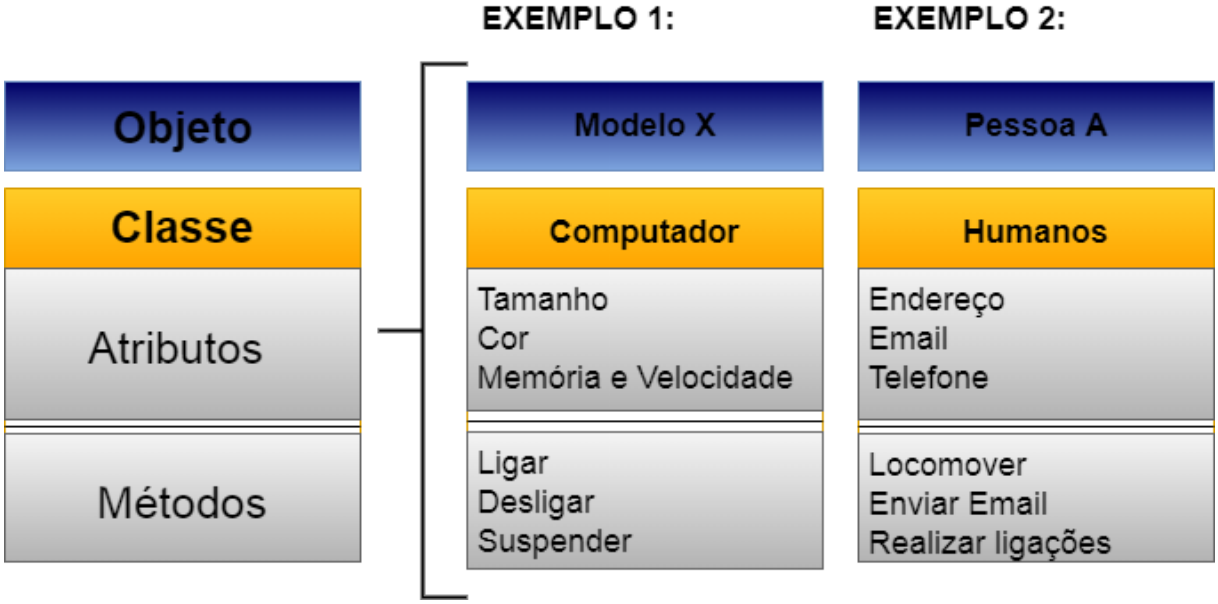
A análise e projeto orientados a objetos identifica o melhor conjunto de objetos para descrever um sistema de software. O funcionamento deste sistema se dá através do relacionamento e troca de mensagens entre estes objetos. (PET-TELE, 2009)

Na POO deve-se no primeiro momento estabelecer quais objetos serão necessários. Os objetos podem ser entendidos como a representação computacional de algo que existe. Os objetos são instâncias das classes, ou seja, criados a partir de uma classe. As classes são basicamente uma estrutura com dados (propriedades) e funções, denominados respectivamente, como atributos e métodos. As classes são elementos centrais para a metodologia de programação orientada objeto (SAVITCH, 2004).

Uma classe é um gabarito, um modelo, para a definição de objetos. Considerando uma analogia simples para facilidade de compreensão, considere, por exemplo, o computador como uma classe. Ele possui diversas características, como: cor, tamanho da tela, velocidade, memória, dentre outros, que se constituem como as propriedades ou atributos dessa classe. Além disso, o computador possui diversas funcionalidades: ligar, suspender, abrir documentos, desligar, entre outros. Essas ações são correspondentes aos métodos, visto que descrevem o

comportamento do objeto. Podem ser criados diversos objetos do tipo computador, de diferentes modelos. A Figura 32 ilustra como esses conceitos podem ser compreendidos.

Figura 32 – Programação Orientada a objeto – classes, atributos e métodos.



Fonte: A autora (2020).

É através da mensagem, ou seja, chamada do procedimento, que o método é ativado e o objeto executa suas ações.

3.8.2 Princípios da Programação Orientado Objeto

A POO é baseada nos princípios de encapsulamento/abstração, herança e polimorfismo.

Encapsulamento consiste em definir uma classe, de modo que, a implantação das funções membros e dos dados nos objetos não seja conhecida do programador que utiliza a classe. Uma das formas de aplicar esse princípio é tornar as variáveis-membros privadas, ou seja, não podem ser referenciados por nomes em nenhum lugar, exceto dentro das definições das funções-membros da classe (SAVITCH, 2004). Dessa forma, o usuário não tem acesso a detalhes internos do objeto. Todo acesso aos dados é feito através da chamada a um método definido pelo objeto. Essa propriedade é importantíssima, pois mudanças na implementação interna do objeto (que preservem a interface externa) não afetam o resto do sistema, além disso, garantem a proteção dos dados ao usuário, evitando que algum dado seja alterado por engano ou de forma descontrolada.

Abstração é o processo de identificar as propriedades importantes para o problema, se concentrando apenas nas características relevantes. Logo, objetos somente revelam mecanismos internos que são relevantes para uso de outros objetos, não revelando a implementação interna do código.

A herança é um processo pelo qual uma nova classe, denominada como classe derivada ou subclasse, é criada a partir de outra classe, chamada classe-base ou superclasse. Uma classe derivada pode possuir automaticamente todas as variáveis-membros e todas as funções-membros ordinárias que a classe-base possui, e pode ter funções e variáveis adicionais (SAVITCH, 2004). Esse recurso está relacionado com a possibilidade de se reutilizar código. Na subclasse é possível adicionar métodos e variáveis à superclasse, ou redefinir métodos herdados, ou seja, refazer sua implementação o que é chamado de *overriding* (PET-TELE, 2009).

A hierarquia de classes está definida de modo que as classes mais genéricas estão no topo e as mais específicas na base. Dessa forma, as classes de níveis mais baixos herdam as características das classes acima delas.

O polimorfismo permite escrever programas que processam objetos que compartilham a mesma superclasse (direta ou indiretamente) como se todos fossem objetos da superclasse; isso pode simplificar a programação (DEITEL e DEITEL, 2010, p. 305).

No polimorfismo, uma ou mais classes respondem a uma mesma mensagem de forma diferente. Um mesmo método é executado de forma diferente de acordo com a classe do objeto que aciona o método e com os parâmetros passados para o método. (CARVALHO, 2012). Um conceito importante de ser definido é o de *overloading* (sobrecarga), que é a habilidade de criar métodos com mesmos nomes, mas diferindo-se entre si pelos dados de entrada (parâmetros).

3.8.3 Programação Orientado Objeto no MATLAB

No MATLAB, os componentes da Programação Orientado Objeto são definidos conforme apresentado no Quadro 17.

Quadro 17 – MATLAB - POO

Programação Orientada Objeto – MATLAB
Definição da classe.

Programação Orientada Objeto – MATLAB

```
classdef (ClassAttributes) ClassName < SuperClass
%Ex: (ClassAttributes) =(Sealed)
    ...
End
```

Definição das propriedades, incluindo valores padrões.

```
classdef  ClassName
    properties (PropertyAttributes)
    %Ex:(PropertyAttributes)=(SetAccess = private)
        Prop1 = date
        ...
    end
    ...
End
```

Definição de métodos.

```
classdef  ClassName
    methods (MethodAttributes)
        % Ex: (MethodAttributes) =(SetAccess = private)
        function obj = myMethod(obj)
        ...
    end
    ...
End
```

Há ainda a definição de blocos denominados *event* e *enumeration*, que seguem o mesmo padrão apresentado acima.

Fonte: A autora (2020).

A seguir são apresentados três exemplos com o intuito de criar familiaridade com a programação orientada objeto, bem como, com seus conceitos em casos práticos.

O primeiro exemplo consiste em um caso básico, com a definição de uma propriedade, que contém o valor armazenado no objeto da classe e três métodos, um é denominado construtor, responsável por criar objetos da classe e validar o valor da propriedade. Os demais permitem realização de operações com os objetos: um arredonda o valor para duas casas decimais e o outro multiplica o valor da propriedade por um número específico (MATHWORKS, 2015).

Algoritmo 32 – MATLAB – POO – Exemplo 1

```
classdef BasicClass
  properties
  Value
  end
methods
   function obj = BasicClass(val)
     if nargin > 0
       if isnumeric(val)
       obj.Value = val;
       else
       error('Value must be numeric')
       end
     end
   end

   function r = roundOff (obj)
   r = round([obj.Value],2)
   end

   function r = multiplyBy (obj,n)
   r = [obj.Value]*n
   end
end
 end
```

Fonte: MATHWORKS (2015).

Para criar um objeto da classe, atribuir valores e chamar os métodos deve-se proceder conforme indicado a seguir.

Algoritmo 33 – MATLAB – POO – Exemplo 1 (utilização)

```
% cria um objeto da classe:
%a = BasicClass with properties:
%Value: []
a = BasicClass

% atribui um valor a propriedade Value, usado o formato:
%(nome_da_classe.proriedade)
a.Value = pi/3;

% chamar os métodos:
%o objeto é passado como argumento da função.
%Pode-se ainda utilizar a notação como pontos em que o objeto não é
%passado explicitamente como argumento
roundOff(a)
multiplyBy(a,3)
a.multiplyBy(3)

% criar um objeto da classe utilizando o método construtor
b = BasicClass(pi/4);
```

Fonte: MATHWORKS (2015).

Pode-se ainda utilizar o conceito de *overload functions* para implementar funcionalidades existentes, com mesmo nome de funções existentes no MATLAB, como por exemplo, adição para soma dois objetos da classe `BasicClass`.

Algoritmo 34 – MATLAB – POO – Exemplo 1 (*overloading*)

```
%Sem definir o método dará erro:
a = BasicClass(pi/3);
b = BasicClass(pi/4);
>> a + b
Error: Undefined operator '+' for input arguments of type
'BasicClass'.

%Acrescentando a função plus em métodos
method
 function r = plus(o1,o2)
 r = [o1.Value] + [o2.Value];
 end
end

%Definido o método é possível realizar a operação
a = BasicClass(pi/3);
b = BasicClass(pi/4);
>> a + b
ans =
 1.8326
```

Fonte: MATHWORKS (2015).

Utilizando o comando `help matlab/ops`,é possível ter acesso a outros nomes reservados do MATLAB, visualizando o nome e o operador correspondente, como por exemplo: `plus (+), minus (-), mtimes (*),times (.*), mpower (^), power (.^), mldivide(\), mrdivide (/), ldivide (.\), rdivide (./),` dentre outros.

O segundo exemplo consiste em criar uma classe para números racionais. Essa classe deve receber como parâmetros o denominador e o numerador (GOVINDJEE, 2013). O Algoritmo 35 exemplifica a criação dessa classe.

Algoritmo 35 – MATLAB – POO – Exemplo 2

```
classdef ratnum % RATNUM - classe de números racionais
 properties (Access=protected) % Propriedades de classe
  n % Numerador
  d % Denominador
 end
 Methods
  function r = ratnum(nume,deno)
        %Constrói objetos da classe racional, recebendo como
        %argumento o numerador e denominador
r.n = nume;
r.d = deno;
  end
```

Algoritmo 35 – MATLAB – POO – Exemplo 2

```
    end
 End
>> a = ratnum(1,3)
a =
ratnum with no properties

>> a.n
No public property 'n' for class 'ratnum'.
```

Fonte: GOVINDJEE (2013).

Note que ao criar um objeto da classe `ratnum`, com numerador 1 e denominador 3, não é possível ter acesso as suas propriedades, de modo que seu acesso só é permitido para métodos existentes na própria classe, seguindo a característica do encapsulamento descrita acima. A seguir serão apresentados alguns métodos criados na classe `ratnum`, para dispor na tela os valores do número racional, para realizar a adição de dois números racionais e métodos que permitem ao usuário obter e definir o numerador e denominador do número racional.

Algoritmo 36 – MATLAB – POO – Exemplo 2 – métodos

```
function disp(r)
  if (r.d ~= 1)
    fprintf('%d/%d\n',r.n,r.d);
  else
    fprintf('%d\n',r.n);
  end
end

function r = add(r1,r2)
r = ratnum(r1.n*r2.d +
r2.n*r1.d,r1.d*r2.d);
end


function r = plus(r1,r2)
r = ratnum(r1.n*r2.d +
r2.n*r1.d,r1.d*r2.d);
end


function n = getN(r)
  n = r.n;
end

function r = setN(r,numerator)
  r.n = numerator;
end
```

```
% mostrar o valor do número
racional. Ex:
>> a=ratnum(1,3)
a = 1/3
% Por padrão o MATLAB chama o
método disp no arquivo de
definição da classe para o
objeto. (exemplo de
overloading)
% adicionar dois números
racionais a/b + c/d = (ad+cb)/bd. Ex:
>> a = ratnum(1,3);
>> b = ratnum(1,2);
>> c = add(a,b)
c=5/6
% adicionar dois números
racionais com overloading. Ex:
>> a = ratnum(1,3);
>> b = ratnum(1,2);
>> a+b
ans=5/6

% Consultar o numerador do
número racional
>> getN(a)
ans = 1

%Redefinir valor da
propriedade. Ex:
>> a=setN(a,5)
a = 5/3
```

Fonte: GOVINDJEE (2013).

Por fim, o último exemplo consiste em escrever um programa que lida com formas geométricas, como círculos e retângulos. Todas as formas possuem uma posição central e uma cor, mas diferem entre si pelos métodos que se aplicam a cada. Dessa forma, será criada uma classe para forma, com propriedades comuns deles e classes separadas para retângulos e círculos, com as especificidades de cada (GOVINDJEE, 2013). Esse exemplo evidencia a aplicabilidade do conceito de herança, conforme pode ser observado no Algoritmo 37.

Algoritmo 37 – MATLAB – POO – Exemplo 3 – herança

```
1.  classdef shape
2.   % define a classe com as propriedades do centro e cor.
3.  properties (Access=protected)
4.    x
5.    y
6.    color
7.   end
8.   methods
9.    %Possui como método um construtor, uma função para
10.   %apresentar o método e uma função para definir a cor
11.         function s=shape(x,y,color)
12. s.x = x;
13. s.y = y;
14. s.color = color;
15.   end
16.   function disp(s)
17. fprintf('The shape is centered...
18. at (%f,%f) and has color %s\n',...
19. s.x,s.y,s.color);
20.   end
21.   function color=get_color(s)
22. color = s.color;
23.   end
24.  end
25. end
26. % Define a classe circle como subclasse da classe shape
27. classdef circle < shape
28.  properties (Access=protected)
29.   r
30.  end
31.  methods
32.   % Construtor de circle
33.         function c = circle(radius,x,y,color)
34. % Construção especial para instanciar a superclasse
35.           c = c@shape(x,y,color);
36.           c.r = radius;
37.   end
38.   function disp(c)
39. % Apresenta primeiro a superclasse (opcional)
40. disp@shape(c);
41. fprintf('Radius = %f\n',c.r);
42.   end
```

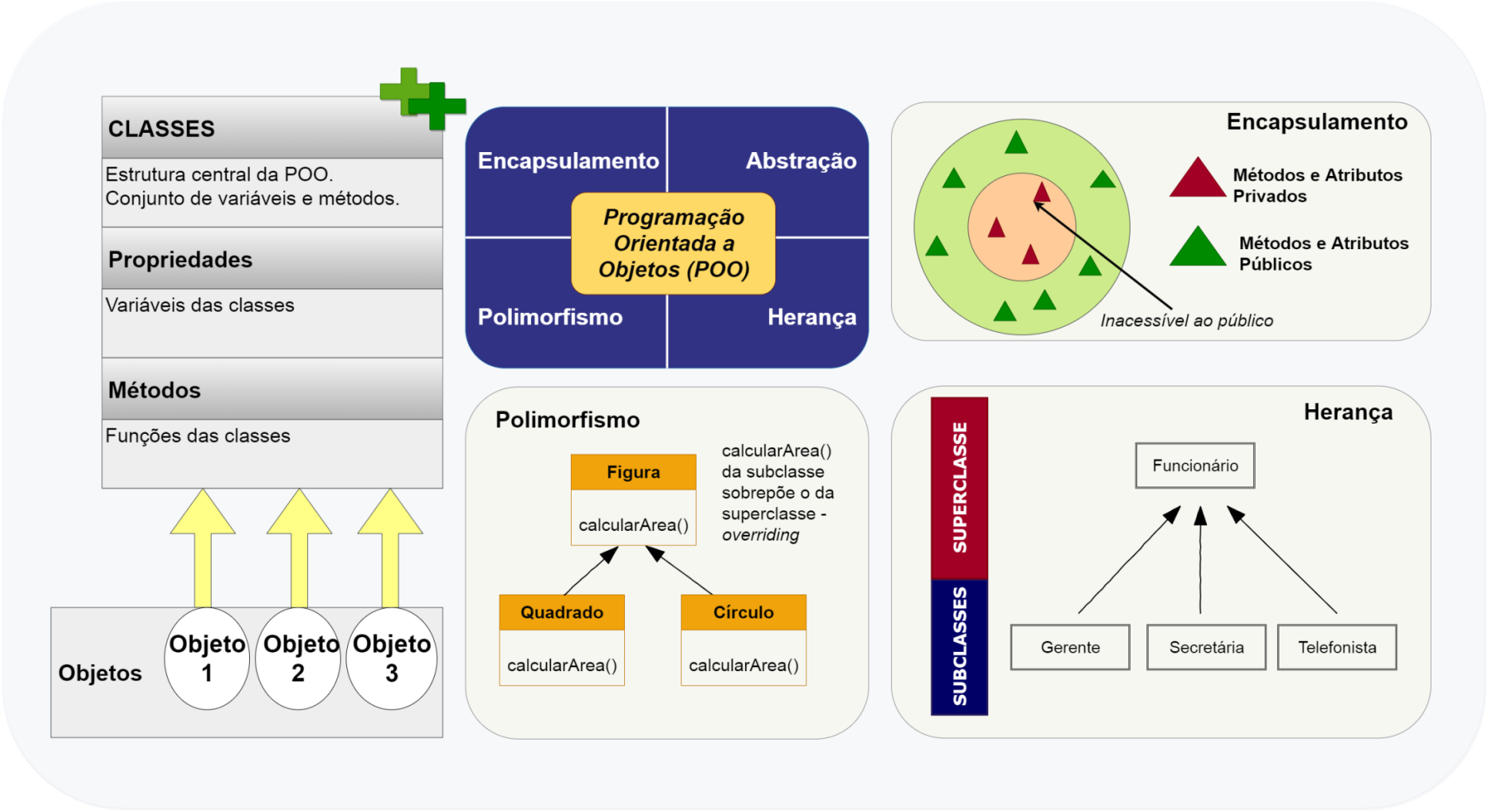
Algoritmo 37 – MATLAB – POO – Exemplo 3 – herança

```
    function a = area(c)
%Calcula a área
a = pi*c.r^2;
    end
  end
end
% Define a classe rect como subclasse da classe shape
classdef rect < shape
 properties (Access=protected)
    h
    w
 end
 methods
    % Construtor de rect
function r =rect(height,width,x,y,color)
% Construção especial para instanciar a superclasse
r = r@shape(x,y,color);
r.h = height;
r.w = width;
    end
    function disp(r)
% Apresenta primeiro a superclasse (opcional)
disp@shape(r);
fprintf('Height = %f and ...
    Width = %f\n',r.h,r.w);
    end
    function a = area(r)
a = r.w*r.h;
    end
 end
end
>> c = circle(3,1,1,'blue')
c =
The shape is centered at (1.000000,1.000000) and has color
blue
Radius = 3.000000

>> get_color(c)
ans = 'blue'

>> r=rect(1,2,0,0,'Black')
r =
The shape is centered at (0.000000,0.000000) and has color
Black
Height = 1.000000 and Width = 2.000000

>> get_color(c)
ans = 'Black'

%Note que esse método está definido apenas na superclasse, mas a
subclasse herda suas funções
```

Fonte: GOVINDJEE (2013).

A Figura 33 apresenta um resumo das principais características apresentadas nessa seção.

Figura 33 – Programação Orientada A Objeto



Fonte: A autora (2020).

3.9 DIFERENCIAÇÃO AUTOMÁTICA

Pela diferenciação automática as funções analíticas, independente de sua complexidade, podem ser divididas em um conjunto limitado de operações (+, -, *, /, etc) e um conjunto de funções como exponencial, logaritmo e funções trigonométricas, para as quais, as derivadas são conhecidas. Dessa forma, é possível obter a derivada da função, simultaneamente ao valor. (LIE, 2016).

A diferenciação automática (*Automatic differentiation - AD*) usa fórmulas exatas junto com valores de ponto flutuante, em vez de cadeias de expressão como na diferenciação simbólica e não envolve erro de aproximação como na diferenciação numérica usando quocientes de diferenças (NEIDINGER, 2010). Dessa forma, a partir da diferenciação automática é possível obter valores acurados para as derivadas e jacobianas que podem ser usadas em outros métodos numéricos, como, por exemplo, no método de Newton.

A ideia chave é que o cálculo da derivada aconteça automaticamente quando se interpreta uma função. São utilizadas regras básicas de derivadas do cálculo, como a regra da cadeia para obtenção das derivadas. O Quadro 18 mostra como se processa o cálculo da função $y = x \sin\left(x^2\right)$ e sua derivada em x = 3.

Quadro 18 – Ideia da diferenciação automática

$x = 3$	$x' = 1$
$y_1 = x^2 = 9$	$y_1' = 2x\,x' = 6$
$y_2 = \sin\left(y_1\right) = 0.4121$	$y_2' = \cos\left(y_1\right)y_1' = -5.46668$
$y = x\,y_2 = 1.2363$	$y' = x'y_2 + x\,y_2' = -15.9883$

Fonte: NEIDINGER (2010).

A programação orientada a objeto permite concretizar essa ideia expressa de obtenção simultânea das derivadas de uma forma simples.

Utilizando programação procedural, poderia se definir um vetor 1x2 para armazenar as linhas e colunas e criar novas funções `adtimes` e `adsin` para realizar as operações e obter o valor das funções e derivadas. Dessa forma, para o exemplo acima, seria definido $x = [3,1]$ e calculado `adtimes(x, adsin(adtimes(x,x)))`.

Na POO se define uma nova classe com propriedades de valor e derivada, bem como os métodos associados a esses atributos. Além disso, pode ser utilizada a propriedade denominada

overloading, de forma que as operações padrões como `*` e `sin`, são definidas na classe e permitem obter não só o valor da função, mas também o valor da derivada. Dessa forma, para o exemplo apresentado $f\left(x\right)=x\sin\left(x^2\right)$, é criado um objeto com valor 3 e derivada 1 ao realizar a operação `x*sin(x*x)`.

O Quadro 19 mostra exemplos de como são realizadas as operações para se obter o valor numérico das funções e suas derivadas em um ponto a.

Quadro 19 – Exemplo de objeto com valor e derivada

Função		Valor e derivada do objeto	
		val	Der
$f\left(x\right)=c$	c=	c	0
$f\left(x\right)=x$	x=	a	1
$u\left(x\right)$	u=	$u\left(a\right)$	$u'\left(a\right)$
$v\left(x\right)$	v=	$v\left(a\right)$	$v'\left(a\right)$
$u\left(x\right)*v\left(x\right)$	u*v=	$u\left(a\right)*v\left(a\right)$	$u'\left(a\right)*v\left(a\right)+u\left(a\right)*v'\left(a\right)$
$\sin\left(u\left(x\right)\right)$	sin(u)=	$\sin\left(u\left(a\right)\right)$	$\cos\left(u\left(a\right)\right)*u'\left(a\right)$

Fonte: Adaptado de NEIDINGER (2010).

O Algoritmo 38 exemplifica como, através da Programação Orientada a Objeto, é possível executar a ideia da diferenciação automática de forma simples.

Algoritmo 38 – Exemplo da classe de dif. automática no MATLAB

```
1.
2.  % define a classe com as propriedades de valor e derivada.
3.   classdef valder
4.   properties
5.    val
6.    der
7.   end
8.   methods
9.
10.   %Possui como método um construtor
11.   function obj=valder(a,b)
12.     if nargin == 0
13.     % cria um objeto vazio se não recebe argumento
14.     obj.val = [];
15.     obj.der = [];
16.     elseif nargin == 1
17.     % cria uma função constante com derivada igual a zero
18.     obj.val = a;
19.     obj.der = 0;
20.     else
21.     % cria um objeto com o valor da função e o valor da derivada
```

Algoritmo 38 – Exemplo da classe de dif. automática no MATLAB

```
      obj.val = a;
      obj.der = b;
      end
    end

% define o método sin para objetos da classe valder

    function h=sin(u)
      h = valder(sin(u.val),cos(u.val)*u.der);
    end

    function h =mtimes(u,v) % overloads *
      if ~isa(u,'valder')
      % u sendo escalar


      h = valder(u*v.val,u*v.der);
      elseif ~isa(v,'valder')
      % v sendo escalar
      h = valder(v*u.val,v*u.der);
      else
      % u e v da classe valder
      h = valder(u.val*v.val,u.der*v.val+u.val*v.der);
      end
    end
  end
end

% Para o exemplo do Quadro 18:

>> x=valder(3,1);
>> x*sin(x*x)
ans =
valder with properties:

  val: 1.2364

  der: -15.9882
```

Fonte: NEIDINGER (2010).

Para expandir a diferenciação automática em casos mais complexos, o MATLAB oferece diferentes bibliotecas, por exemplo, ADiMat (DARMSTADT, 2008?), ADMAT (VERMA, 1999), MAD (TOMLAB, 2008?) ou do MATLAB Central (FINK, 2007).

3.9.1 Diferenciação Automática no MRST

O MRST possui sua própria implementação para a diferenciação automática, que foi desenvolvida especialmente para resolver o fluxo de equações, ou seja, trabalha com vetores longos e matrizes esparsas (LIE, 2016).

De acordo com Bao *et al.* (2017) e Lie (2016), a classe de diferenciação automática no MRST (AD) difere das demais bibliotecas, no fato de que ao invés de utilizar uma única matriz jacobiana do sistema discreto completo, o MRST utiliza uma lista de matrizes que representam as derivadas em relação a cada variável primária. Essas listas de matrizes irão formar sub-blocos na Jacobiana do sistema completo. Essa propriedade será apresentada e utilizada para a implementação do TPWL.

As vantagens dessa escolha estão relacionadas com o desempenho, visto que se evita a manipulação de grandes matrizes esparsas, e a facilidade de utilização, visto que é possível modificar apenas blocos específicos da Jacobiana referentes a uma determinada equação do sistema completo de equações.

No *mrst-core* está definida a classe ADI, que possui a seguinte sinopse: `x = ADI(value, jacobian),` onde `value` é o valor numérico do objeto e `jacobian` é sua matriz jacobiana. A definição dessa classe, com propriedades e com o primeiro método (construtor) é apresentado no Algoritmo 39.

Algoritmo 39 – Definição da classe ADI no MRST

```
1.  %Definição da classe ADI
2.  classdef ADI
3.  properties%Definição das propriedades
4.     val %valor da função como um vetor coluna de doubles
5.     jac % cell array de matrizes jacobianas esparsas
6.  end
7.  methods
8.       function obj = ADI(a,b) % construtor da classe ADI
9.             if nargin == 0 % construtor vazio
10.            obj.val  = [];
11.            obj.jac  = {};
12.            elseif nargin == 1
13.            if isa(a, 'ADI')
14.            % Somente permitido para valores que já são
15.            %da classe ADI
16.            obj = a;
17.            else
18.              error('Contructor requires 2 inputs')
19.             end
20.            elseif nargin == 2 % val + jac são fornecidos.
```

Algoritmo 39 – Definição da classe ADI no MRST

```
21.                 obj.val = a;
22.                 if ~iscell(b)
23.                  b = {b};
24.                 end
25.                 obj.jac = b;
26.               else
27.                error('Input to constructor not valid')
28.               end
29.         end
30.     end
31.  end
```

Fonte: MRST (2018).

Em geral, a classe ADI é instanciada por um conjunto de diferentes variáveis, através da função `initVariablesADI`. Utilizando `[a, b] = initVariablesADI(a,b),` a e b serão definidos como objetos da classe ADI, ou seja, ambos possuíram campos `val` e `jac`, conforme definido nas propriedades da classe. Essas variáveis começarão com o campo `jac` constando dos jacobianos de identidade em relação a si mesmos e jacobianos zero em relação a outras variáveis.

Considere, por exemplo, a expressão $z = 3e^{-xy}$ para os valores $x = 1$ e $y = 2$, utilizando diferenciação automática deve-se obter:

$$z = \left\langle z, \partial z/\partial x, \partial z/\partial y \right\rangle = \left\langle 3e^{-xy}, -3ye^{-xy}, -3xe^{-xy} \right\rangle \approx \left\langle 0.4060, -0.8120, -0.4060 \right\rangle \quad (29)$$

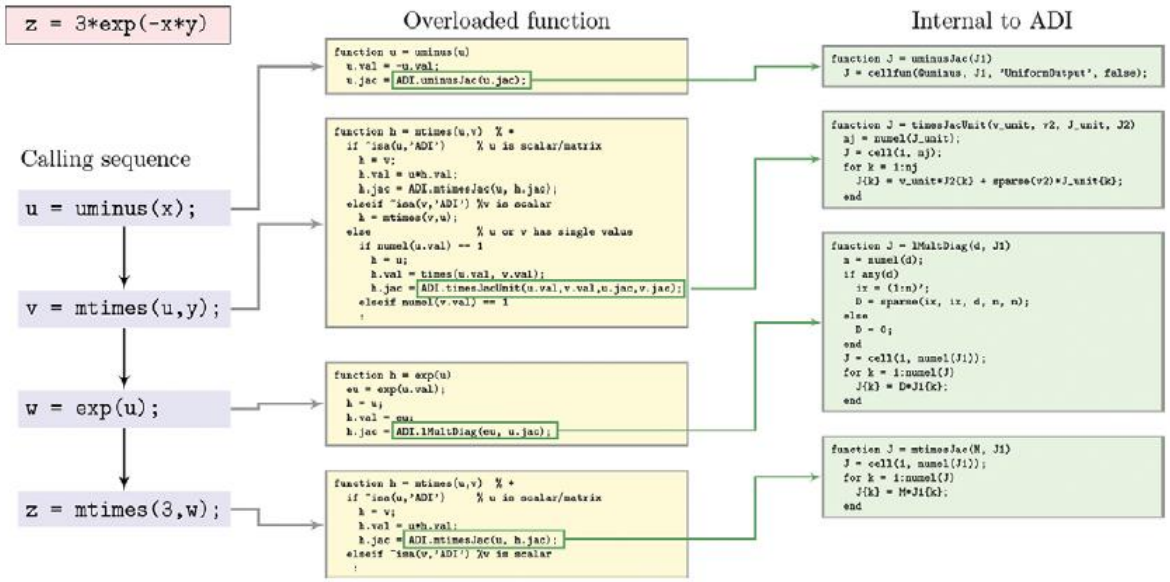
No MRST, a expressão da Eq. (29) será computada a partir do Algoritmo 40. Na primeira linha são criados `x` e `y` como dois objetos da classe ADI, enquanto a segunda expressa a operação a ser realizada. Como z depende de duas variáreis ADI, as operações descritas serão realizadas na própria classe, utilizando o conceito de *overloading*, ou seja, são utilizadas as funções `uminus, mtimes` e `exp` que possuem o mesmo nome das funções do MATLAB, mas são específicas para argumentos objetos da classe ADI. A Figura 34 apresenta como se procede as operações internas da classe. É possível ter acesso a essas funções utilizando o comando *Set Breakpoints* (Figura 14) e *Step in.* A Figura 35 apresenta o resultado para o Algoritmo 40. Note que `z` também será objeto da classe ADI.

Algoritmo 40 – Exemplo da classe ADI no MRST

```
1.    [x,y] = initVariablesADI(1,2)
2.    z = 3*exp(-x*y);
3.
```

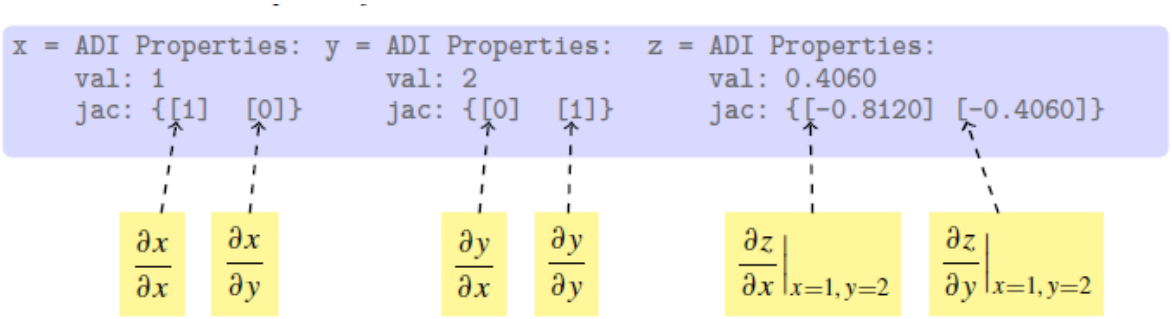
Fonte: LIE (2016).

Figura 34 – Operações na classe ADI



Fonte: LIE (2016).*Nota: Figura ilustrativa

Figura 35 – Classe ADI



Fonte: LIE (2016).

O principal uso da classe ADI é linearizar e montar sistemas de equações discretas (LIE, 2016). Dado, por exemplo, o sistema linear $Ax = b$, para resolvê-lo, deve-se escrever na forma residual (Eq. (30)). Note que, pela Eq. (31), a matriz A é igual a derivada de do sistema. Admitindo-se que um valor inicial aleatório y, é possível obter valor de x pela Eq. (32). Pela Eq. (31) e Eq. (32), obtém a Eq. (33), evidenciando que a diferenciação automática pode ser utilizada para obter a solução do sistema de equações lineares.

$$r(x) = Ax - b = 0 \tag{30}$$

$$r'(x) = A \tag{31}$$

$$r(y) = Ay - b \rightarrow r(y) = A(y - x) \rightarrow x = y - A^{-1}r(y) \tag{32}$$

$$x = y - \left[r'(x) \right]^{-1} r(y) \tag{33}$$

A seguir, serão apresentados quatro exemplos com o intuito de ratificar a importância da diferenciação automática para a obtenção de um código eficiente.

Inicialmente, o Exemplo 1 consiste em um simples sistema descrito pela Eq. (33), com solução $x = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}^T$. O Algoritmo 41 mostra os comandos aplicados para obter a solução do sistema.

$$\begin{bmatrix} 3 & 2 & -4 \\ 1 & -1 & 2 \\ -2 & -2 & 4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -5 \\ -1 \\ 6 \end{bmatrix} \tag{34}$$

Algoritmo 41 – Classe ADI e solução de Sistema de Equações

```
1.
2.  %% Sem utilização da classe ADI
3.  x = [3, 2, -4; 1, -4, 2; -2,- 2, 4]\ [-5; -1; 6]
4.
5.  %% Com utilização da classe ADI – Met. 1
6.  %Admitindo um valor inicial aleatório
7.  y = initVariablesADI(zeros(3,1));
8.  % Matriz do sistema de equações
9.  A = [3, 2, -4; 1, -4, 2; -2,- 2. 4];
10. % substituindo y no sistema de equações
11. eq = A*y + [5; 1; -6];
12. %obtendo a solução do sistema de equações usando diferenciação
13. automática, conforme Eq. (34).
14. x = -Eq. jac{1}\Eq. val
15.
16. %% Com utilização da classe ADI – Met. 2
17. y = initVariablesADI(zeros(3,1));
18. % Especificando as equações residuais linha por linha.
19. eq1 = [ 3, 2, -4]*y + 5;
20. eq2 = [ 1, -4, 2]*y + 1;
21. eq3 = [-2, -2, 4]*y - 6;
22.
23. Ex: eq1.val=5 e eq1.jac=[ 3 2 -4]
24. %concatenando as equações, ou seja, concatena tanto o campo val como
25. o campo jac
26. eq = cat(eq1,eq2,eq3);
27. %obtém a solução do sistema
28. x = -Eq. jac{1}\Eq. Val
```

Fonte: LIE (2016).

Cabe destacar que se escolheu um sistema linear apenas para ilustrar como a diferenciação automática pode ser aplicada na solução de sistema de equações, no entanto, o código não demonstra a eficiência da diferenciação automática. Sua maior aplicabilidade ocorrerá em sistemas de equações não lineares, como será demonstrado adiante. Em geral, sistemas não lineares de equações discretas precisam ser linearizados e resolvidos pelo método de Newton-Raphson (ver APÊNDICE A – MÉTODO DE NEWTON-RAPHSON).

O Exemplo 2 expressa um problema clássico de otimização, denominado por equação de Rosenbrock, descrita pela Eq. (35), também conhecida como função banana, visto que o mínimo global (a,a^2) encontra-se dentro da vale de uma forma parabólica.

$$f(x,y) = (a-x)^2 + b(y-x^2)^2 \tag{35}$$

A condição necessária para o mínimo global é que $\nabla f\left(x,y\right) = 0$, conforme expressa a Eq. (36).

$$g(x) = \begin{bmatrix} \partial_x f(x,y) \\ \partial_y f(x,y) \end{bmatrix} = \begin{bmatrix} -2(a-x) - 4bx(y-x^2) \\ 2b(y-x^2) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{36}$$

Utilizando o método de Newton-Raphson, parte-se de um valor inicial em busca da melhor aproximação $x + \Delta x$, onde Δx é obtido pela equação linearizada representada abaixo (Eq. (37)).

$$0 = g(x+\Delta x) \approx g(x) + \nabla g(x)\Delta x \tag{37}$$

O Algoritmo 42 exemplifica como é simples aplicar a diferenciação automática nesse tipo de problema.

Algoritmo 42 – Função de Rosenbrock e classe ADI

```
1.  % Definição dos parâmetros a e b da função, da tolerância tol,
2.  %do valor inicial x0 e do incremento.
3.  [a, b, tol] = deal(1, 100, 1e-6);
4.  [x0, incr] = deal([-.5; 4]);
5.
6.  %Determina o valor de x
7.  while norm(incr)>tol
8.  x = initVariablesADI(x0);
9.  eq1= 2*(a-x(1)) - 4*b.*x(1).*(x(2)-x(1).^2);
10. eq2 = 2*b.*(x(2)-x(1).^2);
11. eq = cat( eq1, eq2);
12. incr = - Eq. jac{1}\Eq. val;
13. x0 = x0 + incr;
14. end
```

Fonte: LIE (2016).

O Exemplo 3 apresenta o uso de diferenciação automática aplicado em um problema de otimização sujeito a restrições não lineares. O objetivo é minimizar o volume do pórtico representado na Figura 36. Matematicamente, a formulação do problema é dada pelas seguintes equações:

$$\text{Minimizar } f(x) = 2\alpha L x_1^2 + L x_2^2 \qquad x_1, x_2 \geq 0 \tag{38}$$

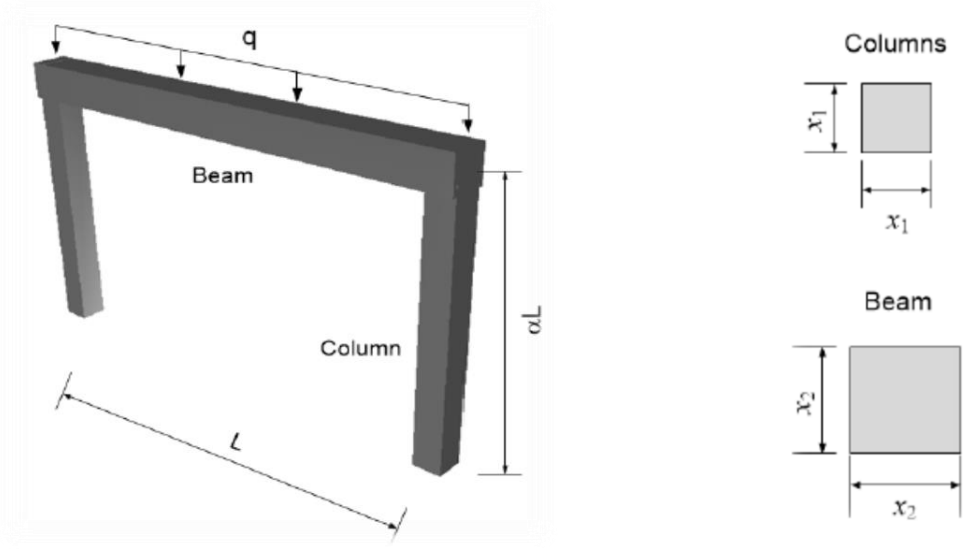
$$\text{Sujeito a: } g_1(x) = \frac{qL}{2x_1^2} + \frac{3qL^2 x_1}{6x_1^4 + 4\alpha x_2^4} - \sigma_{adm} \leq 0 \tag{39}$$

$$g_2(x) = \frac{x_1^4(x_2 + 6\alpha L)}{x_2^3(6x_1^4 + 4\alpha x_2^4)} \frac{qL}{2\alpha} - \sigma_{adm} \leq 0 \tag{40}$$

$$g_3(x) = \frac{x_1^4 x_2 + \alpha L(3x_1^4 + 6\alpha x_2^4)}{x_2^3(6x_1^4 + 4\alpha x_2^4)} \frac{qL}{2\alpha} - \sigma_{adm} \leq 0 \tag{41}$$

Dados: $\alpha = 1, L = 550\,cm, q = 15\,kN, \sigma_{adm} = 130\,kN/cm^2$

Figura 36 – Pórtico



Esse problema pode ser resolvido de forma eficaz utilizando a ferramenta de diferenciação automática do MRST e função `fmincon` pertencente a *toolbox* de otimização do MATLAB, conforme apresenta o Algoritmo 43. Inicialmente são definidas as funções para as restrições e objetivo. Note que o gradiente dessas funções também pode ser facilmente obtido

através do campo `jac`, a ser utilizado quando solicitado pelo otimizador. Dessa forma, a diferenciação automática permite a escrita de um código mais sintético e menos redundante.

Algoritmo 43 – Otimização sujeita a restrições não lineares - ADI

```
1.
2.      function [c,ceq,GC,GCeq] = restricoes(x)
3.
4.      %Define restrições não-lineares c(x) <= 0 e ceq(x) = 0, e seus
5.      gradientes.
6.
7.      alfa = 1; q = 15;
8.      L = 550; sigadm = 130;
9.
10.     x = initVariablesADI(x);
11.
12.     g1 = 0.5*q*L./(x(1).^2) + ...
13.     (q*L^2./(12+8*alfa*x(2).^4./x(1).^4))./(x(1).^3./6)...
14.      - sigadm;
15.     %caso o gradiente seja solicitado
16.     g2 = q*L^2./(12+8*alfa*x(2).^4./x(1).^4)./(alfa*L) ...
17.     ./ (x(2).^2) + q*L^2./(12+8*alfa*x(2).^4./x(1).^4) ...
18.     ./(x(2).^3./6) - sigadm;
19.
20.     g3 =(q*L^2./(12+8*alfa*x(2).^4./x(1).^4))./(alfa*L)...
21.     ./(x(2).^2) +(q*L^2./8 - q*L^2./ ...
22.     (12+8*alfa*x(2).^4./x(1).^4))./(x(2).^3./6) - sigadm;
23.
24.     g = [g1; g2; g3]; c = g.val; ceq = [];
25.
26.      if nargout > 2
27.      GC = full(g.jac{1}');
28.      GCeq = [];
29.       end
30.     end
31.
32.
33.
34.     function [v, dv] = volportico(x)
35.     %Define a função objetivo
36.     alfa = 1;
37.     L = 550;
38.
39.     x = initVariablesADI(x);
40.     vol = 2*alfa*L*x(1).^2 + L*x(2).^2;
41.     v = vol.val;
42.      %caso o gradiente seja solicitado
43.       if nargout > 1
44.      dv = full(vol.jac{1});
45.       end
46.     end
47.
48.
49.
50.
```

Algoritmo 43 – Otimização sujeita a restrições não lineares - ADI

```
51.
52.   %% Otimizador sqp
53.   %utilizando a função fmincon
54.
55.   options = optimoptions('fmincon','Display','iter', ...
56.   'Algorithm','sqp','SpecifyObjectiveGradient',true, ...
57.   'SpecifyConstraintGradient',true,...
58.   'PlotFcns',{@optimplotfval,@optimplotstepsize});
59.   %ponto inicial
60.
61.   x0 = [7;30];
62.
63.   lb = [0,0]; %limite inferior
64.   ub = [Inf,Inf]; %limite superior
65.
66.   x = fmincon(@volportico,x0,[],[],[],[],lb,ub, ...
67.   @restricoes,options);
68.
69.   fprintf('x(1): %.3f cm\nx(2): %.3f cm\nVolume: %.3f
70.   cm³\n'...
71.   ,x(1),x(2),volportico(x));
72.
73.   >>x(1): 6.353 cm
74.     x(2): 29.672 cm
75.     Volume: 528629.447 cm³
76.
77.
78.
```

Por fim, o Exemplo 4, utiliza os operadores discretos, definidos na seção 3.6 em conjunto com a diferenciação automática para resolver a clássica equação de Poisson $-\nabla \cdot \left(-K \nabla P\right) = q, \quad \text{em } \Omega \subset \mathbb{R}^d$. Considerando uma malha cartesiana 2D, de 5x5 elementos de dimensões [1x1]. As condições de contorno são de fluxo nulo e existe um ponto de injeção localizado em (0,0) e produção em (1,1). Para resolver o problema deve-se proceder conforme indicado no Algoritmo 44.

Algoritmo 44 – Diferenciação automática e Operadores discretos

```
1.
2.    % Definição da malha
      G = computeGeometry(cartGrid([5 5],[1 1]));
```

Algoritmo 44 – Diferenciação automática e Operadores discretos

```
3.  C = G.faces.neighbors;
4.  C = C(all(C~= 0, 2), :);
5.  nf = size(C,1);
6.  nc = G.cells.num;
7.  D = sparse([(1:nf)'; (1:nf)'], C, ...
8.  ones(nf,1)*[-1 1], nf, nc);
9.
10. %Determina o operador divergente e gradiente
11. grad = @(x) D*x;
12. div = @(x) -D'*x;
13.
14. %Inicializa variável da Classe ADI
15. p = initVariablesADI(zeros(nc,1));
16. % Determina termo fonte um quarto de five-spot
17. q = zeros(nc, 1);
18. q(1) = 1; q(nc) = -1
19. % Escreve a equação utilizando operadores discretos
20. eq = div(grad(p))+q;
21.
22. % Faz a solução única
23. eq(1) = eq(1) + p(1);
24. % Resolve a equação
25. p = -Eq. jac{1}\Eq. val
26.
27.
```

Note que a combinação dos operadores discretos e da diferenciação automática torna o código mais compacto, podendo ser expandido com maiores facilidades para casos mais complexos.

3.10 SIMULADOR COM DIFERENCIAÇÃO AUTOMÁTICA

Todos os conceitos introduzidos até aqui, permitem abordar nessa seção a simulação utilizando diferenciação automática no MRST. Cabe destacar que esse é um ponto chave para a formulação do TPWL utilizando o MRST, visto que permitirá o conhecimento do simulador que proverá as informações requeridas para a implementação desse método semi-intrusivo.

A formulação matemática e numérica para uma simulação de reservatórios totalmente implícita admitindo o modelo *black-oil* foi apresentada na seção 1.3.

Nessa seção, baseado nesse modelo, apresentaremos um exemplo completo para a utilização do simulador *ad-blackoil*. Dessa forma, serão apresentados os trechos de códigos referentes à geração da malha, inicialização, determinação de parâmetros petrofísicos e locação dos poços.

3.10.1 Definição do reservatório

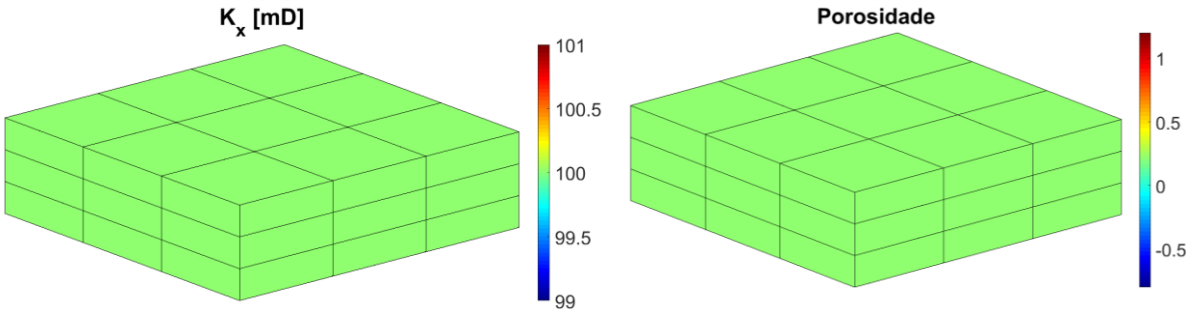
Considere uma malha cartesiana 3D, composta por [3 x 3 x 3] elementos, com dimensões de [50 m x 50 m x 5 m], com permeabilidade e porosidade constante nos valores de 100 mD e 0.2, respectivamente.

Cabe destacar que esse modelo não representa um reservatório realista, e sim um esquema idealizado para a facilidade da compreensão dos objetos utilizados na simulação com diferenciação automática. Esse modelo será referido como micro reservatório nos itens subsequentes. O Algoritmo 45 apresenta sua definição no MRST.

Algoritmo 45 – Definição do micro reservatório

```
% Definição da geometria. Seção 3.3 para maiores detalhes
cartDim = [3 3 3];
domain = cartDim.*[50 50 5];
G = cartGrid(cartDim,domain);
G = computeGeometry(G);

%Definição da permeabilidade e porosidade.
%Seção 3.4 para maiores detalhes
rock = makeRock(G, 100*milli*darcy, 0.2);
```

Figura 37 – Definição do micro reservatório



Fonte: A autora (2020).

3.10.2 Definição do Fluido

Para definir o fluido de forma básica, assim como ocorre no caso incompressível, existe uma função denominada `initSimpleADIFluid`, que recebe uma lista de parâmetros como argumento e retorna uma estrutura com campos pré-definidos de *function handles* estabelecendo as fases (água, óleo e água) e suas propriedades, como densidade e viscosidade, saturação das fases e as curvas de permeabilidade relativa. O Algoritmo 46 apresenta como

criar um fluido incompressível bifásico, com densidade da água (ρ_w) 1000 kg/m^3 e densidade do óleo (ρ_o) 850 kg/m^3, ambos com viscosidade μ=1 cP e permeabilidade relativa quadrática.

Algoritmo 46 – Definição do Fluido Classe AD

```
% Definição do fluido:
fluid = initSimpleADIFluid('phases','WO', ...
 'rho',[1000, 850].* kilogram/meter^3, ...
 'n',  [2, 2], ...
 'mu', [1, 1].*centi*poise);

%'phases' – Fases: água e óleo
%'rho' – Densidade
%'n' – Expoente da curva de permeabilidade relativa
%'mu' – Viscosidade
```

O Quadro 20 apresenta os possíveis parâmetros de entrada e o retorno da função `initSimpleADIFluid`.

Quadro 20 –Fluido Classe AD

	FUNÇÃO – `initSimpleADIFluid`
	PARÂMETROS DE ENTRADA:
phases–	Letras W, O, G, representando água, óleo e gás, respectivamente. O padrão é WOG, mas pode ser definido pelo usuário, determinando a interpretação dos valores fornecidos pelos demais parâmetros.
mu –	Vetor com valor das viscosidades [`muW, muO, muG`]. Padrão: [1 1 1].
rho –	Vetor com valor das densidades de superfície para as fases presentes. [`rhoWS, rhoOS, rhoGS`]. Padrão: [1 1 1].
n –	Vetor com os graus dos monômios, descrevendo a permeabilidade relativa para cada fase. Padrão: [1 1 1]
b –	Inverso do fator volume de formação. `rho_reservoir = rho_surf * b` Onde: `rho_reservoir` = densidade do reservatório; `rho_surf` = densidade na superfície.
c –	Fator de compressibilidade. Se especificado para cada fase, resulta no fator de formação da forma: `b(p) = b_ref * exp((p-pRef)*c` onde `b_ref` é o fator de formação conforme especificado acima `'b'` `pRef` é especificado pela palavra chave `'pRef'`. Padrão: 0.
cR –	Compressibilidade da rocha. Se fornecida, o fluido possuirá um multiplicador do volume de poros que fornece um crescimento linear do volume dos poros com a pressão, conforme indicado abaixo: `pv = pv_ref * (1 + (p-pRef)*cR)`

	FUNÇÃO – `initSimpleADIFluid`
	PARÂMETROS DE ENTRADA:
	onde `pv_ref` = poro volume especificado pela malha e modelo do reservatório; `pRef` = Pressão de referência usada em conjunto com a rocha e a rocha e o fluido.
	RETORNO:
fluid -	Estrutura contendo as seguintes funções (com X = 'W' [água], 'O' [óleo] and 'G' [gás])
	*** krW, krO, krG –** Função da permeabilidade relativa; *** rhoXS-** Densidade de X nas condições de superfície; *** bX(p)-** Inverso do fator volume formação *** muX(p) -** Função da viscosidade (constante) *** krX(s) -** rel.perm para X *** krOW, krOG -** Se o modelo é trifásico, curvas de permeabilidade relativa óleo-água e óleo-gás.

Fonte: Adaptado do MRST (2018).

A função `initSimpleADIFluid` pertence ao módulo `ad-props`. Esse módulo apresenta funcionalidades relacionadas aos cálculos de propriedades para a estrutura `ad-core`. Especificamente, o módulo implementa uma variedade de fluidos teste e funções que são usadas para criar fluidos para o simulador AD, a partir de conjuntos de dados externos (MRST, 2020).

Além da função `initSimpleADIFluid`, o módulo `ad-props` traz a função `initDeckADIFluid`, responsável por inicializar a estrutura AD para o fluido, recebendo como parâmetro de entrada a estrutura `deck` (criada pela função `readEclipseDeck` ao ler o arquivo de texto do Eclipse).

Além disso, esse módulo disponibiliza as funções `coreyPhaseRelpermAD` e `tableByPressureLinearAD`, que ser utilizadas como *handle functions* na determinação das propriedades dos fluidos.

3.10.3 Definição do Modelo

Após a criação do fluido, o próximo passo é criar um objeto que represente todo o modelo de simulação com descrição da geometria do reservatório, parâmetros petrofísicos e de fluidos, bem como os mecanismos e parâmetros que determinem as equações corretas de fluxo. Para o exemplo discutido nessa seção o escoamento é bifásico de óleo e água, dessa forma, utiliza-se o comando apresentado no Algoritmo 47.

Algoritmo 47 – Definição do Objeto Model - Classe AD

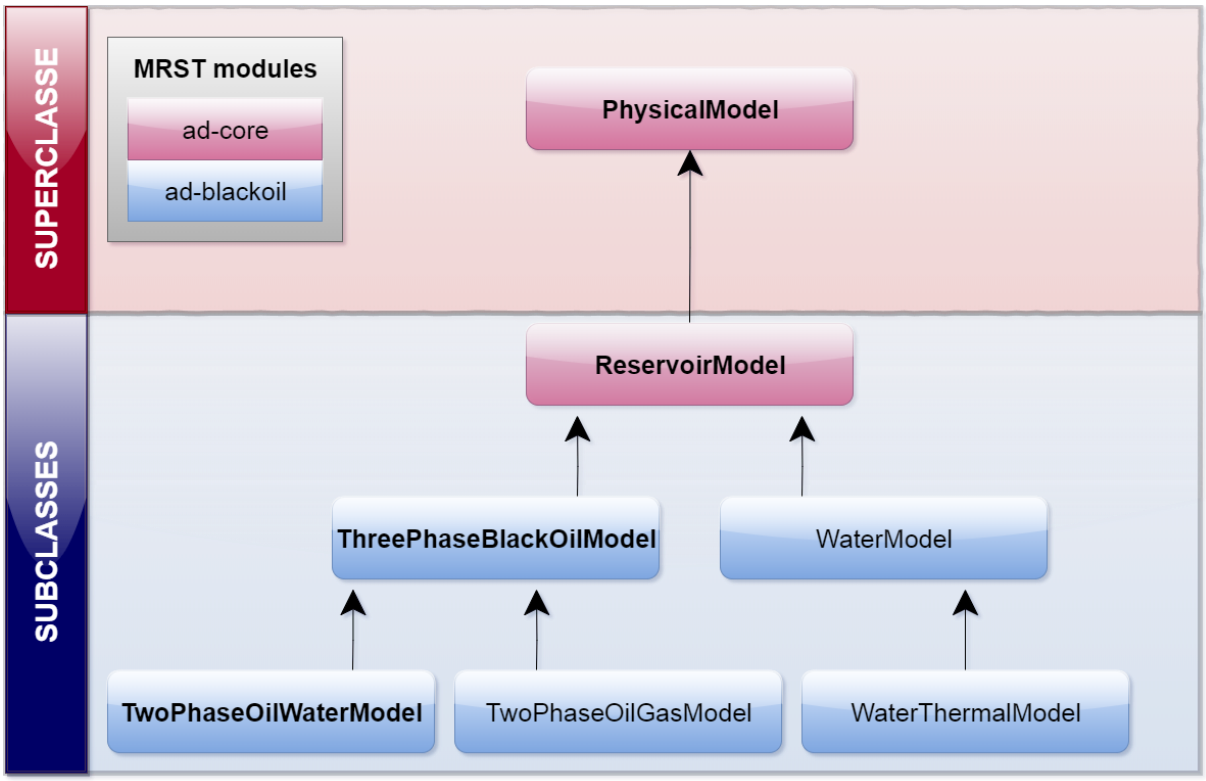
```
1.   % cria o modelo para o escoamento óleo e água
2.   model = TwoPhaseOilWaterModel(G, rock, fluid);
3.
```

O objeto `model` é uma instância da classe `TwoPhaseOilWaterModel`. Essa classe pertence ao módulo *ad-blackoil*. Conforme indicado na seção 3.1, o módulo *ad-blackoil* estende a estrutura encontrada em *ad-core* para problemas *black-oil*, adicionando modelos que implementam as equações *black-oil* para escoamento multifásico, miscível e compressível. Um dos modelos presentes é aquele utilizado para o escoamento bifásico, conforme indicado no presente exemplo.

Cabe, porém, destacar que `TwoPhaseOilWaterModel` é na verdade uma subclasse, conforme indica a Figura 38. Dessa forma, conforme os conceitos apresentados na seção 0, ela herda todas as varáveis e funções membros da superclasse. A Figura 38 traz toda a hierarquia de classes para a definição do objeto *model* no MRST AD-OO. Note que se o exemplo tratasse das fases óleo, água e gás seria utilizado o comando `ThreePhaseBlackOilModel(G,rock,fluid)`.

Um comentário importante a se adicionar é que ao se definir uma classe, estão sendo definidas as propriedades e equações a serem empregadas. Um exemplo disso, é que na superclasse `PhysicalModel` existe o método denominado `getEquations`, destinado a obter o conjunto de equações do modelo linearizado com possíveis jacobianos. Ao se utilizar a classe `ThreePhaseBlackOilModel` esse método é definido para utilizar a função `equationsBlackOil`. Porém se a opção for a subclasse `TwoPhaseOilWaterModel`, esse método é redefinido para utilizar a função `equationsOilWater`. Esse é um exemplo do conceito de *overriding*, apresentado na seção 0.

Figura 38 – Classes relacionadas ao objeto *model* na estrutura do MRST AD-OO



Fonte: A autora (2020).

As propriedades do objeto `model` são definidas pela junção das propriedades presentes nas classes `PhysicalModel`, `ReservoirModel`, `ThreePhaseBlackOilModel` e `TwoPhaseOilWaterModel`. A Figura 39 apresenta uma síntese dessas classes e de suas propriedades, que resultarão nas propriedades do modelo.

Além dos campos `G, rock, fluid` que foram fornecidos como parâmetros de entrada, são definidos *flags* e tolerâncias a respeito da discretização do modelo físico do reservatório.

Cabe destacar duas propriedades. A primeira, denominada por `FacilityModel` que consiste em uma instância da classe `FacilityModel` (que também é uma subclasse de `PhysicalModel`) criada para descrever modelos de poços e instalações de superfície. A segunda, denominada `operators`, consiste em uma estrutura criada pela função `s = setupOperatorsTPFA(G, rock)`, responsável por definir os operadores discretos, e propriedades provenientes dos reservatórios, como transmissibilidade e poro volume, necessários para o esquema TPFA, conforme descrito na seção 3.6.

Nesse ponto, cabe criar um adendo e detalhar a estrutura `operators`, em virtude da sua importância para simulação *ad-blackoil*, visto que, esses operadores permitirão a funcionalidade do modelo discreto das equações de fluxo.

Dessa forma, apresenta-se a seguir os campos da estrutura `operators`. Destaca-se que será realizado um paralelo entre a nomenclatura adotada nas seções anteriores (3.5.7 e 3.6), com aquelaq que efetivamente presente no código do MRST.

- `T_all` – representa a meia transmissibilidade ($T_{i,k}$);
- `T` – referente à transmissibilidade para todas as faces internas (T_{ik});
- `C` – matriz esparsa responsável pela transferência entre faces e células usada parar criar os operadores gradiente e divergente;
- `Grad` – operador gradiente definido pela *handle function:* `@(x)-C*x`. . Calcula o gradiente em cada interface por meio de uma aproximação de diferença finita de primeira ordem usando os valores das células conectadas à face. Note que o gradiente discreto não se divide pela distância entre os pontos.
- `Div` – operador divergente definido pela *handle function*: `@(x)C'*x`. Integra valores de todas as células para fornecer a divergência.
- `internalConn` – índice lógico para mapear as células internas.
- `N` – Vizinhos para faces internas da malha, representados na seção 3.6 por C_1 e C_2.
- `M` – matriz esparsa responsável por determinar o valor médio correspondente a cada face.
- `faceAvg` – *handle function:* `@(x)M*x`. Para cada interface, calcula o valor médio de uma quantidade definida nas células. Se uma face estiver conectando duas células, a função `faceAvg` calculará a média aritmética dos valores nas duas células. Utilizando a mesma notação apresentada na seção 3.6, matematicamente ele é representado conforme indicado na Eq. (42).

$$avg_{\alpha}\left[f\right]=\frac{1}{2}\left(q\left[C_1\left(f\right)\right]+q\left[C_2\left(f\right)\right]\right) \tag{42}$$

- `faceUpstr` – *handle function:* `@(flag,x)faceUpstr(flag,x,N,[nf,nc])`. Realiza a ponderação dos valores a montante. Também utilizando a mesma notação apresentada na seção 3.6, esse operador é descrito por:

$$upw\left(h\right)\left[f\right]=\begin{cases}h\left[C_1\left(f\right)\right], \text{se grad}\left(p\right)\left[f\right]\text{-g}\,avg_{\alpha}\left(\rho\right)\left[f\right]grad\left(z\right)\left[f\right]>0,\\ h\left[C_2\left(f\right)\right], \text{caso contrário.}\end{cases} \tag{43}$$

- `splitFaceCellValue` – *handle function:*

`@(operators,flag,x)splitFaceCellValue(operators,flag,x,[nf,nc])`.

Em geral, consiste na aplicação de um operador *upwind* pré-existente para obter a ponderação a montante dos valores para quantidades transportadas.

Figura 39 – Definição das propriedades do objeto Model

Classe	PhysicalModel	
Descrição	Classe base para todos os modelos do AD. Implementa um modelo discreto genérico.	
Propriedades	`operators`	Operadores utilizados na construção de sistemas.
	`nonlinearTolerance`	Tolerância para verificação de resíduos. (Padrão 10^{-6})
	`G`	Malha.
	`verbose`	Variável booleana.
	`stepFunctionIsLinear`	Variável booleana.
	`AutoDiffBackend`	**Classe** de diferenciação automática para processos internos. Suporta a inicialização de uma ou mais variáveis AD.

Classe	ThreePhaseBlackOilModel	
Descrição	Trifásico com gás dissolvido opcional e óleo vaporizado	
Propriedades	`Disgas`	*Flag* que decide se o gás pode ser dissolvido na fase oleosa. Padrão: Falso.
	`Vapoil`	*Flag* que decide se o óleo pode ser vaporizado na fase gasosa. Padrão: falso.
	`drsMaxRel`	Incremento máximo de Rs / Rv relativo. Padrão. inf
	`drsMaxAbs`	Incremento máximo de Rs / Rv relativo. Padrão: Inf
OBS: Método construtor	`useCNVConvergence`	Redefine o valor do *flag*. para true

	`model.oil`
	`model.gas`
	`model.water`
		Redefine o valor do *flag* oil, gas, water para true;

Classe	TwoPhaseOilGasModel
Descrição	Sistema de óleo / água bifásico sem dissolução
Propriedades	Somente as herdadas

OBS: Método Construtor	model.oil = true ;
	model.gas = false;
	model.water = true;

Classe	ReservoirModel	
Descrição	Extensão da classe 'Modelo Físico' para acomodar de características específicas do reservatório, como fluido e rocha, bem como fases comumente usadas e variáveis.	
Propriedades	`fluid`	modelo do fluido
	`rock`	Estrutura da rocha (perm/poro/ntg)
	Limites:	
	`dpMaxRel`	Alteração máxima da pressão relativa. Padrão: inf.
	`dpMaxAbs`	Mudança máxima de pressão absoluta. Padrão: inf.
	`dsMaxAbs`	Alteração máxima absoluta da saturação. Padrão: 0.2.
	`maximumPressure`	Pressão máxima permitida no reservatório. Padrão: inf.
	`minimumPressure`	Pressão mínima permitida no reservatório. Padrão: - inf.
	Fases e componentes	
	`water`	Indicador de água mostrando se a fase aquosa / água está presente. Padrão: falso.
	`gas`	Indicador de gás mostrando se a fase de vapor / gás está presente. Padrão: falso.
	`oil`	Indicador de óleo mostrando se a fase líquida / óleo está presente. Padrão: falso.
	Tolerâncias	
	`useCNVConvergence`	Use esquema de tolerância em escala de volume. Padrão: falso.
	`toleranceCNV`	Tolerância CNV (semelhante à norma inf sobre erro de saturação). Padrão: 1e-3;
	`toleranceMB`	Tolerância da soma do erro do balanço de massa. Padrão: 1e-7;
	Input/output	
	`inputdata`	Dado de entrada usado para instanciar o modelo. Padrão: [].
	`extraStateOutput`	Dado de saída extra para states. Padrão: falso.
	`extraWellSolOutput`	Dado de saída extra para wellSols. Padrão: true.
	`outputFluxes`	Armazena fluxos integrados no estado. Padrão: verdadeiro.
	Acoplamento para forças e outros modelos	
	`gravity`	Vetor para força gravitacional. Padrão = gravity();
	`FacilityModel`	Modelo para representar os poços **(classe)**
OBS: método construtor	`operators`	model.operators = setupOperatorsTPFA(G, model.rock, 'deck', model.inputdata);

Fonte: A autora (2020).

3.10.4 Definição do Estado Inicial

A inicialização adotada no MRST está baseada na inicialização realizada no software ECLIPSE, simulador comercial para reservatórios de petróleo. Nele existe uma *keyword* EQUIL que especifica a pressão inicial na profundidade de referência, a profundidade inicial do contato água-óleo e gás-óleo, assim como, a pressão capilar nessas profundidades para cada região de equilíbrio estabelecida utilizando a *keyword* EQLNUM. As regiões dividem o domínio devido ao cálculo das funções das saturações (permeabilidade relativa e pressão capilar) e propriedades PVT (densidade do fluido, FVFs e viscosidade) (SCHIUMBERGER, 2014).

No simulador de diferenciação automática do MRST, para realizar a inicialização, deve-se proceder em dois passos: definição das diferentes regiões e cálculo do equilíbrio vertical dentro de cada região. De acordo com Lie (2016), nas futuras versões pretende-se unificar essa parte do software.

Para a definição das regiões são utilizadas funções existentes no módulo *ad-core:* `getInitializationRegionsBase`, `getInitializationRegionsBlackOil`, `getInitializationRegionsCompositional`, `getInitializationRegionsDeck`. A função `getInitializationRegionsDeck` já extrai as regiões definidas na estrutura deck, criada a partir do ECLIPSE. As funções `getInitializationRegionsBlackOil` e `getInitializationRegionsCompositional` expandem a função `getInitializationRegionsBase` para o caso black-oil e composicional, respectivamente. Em geral, a sintaxe adotada é expressa por:

Quadro 21 – Sintaxe da Inicialização - getInitializationRegionsBlackOil

FUNÇÃO SINTAXE:	
`region =getInitializationRegionsBlackOil(model, contacts,... 'datum_pressure', pRef, 'datum_depth', dRef)`	
	PARÂMETROS DE ENTRADA:
model-	É a instância da classe do modelo. Ex: `TwoPhaseOilWaterModel`.
contacts-	Profundidade do contato entre as fases. No caso do modelo bifásico, é definido um valor que representa a profundidade do contato água-óleo. No caso do modelo trifásico, deverão ser definidos 2 valores, indicando a profundidade do contato óleo-água e gás-óleo.

FUNÇÃO SINTAXE:	
`region =getInitializationRegionsBlackOil(model, contacts,... 'datum_pressure', pRef, 'datum_depth', dRef)`	
pRef -	Pressão inicial na profundidade de referência.
dRef -	Profundidade de referência.
	RETORNO:
region -	Estrutura contendo a profundidade e pressão de referência, a profundidade dos contatos entre as fases e a pressão capilar neles, as saturações máxima e mínima. Além disso, no caso blacck-oil é definido a razão gás-óleo e óleo-gás.
	***datum_pressure–** Pressão de referência;
	***datum_depth–** Profundidade de referência;
	***contacts–** Profundidade dos contatos entre as fases;
	***contacts_pc–** Pressão capilar nos contatos entre as fases. Padrão = 0
	***s_max, s_min–** Saturações máxima e mínima por célula
	***rs, rv–** Razão gás-óleo (rs) e óleo-gás (rv)

Fonte: Adaptado do MRST (2018).

Em seguida, para cálculo do equilíbrio vertical dentro de cada região, utiliza-se a função `initStateBlackOilAD`. Dessa forma, para o modelo bifásico, por exemplo, o reservatório está em equilíbrio com um contato horizontal óleo-água, separando o óleo puro da água pura. O Quadro 22 apresenta a sintaxe da função.

Quadro 22 – Sintaxe da Inicialização - initStateBlackOilAD

FUNÇÃO SINTAXE:	
`state0 = initStateBlackOilAD(model,region)`	
	PARÂMETROS DE ENTRADA:
model–	É a instância da classe do modelo. Ex: `TwoPhaseOilWaterModel`.
region-	É a estrutura gerada pela função `getInitializationRegionsBlackOil`.
	RETORNO:
State0 -	Estrutura os seguintes campos:

FUNÇÃO SINTAXE:	
`state0 = initStateBlackOilAD(model,region)`	
***pressure–**	Pressão inicial para cada célula;
***s–**	Saturação inicial para cada célula. O MRST usa a ordem água, óleo e gás ordem internamente. No caso bifásico, para as saturações no estado inicial, tem-se água na primeira coluna e óleo na segunda.
***rs, rv–**	Razão gás-óleo (rs) e óleo-gás (rv)

Fonte: Adaptado do MRST (2018).

Destaca-se que definir o estado inicial correto é essencial obter estimativas precisas dos fluidos no local e prever como o reservatório se comportará em produção. O Algoritmo 48 define a inicialização para o exemplo do micro reservatório.

Algoritmo 48 – Definição do Estado inicial

```
1.  %Define o estado inicial: pressão de referência na
2.  %profundidade de referência (topo)
3.
4.  p_ref = 100*barsa;
5.  region = getInitializationRegionsBlackOil(model, 20,...
6.  'datum_pressure', p_ref, 'datum_depth', 0);
7.  state0 = initStateBlackOilAD(model, region);
8.
```

3.10.5 Definição dos Poços

A definição dos poços é realizada da mesma forma apresentada na seção 3.5.6. Portanto, considerando um campo de malha 3x3x3, deseja-se estabelecer dois poços, um injetor e outro produtor. O injetor é controlado por BHP de injeção com valor de 150 bar, está localizado no canto inferior esquerdo da malha, com completações nas duas últimas camadas. O produtor também terá o controle do tipo BHP no valor de 50 bar, estando localizado no canto superior direito, com completação nas duas primeiras camadas. A profundidade de referência é admitida no topo. O Algoritmo 49 exemplifica essa função.

Algoritmo 49 – Definição dos Poços

```
1.  % Definição dos Poços
2.  W = struct([]);
3.  W = verticalWell(W, G, rock, 1, 1, 2:3, 'Type', 'bhp',...
4.   'Val', 150*barsa,...
5.   'name', 'I',...
6.   'radius', .1,...
```

Algoritmo 49 – Definição dos Poços

```
    'Comp_i', [1 0],...
    'Sign', 1,...
    'refDepth', 0);

W = verticalWell(W, G, rock, 3, 3, 1:2, 'Type', 'bhp',...
    'Val', 50*barsa,...
    'name', 'P',...
    'radius', .1,...
    'Comp_i', [0 1],...
    'Sign', -1,...
    'refDepth', 0);
```

3.10.6 Definição do *Schedule*

O MRST disponibiliza algumas funções para criar o *schedul*e, apresentando o controle dos poços de acordo com os *timesteps* (passos de tempo) definidos, como a função `initSchedule`, presente no módulo `adjoint` e a função `simpleSchedule` presente no módulo `ad-core`. No presente trabalho, será utilizada a função `simpleSchedule` em virtude do retorno de uma estrutura com campos requeridos pelo simulador adotado. A sintaxe dessa função é definida de acordo com o apresentado no Quadro 23.

Quadro 23 – Sintaxe da Inicialização – simpleSchedule

FUNÇÃO SINTAXE:	
`schedule = simpleSchedule(dt,'W', W, 'src', src, 'bc', bc)`	
	PARÂMETROS DE ENTRADA:
dt–	Vetor (coluna/linha) dos *timesteps* desejados. `(Obrigatório)`
W-	É a estrutura W a ser utilizado no *schedule* definido. `(Opcional)`
src-	Termo fonte/sumidouro a ser utilizado no *schedule*. `(Opcional)`
bc-	Termo fonte a ser utilizado no *schedule*. `(Opcional)`
	RETORNO:
schedule -	Estrutura os seguintes campos: ***step–** campo **val**: constando dos valores dos passos de tempo; campo **control**: indicador de 1 até o número de controles, atribuindo o controle para cada passo de tempo ***control–** Estabelece cada controle. Por exemplo, amazena a estrutura W para o número de controles definidos.

FUNÇÃO SINTAXE:
`schedule = simpleSchedule(dt,'W', W, 'src', src, 'bc', bc)`

Fonte: Adaptado do MRST (2018).

Note que devem ser fornecidos os passos de tempo presentes no *schedule*. Esses podem ser definidos de acordo com a intenção do próprio usuário ou podem ser determinados através de alguma função. A função denominada `rampupTimesteps` possibilita gerar uma sequência de passos de tempo, para um dado tempo total, que crescem geometricamente até alcançar um passo de tempo pré-determinado. O restante intervalo será então subdividido em no valor do passo de tempo pré-determinado (MRST,2020). A sintaxe dessa função é dada conforme o Quadro 24.

Quadro 24 – Sintaxe – rampupTimesteps

FUNÇÃO SINTAXE:		
`dT= rampupTimesteps (time, dt, n)`		
	PARÂMETROS DE ENTRADA:	
time–	Tempo total da simulação	(Obrigatório)
dt-	Passo de tempo de cada intervalo após o crescimento geométrico.	(Obrigatório)
n-	Número de etapas de aceleração. Padrão :8.	(Opcional)
	RETORNO:	
dt -	Vetor com os passos de tempo.	

Fonte: Adaptado do MRST (2018).

Dessa forma, supondo um horizonte de simulação de 600 dias e um intervalo final de 30 dias, estabelecendo apenas um controle ao longo desse intervalo, deve-se proceder conforme indicado no Algoritmo 50 para criação do *schedule*.

Algoritmo 50 – Definição do Schedule

```
% Tempo total de simulação.
T = 600*day;
% Intervalo de tempo após crescimento inicial.
dt = 30*day;
%Define intervalos de tempo menores inicialmente.
dt = rampupTimesteps(T, dt);

%Definição do schedule
schedule = simpleSchedule(dt, 'W', W);
```

Observe que nesse caso é definido apenas um controle com valor constante, ou seja, os poços injetor e produtor possuem o mesmo controle definido inicialmente ao longo de toda simulação. Para estabelecer mais intervalos de controle e variar randomicamente o seu valor, o que será necessário para os testes apresentados na seção 5, procedeu-se conforme indicado no Algoritmo 51. Para maiores detalhes das funções criadas nesse exemplo, deve-se consultar o APÊNDICE C – CÓDIGOS AUXILIARES.

Algoritmo 51 – Definição do Schedule para vários controles e randômicos

```
% Tempo total de simulação.
T = 600*day;
% Intervalo de tempo após crescimento inicial.
dt = 30*day;
%Define intervalos de tempo menores inicialmente.
dt = rampupTimesteps(T, dt);

%Definição do schedule
schedule = simpleSchedule(dt, 'W', W);
%Retorna uma inicialização do schedule para o número de
%controles definidos
ncontrol = 10;
schedule = [];
schedule =controlSchedule(ncontrol,W,schedule,T,dt);

%Retorna o schedule com controle randômico com valores entre a e b
para o poço PRODUTOR.
%rng corresponde ao controlador de números aleatórios.
a = 50*barsa;
b = 80*barsa;
rng=23;
schedule = ...
randSchedule (a,b,W,schedule,ncontrol, rng);

```

3.10.7 Simulador

Finalmente, após a definição do estado inicial, do modelo e do *schedule* é possível realizar a chamada do simulador `simulateScheduleAD`, conforme indica o Algoritmo 52.

Algoritmo 52 – Utilização do simulador simulateScheduleAD

```
% Realizar a simulação
[wellSols, states, report] = simulateScheduleAD(state0,
model, schedule);
```

Essa função pertence ao módulo `ad-core`, e utiliza um *schedule* válido para realizar a simulação de um modelo físico não linear usando diferenciação automática. Essa função depende do modelo, que deve ser derivado da classe `PhysicalModel` , e das classes da *solvers* (não) lineares. O Quadro 25 apresenta a sintaxe dessa função.

Quadro 25 – Sintaxe – simulateScheduleAD

FUNÇÃO SINTAXE:	
`[wellSols, states, report] = simulateScheduleAD(state0, model, schedule,'pn', pv );`	
	PARÂMETROS DE ENTRADA:
state0–	Estado inicial do reservatório .
model-	Modelo físico que determina a jacobiana/convergência do problema. Deve ser uma subclasse de `PhysicalModel`.
schedule	Estrutura possuindo os campos: ***step** (estrutura contendo campos **val** e **control**) ***control** (estrutura estabelecendo os controles)
`'pn',pv`	`'Verbose'` – Indica se uma saída extra deve ser impressa, como relatórios detalhados de convergência e assim por diante. `'OutputMinisteps'` – O *solver* pode não usar *timesteps* iguais aos de etapas de controle, dependendo da rigidez do problema e seleção do timestep. Ativar esta opção fará com que o *solver* gere os estados e relatórios para todas etapas realmente executadas e não apenas no controle. Consulte também `convertReportToSchedule`, que pode será usado para construir uma nova programação a partir desses *timesteps*. `'initialGuess'` – *cell array* com uma entrada por etapa de controle. `'NonLinearSolver'` – instância da classe 'NonLinearSolver', caso se deseje especificar uma seleção do algoritmo.

FUNÇÃO SINTAXE:	
`[wellSols, states, report] = simulateScheduleAD(state0, model, schedule,'pn', pv );`	
	`'OutputHandler'` – permite escrever o estado em disco durante a simulação. Associado a classe ResultHandler usada para recuperar e armazenar resultados de uma simulação. `'WellOutputHandler'` – o mesmo que 'OutputHandler' para as soluções dos poços para etapas individuais. `'ReportHandler'` – o mesmo que 'OutputHandler' para relatórios de etapas individuais. `'LinearSolver'` – instância da classe LinearSolverAD, usada para resolver problemas linearizados na classe NonLinarSolver. `'afterStepFn'` - *Function handle* para uma função opcional que chamada após cada passo de tempo bem sucedido. `'controlLogicFn'` - *Function handle* para a função opcional que será chamada após cada etapa, permitindo que as atualizações do schedule sejam acionadas em eventos especificados.
	RETORNO:
`wellSols` –	Solução nos poços para cada etapa de controle (ou para cada *timestep* de `'OutputMinisteps'` estiver ativado). Dessa forma, para cada etapa do controle, há uma estrutura W.
`states` –	Estado do reservatório para cada etapa de controle (ou para cada *timestep* de `'OutputMinisteps'` estiver ativado). Dessa forma, para cada etapa do controle, há uma estrutura state com campos. **`*pressure`** – pressão em cada célula; **`*rs, rv`** – Razão gás-óleo (rs) e óleo-gás (rv); **`*s`** – saturação em cada célula; **`*flux`** – fluxo em cada face; **`*wellSol`** – solução nos poços
`Report` –	Relatório para a simulação. Contém informações detalhadas para de toda a programação executada e de rotinas chamadas durante da simulação.

FUNÇÃO SINTAXE:	
`[wellSols, states, report] = simulateScheduleAD(state0, model, schedule,'pn', pv );`	
***ControlstepReports** –	*cell array* com informação das rotinas utilizadas durante a simulação, trazendo tempo de simulação, número de iterações, por exemplo. Destaca-se que ele possui o campo *StepReports* onde está o *NonlinearReport* com fornece as informações sobre a linearização da equação.
***ReservoirTime –**	Tempo de simulação definido para cada etapa de controle
***Converged –**	Vetor lógico (verdadeiro ou falso) sobre a convergência a cada passo de tempo
***SimulationTime –**	Tempo da simulação de cada passo de tempo;
***Failure –**	Variável lógica sobre a falha da simulação

Fonte: Adaptado do MRST (2018).

3.10.8 Exemplo Completo

O Algoritmo 53 apresenta o exemplo completo para a simulação de um escoamento bifásico utilizando a diferenciação automática. Os resultados obtidos para esse micro reservatório são apresentados na Figura 40. Para maiores detalhes a respeito da plotagem de resultados e da interface interativa do MRST, consultar o **APÊNDICE B – CRIAÇÃO DAS FIGURAS**.

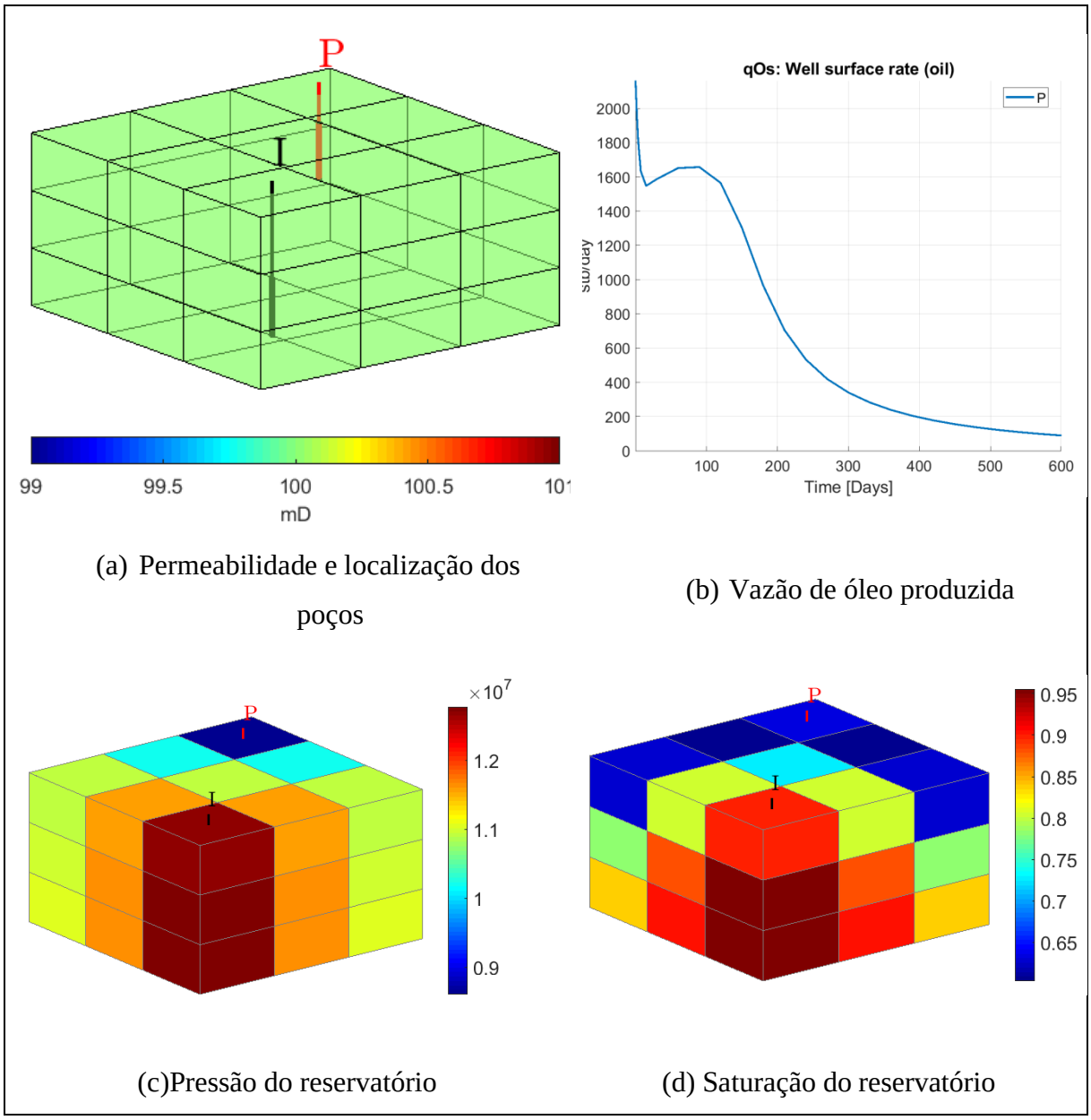
Algoritmo 53 – Exemplo completo da simulação AD

```
close all; clear; clc;
% Ativar módulos utilizados na simulação
mrstModule add ad-core ad-blackoil ad-props mrst-gui
% Definição da geometria
% Seção 3.3 para maiores detalhes
cartDim = [3 3 3];
domain = cartDim.*[50 50 5];
G = cartGrid(cartDim,domain);
G = computeGeometry(G);

%Definição da permeabilidade e porosidade
%Seção 3.4 para maiores detalhes
rock = makeRock(G, 100*milli*darcy, 0.2);

% Definição do fluido:
fluid = initSimpleADIFluid('phases','WO', ...
 'rho',[1000, 850].* kilogram/meter^3, ...
 'n',  [2, 2], ...
 'mu', [1, 1].*centi*poise);

%'phases' - Fases: água e óleo
%'rho' - Densidade
%'n' - Expoente da curva de permeabilidade relativa
%'mu' - Viscosidade

% cria o modelo para o escoamento óleo e água
gravity  on
model = TwoPhaseOilWaterModel(G, rock, fluid);

%Define o estado inicial: pressão de referência na
profundidade de referência (topo)
p_ref = 100*barsa;
region = getInitializationRegionsBlackOil(model, 20,
'datum_pressure', p_ref, 'datum_depth', 0);
state0 = initStateBlackOilAD(model, region);

%Definição dos poços
W = struct([]);
W = verticalWell(W, G, rock, 1, 1, 2:3,...
```

Algoritmo 53 – Exemplo completo da simulação AD

```
43.       'Type', 'bhp',...
44.    'Val', 150*barsa,...
45.    'name', 'I',...
46.    'radius', .1,...
47.    'Comp_i', [1 0],...
48.    'Sign', 1,...
49.    'refDepth', 0);
50.
51.  W = verticalWell(W, G, rock, 3, 3, 1:2,...
52.    'Type', 'bhp',...
53.    'Val', 50*barsa,...
54.    'name', 'P',...
55.    'radius', .1,...
56.    'Comp_i', [0 1] ,...
57.    'Sign', -1,...
58.    'refDepth', 0);
59.
60.  % Tempo total de simulação.
61.  T = 600*day;
62.  % Intervalo de tempo após crescimento inicial.
63.  dt = 30*day;
64.  %Define intervalos de tempo menores inicialmente.
65.  dt = rampupTimesteps(T, dt);
66.  %Definição do schedule
67.  schedule = simpleSchedule(dt, 'W', W);
68.
69.  % Realizar a simulação
70.  [wellSols, states, report] = simulateScheduleAD(state0, model,
71.  schedule);
72.
73.
74.
75.
76.
77.
78.
79.
80.
```

Figura 40 – Resultados simulação teste – micro reservatório



(a) Permeabilidade e localização dos poços

(b) Vazão de óleo produzida

(c)Pressão do reservatório

(d) Saturação do reservatório

Fonte: A autora (2020).

Destaca-se que esse exemplo é meramente ilustrativo, elaborado com fins didáticos para a compreensão da implementação e elementos utilizados pelo MRST. Na seção 5.3 será realizada a aplicação do simulador em modelo mais realístico, bem como, realizada a sua comparação com um simulador comercial.

4 TRAJECTORY PIECEWISE LINEARIZATION - TPWL

Neste capítulo, serão apresentados detalhes a respeito da técnica redução da complexidade numérica, conhecida como Trajectory Piecewise Linearization – TPWL. Essa consiste na linearização da equação do resíduo em torno de estados salvos durante uma simulação previamente executada, também chamada de simulação de treinamento (MACHADO, 2014).

A primeira parte do capítulo apresenta detalhes sobre o método de linearização aplicado à equação de fluxo. Em seguida, apresentam-se em as informações que necessitam ser exportadas do simulador para a implementação do método. Por fim, serão discutidos detalhes a respeito da implementação do método no MRST.

4.1 LINEARIZAÇÃO UTILIZANDO TPWL

Considerando as equações governantes do escoamento bifásico Eq. (11) e Eq. (12), apresentadas na seção 2.1.2, o modelo discreto, dado pela Eq. (20), pode ser reescrito como a soma das três parcelas (MACHADO, 2014):

$$\mathrm{g}(x^{n+1}, x^n, u^{n+1}) = F(x^{n+1}) + A(x^{n+1}, x^n) + Q(x^{n+1}, u^{n+1}) = 0 \qquad (1)$$

Onde x representa o vetor de estados (pressão e saturação) e u indica o vetor de controle (BHPs nos poços). Os índices n e $n+1$ representam os passo de tempo consecutivos da simulação. O termo de fluxo $T^{n+1}x^{n+1}$, designado por $F(x^{n+1})$, representa a massa que flui de um bloco para outro e depende das transmissibilidades das interfaces. O termo de acumulação $-D^{n+1}\left(x^{n+1} - x^n\right)$, denominado por $A\left(x^{n+1}, x^n\right)$, representa a massa acumulada no bloco e depende da compressibilidade dos fluidos. O termo $Q(x^{n+1}, u^{n+1})$ corresponde ao termo fonte/sumidouro representando a massa produzida/injetada em eventuais poços completados nos blocos, dependente do modelo de poço. Por fim, conforme apresentado na seção 2.2, **g** representa o vetor residual, que se busca conduzir à zero, através do Método de Newton.

A ideia chave do *Trajectory Piecewise Linearization - TPWL* é linearizar a equação residual em torno de estados salvos durante uma simulação previamente executada, também conhecida como simulação de treinamento. A cada passo de tempo lineariza-se o modelo em

torno de um ponto de alguma trajetória de treinamento, cujos estados foram previamente convergidos e armazenados (CARDOSO, 2009; HE, 2010; MACHADO, 2014).

Através da Eq. (1) sabe-se que o resíduo **g** é função de x^{n+1}, x^n e u^{n+1}, correspondentes, respectivamente ao vetor de estados do passo atual, vetor de estados do passo anterior e do vetor de controles dos poços.

Para determinado estado atual x^n, denomina-se o estado gravado mais próximo durante a simulação de treinamento como x^i. (HE, 2010). Sendo o vetor x^i, bem como x^{i+1}, vetores de estado convergidos e memorizados durante uma simulação de treinamento para um vetor de controles u^{i+1}. Dessa forma, partindo da ideia chave do TPWL lineariza-se a equação residual (Eq. (1)) em torno dos estados gravados mais próximos (x^{i+1}, x^i e u^{i+1}) utilizando a expansão em série de Taylor do resíduo $\mathbf{g}^{n+1}$. Logo, o resíduo de uma simulação $\mathbf{g}^{n+1}$, calculado no passo de tempo x^{n+1}, pode ser aproximado utilizando dados memorizados pelo truncamento da expansão, conforme representado na Eq. (2).

$$\mathrm{g}^{n+1} = \mathrm{g}^{i+1} + \left(\frac{\partial \mathrm{g}^{i+1}}{\partial x^{i+1}}\right)\left(x^{n+1} - x^{i+1}\right) + \left(\frac{\partial \mathrm{g}^{i+1}}{\partial x^{i}}\right)\left(x^{n} - x^{i}\right) + \left(\frac{\partial \mathrm{g}^{i+1}}{\partial u^{i+1}}\right)\left(u^{n+1} - u^{i+1}\right) \quad (2)$$

Onde $\mathrm{g}^{n+1} = \mathrm{g}\left(x^{n+1}, x^n, u^{n+1}\right)$ e $\mathrm{g}^{i+1} = \mathrm{g}\left(x^{i+1}, x^i, u^{i+1}\right)$. Destaca-se que o sobrescrito *i* representa o passo de tempo da simulação de treinamento onde as derivadas, estados e controles utilizados foram convergidos e memorizados e o sobrescrito *n* representa os passos de tempo de uma nova simulação, considerando o mesmo estado inicial, mas outros controles. (MACHADO, 2014)

O termo $g^{i+1} = g\left(x^{i+1}, x^i, u^{i+1}\right) = 0$, já que o resíduo foi previamente convergido durante a simulação do treinamento. As derivadas apresentadas na Eq. (2) podem ser simplificadas como será apresentado pelas Eq. (3), (4) e (5).

Por definição a matriz jacobiana é a derivada do resíduo em relação ao estado. Desta forma, a primeira derivada parcial que aparece na Eq. (2) pode ser renomeada conforme Eq. (3)
.

$$\frac{\partial g^{i+1}}{\partial x^{i+1}} = J^{i+1} \quad (3)$$

A segunda derivada parcial da Eq. (2) pode ser simplificada escrevendo os termos de g^{i+1} , conforme a Eq. (1), ou seja, separando nos termos referentes à acumulação, fluxo e poços. Desse modo, nota-se que apenas o termo de acumulação $A(x^{i+1},x^{i})$ possui dependência em relação ao passo de tempo anterior x^{i}, conforme indica a Eq. (4).

$$\frac{\partial g^{i+1}}{\partial x^{i}} = \frac{\partial}{\partial x^{i}}\left[A\left(x^{i+1},x^{i}\right)+F\left(x^{i+1}\right)+Q\left(x^{i+1},u^{i+1}\right)\right] = \frac{\partial A^{i+1}}{\partial x^{i}} \tag{4}$$

Procedendo do mesmo modo para a terceira derivada parcial da Eq. (2), conclui-se que apenas o termo de fonte/sumidouro tem relação com o vetor de controles u^{i+1}, conforme indica a Eq. (5).

$$\frac{\partial g^{i+1}}{\partial u^{i+1}} = \frac{\partial}{\partial u^{i+1}}\left[A\left(x^{i+1},x^{i}\right)+F\left(x^{i+1}\right)+Q\left(x^{i+1},u^{i+1}\right)\right] = \frac{\partial Q^{i+1}}{\partial u^{i+1}} \tag{5}$$

Portanto, substituindo as Eq. (5), Eq. (4) e Eq. (3) na Eq. (2) e usando o fato que após a solução $g^{n+1}=0$. A equação resultante do TPWL é expressa por (HE,2010; MACHADO, 2014) :

$$J^{i+1}\left(x^{n+1}-x^{i+1}\right) = -\left[\frac{\partial A^{i+1}}{\partial x^{i}}\left(x^{n}-x^{i}\right)+\frac{\partial Q^{i+1}}{\partial u^{i+1}}\left(u^{n+1}-u^{i+1}\right)\right] \tag{6}$$

A Eq. (6) é uma forma de relacionar o desvio entre os vetores de estado simulado pelo TPWL e gravado na simulação de treinamento ($x^{n+1}-x^{i+1}$) com os desvios entre os estados simulados e gravados para o passo de tempo anterior ($x^{n}-x^{i}$) e os desvios entre os controles aplicados na simulação TPWL e na simulação de treinamento ($u^{n+1}-u^{i+1}$) (MACHADO,2014).

Além disso, conforme indica a Eq. (7) para um estado x^{n} e determinadas as informações da simulação de treinamento, é possível calcular linearmente x^{n+1}, ou seja, sem a necessidade de um processo iterativo de solução (HE,2010; MACHADO,2014).

$$x^{n+1} = x^{i+1} - \left(J^{i+1}\right)^{-1}\left[\frac{\partial A^{i+1}}{\partial x^{i}}\left(x^{n}-x^{i}\right)+\frac{\partial Q^{i+1}}{\partial u^{i+1}}\left(u^{n+1}-u^{i+1}\right)\right] \tag{7}$$

Observe que para a implementação do TPWL, devem ser exportadas as seguintes informações da simulação de treinamento: os vetores de estado no passo de tempo atual (x^{i+1})

e anterior (x^i), a matriz jacobiana (J^{i+1}), a derivada do termo de acumulação em relação ao tempo anterior ($\partial A^{i+1} \big/ \partial x^i$) e a derivada do termo fonte em relação aos controles ($\partial Q^{i+1} \big/ \partial u^{i+1}$).

Note que essas informações, a exceção da derivada $\partial A^{i+1} \big/ \partial x^i$, podem ser obtidas no próprio simulador, uma vez que o termo de acumulação tem sua derivada em relação ao tempo anterior, o que aparentemente não facilmente disponível. Porém ao observar a Eq. (16) nota-se que o termo de acumulação é composto pela diferença de duas parcelas quase idênticas, a primeira tomada no passo de tempo atual $\left(i+1\right)$ e a segunda no passo de tempo anterior $\left(i\right)$. Diante disso, pode-se expressar uma relação entre a derivada que aqui se deseja calcular $\partial A^{i+1} \big/ \partial x^i$ e a derivada do termo de acumulação no passo anterior em relação ao vetor de estados também do passo anterior $\partial A^{i} \big/ \partial x^i$. Para tanto, deve-se fazer uma correção conforme se expressa na Eq. (8), onde Δt^i e Δt^{i+1} representam a duração do passo de tempo anterior e atual, respectivamente (MACHADO, 2014).

$$\frac{\partial A^{i+1}}{\partial x^i} = -\frac{\Delta t^i}{\Delta t^{i+1}} \frac{\partial A^i}{\partial x^i} \tag{8}$$

Por fim, como última questão em aberto a respeito dessa técnica, um ponto importante de se abordar é que, como descrito acima, é necessário que primeiro determine-se o estado gravado x^i "mais próximo" da solução no momento anterior passo, x^n. Existem várias formas de medir essa proximidade (CARDOSO, DURLOFSKY, 2010). No modo adotado, o estado gravado x^i será aquele cujo mapa de saturações z^i for mais próximo de z^n, correspondente ao de saturações no passo de tempo *n* da simulação TPWL. Essa distância é dada pela norma euclidiana. Sendo assim, determina-se o índice *i* do estado gravado a ser utilizado no passo de tempo *n* minimizando-se a distância no espaço das saturações, conforme indica a Eq. (9).

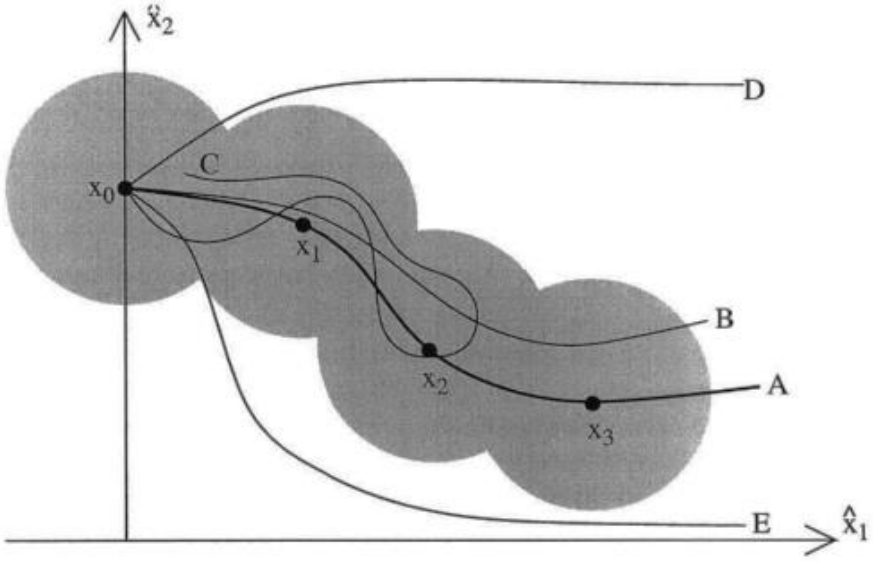
$$i = \arg\min_k \left| z^k - z^n \right| \tag{9}$$

A inclusão dos estados de pressão reduzida nessa determinação não fornece precisão adicional. Isso é razoável porque o problema é linear na pressão, mas não linear na saturação.

Assim, a linearização básica (Eq. (2)) é menos precisa para saturação, o que torna as soluções TPWL mais sensíveis à distância do estado de saturação do que à distância do estado de pressão (CARDOSO, DURLOFSKY, 2010).

Esta linearização é melhor quanto maior a proximidade do estado simulado ao estado gravado. Além disso, é dependente da trajetória, isto é, dos estados pelos quais passou anteriormente (MACHADO, 2014).

Figura 41– Geração de modelos linearizados



Fonte: REWIENSK (2003); REWIENSKI M. e WHITE (2003).

Como ilustrado na Figura 41, o procedimento proposto acima permite "cobrir os modelos" apenas na parte do espaço de estados localizada ao longo da trajetória de treinamento (curva A). Assumindo que o modelo de ordem reduzida seja composto por modelos lineares gerados ao longo dessa trajetória. Se a trajetória de um determinado sistema, correspondente a um dado controle u^{n+1}, estiver dentro da região do espaço de estados coberto por esses modelos (ou, em outras palavras, permanece suficientemente próximo de toda linearização pontos), espera-se que o modelo linear por partes construído aproxime adequadamente comportamento do sistema não linear original (curvas B e C). Cabe salientar que, pode-se aplicar um modelo TPWL para controles significativamente diferentes dos controles de treinamento, desde que trajetórias correspondentes fiquem na região do espaço de estados coberta pelos modelos linearizados (curva C). Uma situação diferente ocorre quando o dado de entrada faz com que a trajetória saia da região coberta pelos modelos linearizados (curvas D e E). Então, o modelo

TPWL provavelmente não proporcionará uma boa aproximação ao sistema não linear (REWIENSK, 2003; REWIENSKI M. e WHITE, 2003).

Em resumo, a metodologia geral, primeiramente, requer cálculos de pré-processamento nos quais o modelo TPWL é construído. E em seguida, o processamento onde é realizada a simulação TPWL, obtendo resultados para cada nova simulação (teste) executada. Esses dois componentes são descritos abaixo. A Figura 42 fornece um fluxograma para os processos (HE, 2013).

- **Pré-processamento**

Sendo T o tempo total de simulação, N_t o número de passos de tempo da simulação e N_c o número de ciclos de controle ao longo do determinando da simulação. O procedimento que define o pré-processamento é descrito por:

1. Determina-se o vetor de controles $\left\{u^i\right\}_{i=1}^{N_T}$ a partir da lista dos controles (pressão do fundo de poço) $\left\{BHP^t\right\}_{t=1}^{N_c}$ a serem simulados.
2. Executar o simulador para a simulação de treinamento com controles $\left\{u^i\right\}_{i=1}^{N_T}$ definidos pelo *schedule*.
3. Salvar os estados $\left\{x^i\right\}_{i=1}^{N_t}$, obtidos como resposta da simulação, para cada passo de tempo, armazenando também os respectivos intervalos $\left\{\Delta t^i\right\}_{i=1}^{Nt}$.
4. Salvar as derivadas convergidas em todos os passos de tempo da simulação $\left\{J^{i+1}\right\}_{i=1}^{Nt}$, $\left\{\frac{\partial A^{i+1}}{\partial x^i}\right\}_{i=1}^{Nt}$, $\left\{\frac{\partial Q^{i+1}}{\partial u^{i+1}}\right\}_{i=1}^{Nt}$.

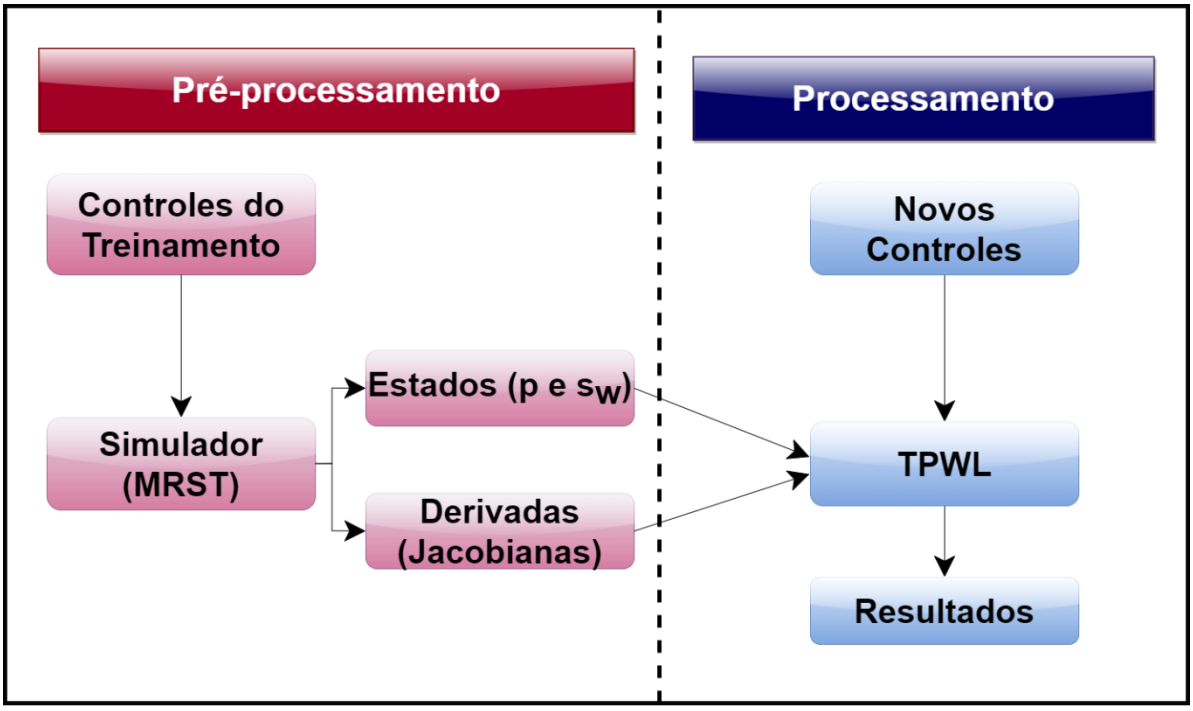
- **Processamento**

O algoritmo executado na simulação TPWL consiste nos seguintes passos (MACHADO, 2014):

1. Faz-se $n = i = 1$, $t^n = 0$ e $x^n = x^i$.

2. Determina-se o vetor de controles do passo de tempo de u^{n+1} a partir da lista de controles a serem simulados $\left\{BHP^{t}\right\}_{t=1}^{N_c}$.
3. Busca-se entre todos os estados gravados $\left\{x^{i}\right\}_{i=1}^{N_t}$ qual é mais próximo de x^{n} considerando somente as saturações. Determina-se i e selecionam-se os estados gravados x^{i} e x^{i+1}, e o intervalo de tempo entre os passos i e $i+1$, Δt^{i}
4. Selecionam-se também as derivadas J^{i+1}, $\left.\frac{\partial A}{\partial x^{i}}\right|^{i+1}$ e $\left.\frac{\partial Q}{\partial u^{i+1}}\right|^{i+1}$.
5. Resolve-se a Eq. (7) com os dados da simulação de treinamento e o estado anterior, obtendo x^{n+1}.
6. Determina-se o tempo transcorrido até então na simulação por $t^{n+1}=t^{n}+\Delta t^{i}$. Além disso, atualiza-se o vetor de estados, $x^{n} \leftarrow x^{n+1}$
7. Calculam-se as vazões nos poços aplicando-se as pressões das células dos poços em x^{n+1} ao modelo de poço.
8. Se $t^{n+1} < T$ retorna-se ao passo 2, caso contrário termina a simulação.

Figura 42 –Metodologia do TPWL



Fonte: Adaptado de HE (2013).

4.2 EXPORTAÇÃO DAS INFORMAÇÕES PARA O TPWL NO MRST

Diante da metodologia apresentada anteriormente (seção 4.1), para realizar a simulação TPWL, se faz necessário armazenar e exportar as seguintes informações ao final da simulação de treinamento após a convergência do método de Newton:

- Incrementos de tempo da simulação $\left\{\Delta t^i\right\}_{i=1}^{Nt}$;
- Controles utilizados $\left\{u^i\right\}_{i=1}^{N_T}$;
- Os estados convergidos $\left\{x^i\right\}_{i=1}^{N_t}$;
- As derivadas convergidas $\left\{J^{i+1}\right\}_{i=1}^{Nt}$, $\left\{\partial A^{i+1}/\partial x^i\right\}_{i=1}^{Nt}$, $\left\{\partial Q^{i+1}/\partial u^{i+1}\right\}_{i=1}^{Nt}$.

Essas informações são extraídas do simulador. Para ter acesso a todas as variáveis requeridas, foi necessário do profundo conhecimento do código para que fosse possível exportar todos os dados requeridos para a simulação de treinamento.

Conforme apresentado na seção 3.10, a simulação é realizada através do comando `simulateScheduleAD` (Algoritmo 52).

Ao realizar essa simulação, os mapas de pressão e saturação convergidos pelo Newton-Raphson a cada passo de tempo são fornecidos na estrutura `states` (especificamente nos campos `states.pressure` e a primeira coluna de `states.s`, conforme detalhado no Quadro 25), necessitando apenas concatenação desses valores em $\left\{x^i\right\}_{i=1}^{N_t}$.

Os incrementos de tempo $\left\{\Delta t^i\right\}_{i=1}^{Nt}$ têm obtenção direta na estrutura `schedule.step.val`. (ver Quadro 23)

Os controles utilizados $\left\{u^i\right\}_{i=1}^{N_T}$ também são diretamente obtidos na variável `wellSols`. Note que, no presente trabalho, será analisado apenas a variável de controle como BHP, dessa forma, os valores de interesse são encontrados em `wellSols.bhp`. No entanto, no caso de expansão do código para controle por vazão, destaca-se que o valor da vazão de óleo e vazão de água também são fornecidos em `wellSols`.

Em se tratando das derivadas, foi necessário conhecer detalhes internos do simulador para ter acesso a elas, e em seguida, exportar esses dados. Esses detalhes são apresentados na seção seguinte.

4.2.1 Exportação das derivadas para o TPWL utilizando MRST

A seguir é detalhado o funcionamento interno do simulador `simulateScheduleAD`, destacando-se a função *equationsOilWater* e o procedimento da exportação das derivadas.

4.2.1.1 Simulador – Processo Interno

O MRST AD-OO utiliza como simulador a função `simulateScheduleAD`. Essa função requer, além do estado inicial e *schedule*, um modelo derivado da classe `Physical Model`(ver Figura 38), como dado de entrada. Esses dados passarão por um processo interno, até que se obtenha como resposta os estados do reservatório e as vazões nos poços.

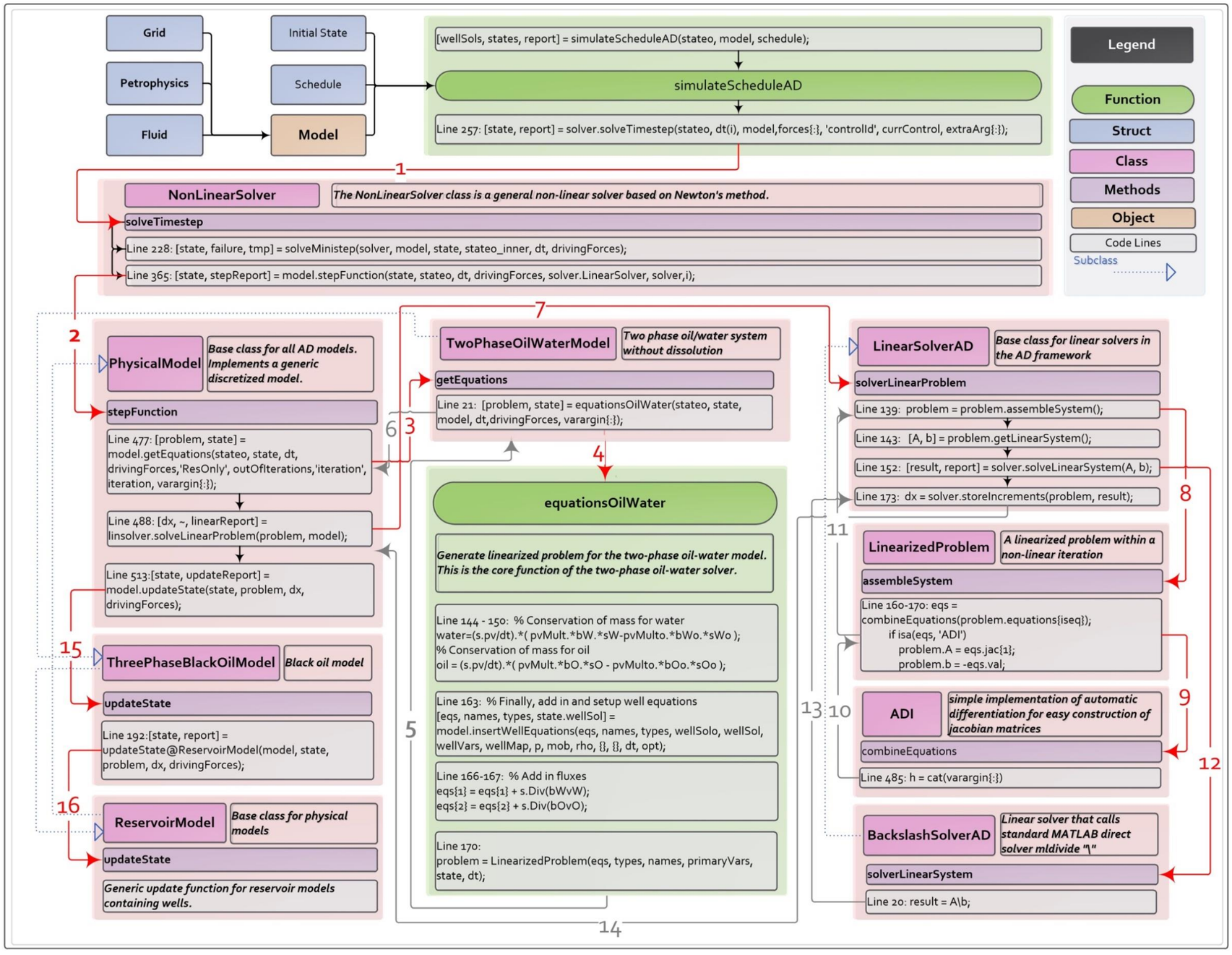
Internamente, o *solver* utilizado nessa função está implementado na classe `NonLinearSolver`, correspondente a classe base do *solver* não linear baseado no Método de Newton. Conforme Lie (2016), ela é capaz ajustar os passos de tempo em *substeps* para garantir cálculos estáveis e convergência adequada. Cada *substep* executa uma ou mais iterações não lineares.

Dentro da iteração não linear, o modelo físico calcula equações residuais. Quando as equações residuais são calculadas, a matriz jacobiana correspondente a cada equação residual linearizada também é computada implicitamente pela biblioteca AD. Isso produz uma coleção de blocos de matrizes que representam a derivada de uma equação residual específica em relação a uma variável primária. O algoritmo de linearização reúne os blocos da matriz em uma matriz jacobiana geral para todo o sistema e, se necessário, elimina certas variáveis para produzir um sistema linear reduzido. O *solver* linear configura pré-condicionadores apropriados e resolve o sistema linear definindo os incrementos de iteração, enquanto o *solver* não linear executa a estabilização necessária.

Finalmente, o modelo físico atualiza os estados do reservatório com incrementos calculados e recalcula os resíduos a serem verificados em relação às tolerâncias prescritas pelo *solver* não linear. Dessa forma, é finalizada uma iteração não-linear e o procedimento é repetido

até obter os estados convergidos. Todo esse processo é realizado por meio das classes do MRST AD-OO e seus métodos, conforme indica o fluxograma e a descrição seguinte.

Figura 43 – Resumo do processo interno – Simulador AD



Fonte: A autora (2020).

De acordo com o fluxograma apresentado, primeiramente deve-se gerar o modelo de reservatório, objeto de uma das classes derivadas de `PhysicalModel` (ver Figura 38), juntamente com o *schedule* e com o estado inicial. Esses parâmetros são passados como dados de entrada para a função `simulateScheduleAD`.

Após a etapa de definições de parâmetros e validações das estruturas de entrada, é executado o método `solveTimestep` presente na classe `NonLinearSolver` (Seta 1).

Esse método destina-se a resolver um passo de tempo para um sistema não linear usando um ou mais *substeps*, através do método `solveMinistep`. O método `solveMinistep` chama a função `stepFunction`, pertencente a classe `PhysicalModel`, para executar uma única etapa não linear. (Seta 2)

Um dos primeiros pontos a se destacar no método `stepFunction` é que ela faz referência ao método `getEquations` da classe referente ao modelo escolhido, no caso em questão a classe é `TwoPhaseOilWaterModel` (Seta 3). Esse método é responsável por chamar a função `equationsOilWater`. Essa é a principal função para solução do escoamento bifásico. Ela reúne as equações residuais para a conservação da água e do óleo e, através da diferenciação automática, a derivada de cada equação residual com respeito às variáveis primárias é fornecida. Perceba que esta é a função de maior interesse para a exportação das derivadas. Em virtude disso, será abordada separadamente, com maiores detalhes, na seção posterior.

Com as informações das equações residuais, destina-se ao método `solveLinearProblem`, pertencente a classe `LinearSolverAD`, classe base para *solvers* lineares, com a implementação de métodos para solução de problemas linearizados (Seta 7).

O primeiro método referenciado é denominado por `assembleSystem`, pertencente a classe `LinearizedProblem` (Seta 8). É a partir desse método que se realiza a montagem de um sistema linear completo, utilizando as informações dos blocos de matrizes jacobianas armazenadas. O método `assembleSystem` utiliza o método `combineEquations` da classe ADI para combinar (concatenar) um conjunto de equações do tipo ADI (Seta 9). Dessa forma, a equação resultante terá o único jacobiano esparso montado, contendo todas as derivadas. É então, no objeto denominado `problem`, que é armazenada a matriz jacobiana completa `A (eqs.jac)` e o vetor residual `b (eqs.val)` (Seta 10).

Portanto, após a verificar a necessidade de redução, reorganizar e aplicar o fator de escala, o sistema é resolvido pelo método `solveLinearSystem` da classe `BackslashSolverAD`,

subclasse de `LinearSolverAD` (Seta 12). Note que esse método utiliza o solver padrão do MATLAB, `mldivide` ("\"'). Em seguida, são determinados os incrementos da iteração não linear (Seta 13).

Após executar a estabilização necessária do incremento de Newton, por fim, o estado é atualizado pelo método `updateState` da classe do modelo físico `ThreePhaseBlackOilModel` que, na verdade, referencia o método `updateState` presente em `ReservoirModel`, a qual também utiliza métodos implementados na superclasse do modelo `PhysicalModel` (seta 15 e 16). Note que `ThreePhaseBlackOilModel` é uma subclasse de `ReservoirModel`, que é uma subclasse de `PhysicalModel`.

Por fim, o modelo físico verifica a convergência e cria o `report` da iteração não linear e o prossegue-se da mesma forma até que se obtenham os estados convergidos em todos os passos de tempo. Cabe destacar que esses são apenas alguns pontos-chaves do funcionamento do simulador, que possuem relevância para a problemática em questão. Para ter acesso a maiores detalhes, devem-se utilizar os códigos presentes no MRST.

Os *solvers* `NonLinearSolver`, `LinearSolverAD`, `LinearizedProblem` e `BackslashSolverAD` pertencem ao módulo *ad-core* do MRST. Os dois primeiros são implementados como uma subclasse da classe denominada por *handle*. Isso significa que o objeto de classe é passado por referência a funções, para que quaisquer alterações que ocorram dentro da função também tenham efeito fora da função sem precisar retornar explicitamente o objeto de classe como argumento de saída. Isso é feito visto que esses *solvers* fazem muita conta interna dentro do simulador e, portanto, precisam ser passadas como argumento para muitas funções.

4.2.1.2 Função *equationsOilWater*

A função *equationsOilWater* pertence ao módulo *ad-blackoil* do MRST. Conforme mencionado, ela é o ponto chave para a solução do escoamento bifásico. Ela reúne a equação residual da fase água e óleo (Eq. (11) e Eq. (12) da seção 2.1.2). Essas equações são apresentadas com os operadores discretos (definidos na seção 3.6). As matrizes jacobianas constando das derivadas com relação às variáveis primárias são obtidas pela diferenciação automática. Será apresentado no Algoritmo 54, o código dessa função, sobre o qual serão ressaltados alguns pontos principais.

Primeiramente, destaca-se que nas linhas 25 e 26, são inicializadas as variáveis primárias pelo comando `initVariablesAD`, ou seja, pertencentes a classe ADI. Essas variáveis são pressão, saturação da água, e variáveis de poço (qWs, qOs, e BHP). Note que como são declaradas em conjunto, quaisquer equações constituídas dessas variáveis também serão objetos classe ADI, constituídos pelas propriedades `(val e jac)`. Cabe destacar que `jac` conterá 5 campos, cada um correspondente a derivada da função com relação a essas 5 variáveis.

Em seguida, nas linhas 37 a 58, são definidas propriedades (permeabilidade relativa, viscosidade, FVF) para o óleo e água, ressaltanto que todas as propriedades também são objetos da classe ADI.

A partir da linha 69, definem-se as equações do escoamento bifásico. Inicialmente, são definidas as equações de conservação de massa da água e do óleo (linha 76 a 80). Note que a equação é escrita na forma discreta que muito se assemelha com a formulação matemática. Observe que objetos "*water* "e "*oil* estão, até então, armazenando apenas o termo de acumulação. Com a diferenciação automática *"water"* e *"oil"* possuem propriedades `val` e `jac`. Portanto, a propriedade `jac`, nesse ponto, possui as derivadas apenas do termo de acumulação com relação as 5 variáveis definidas. Observe que essas derivadas são de interesse para exportação e implantação da técnica do TPWL. Em seguida, os objetos *"water"* e *"oil"* são armazenados em um *cell array* denominado *"eqs"*, onde *"eqs{1}"* e *"eqs{2}"* serão referentes à equação da água e do óleo, respectivamente.

O próximo passo consiste em incorporar os termos de fonte sumidouro e/ou poços. Admitindo o caso de poços, isso será realizado pelo comando apresentado nas linhas 97 e 100. Esse comando direciona para a classe `FacilityModel`, uma subclasse de `PhysicalModel` responsável por acoplacar os poços ao modelo. Após esse comando "*eqs{1}"* possui a equação da água com termo de acumulação e termos referentes aos poços; e *"eqs{2}"* possui a equação do óleo com termo de acumulação e termos referentes aos poços. Além de incorporar três novas equações, referentes à vazão de água, óleo e BHP nos poços.

Para obter somente as derivadas dos termos referentes aos poços, conforme necessita o TPWL, basta apenas realizar a subtração de *eqs{1}"* e *eqs{2}"* após a incorporação dos poços, com o termo de acumulação, anteriormente obtidos. Por fim para obter a equação completa da água e do óleo (conforme formulação das Eq. (11) e Eq. (12) da seção 2.1.2), basta incluir o termo de fluxo, conforme apresentado nas linhas 103 e 104.

Algoritmo 54 – Código do equationsOilWater

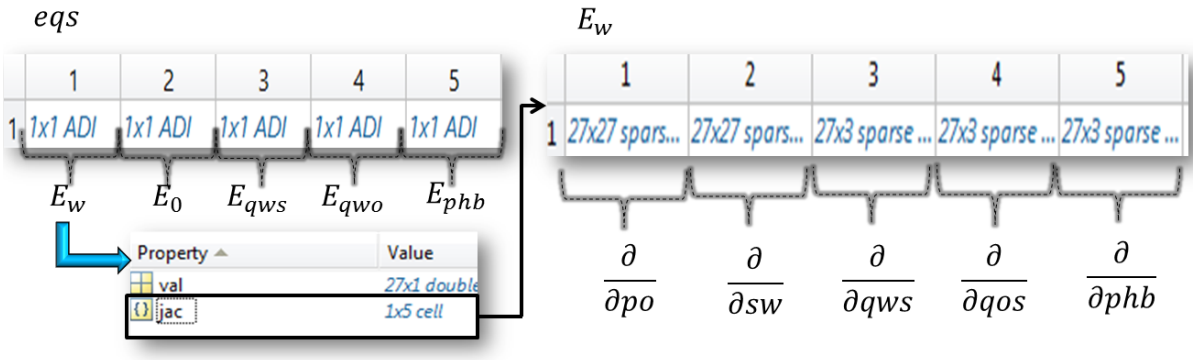
```
1.  [problem,  state]  =  equationsOilWater(state0,  state,  model,
2.  dt,drivingForces, varargin)
3.  %Generate linearized problem for the two-phase oil-water model
4.  opt = struct('Verbose', mrstVerbose, ...
5.   'reverseMode', false,...
6.   'resOnly', false,...
7.   'iteration', -1);
8.  opt = merge_options(opt, varargin{:});
9.  W = drivingForces.W;
10. s = model.operators;
11. % Properties at current timestep
12. [p, sW, wellSol] = model.getProps(state, 'pressure', 'water', ...
13. 'wellsol');
14. % Properties at previous timestep
15. [p0, sW0, wellSol0] = model.getProps(state0, 'pressure', ...
16. 'water', 'wellSol');
17. [wellVars, wellVarNames, wellMap] = ...
18. model.FacilityModel.getAllPrimaryVariables(wellSol);
19.
20. % Initialize independent variables.
21. if ~opt.resOnly
22.  % ADI variables needed since we are not only computing residuals.
23.  if ~opt.reverseMode
24.    [p, sW, wellVars{:}] =  ...
25. model.AutoDiffBackend.initVariablesAD(p, sW, wellVars{:});
26.  else
27.    wellVars0 =  ...
28. model.FacilityModel.getAllPrimaryVariables(wellSol0);
29.    [p0, sW0, wellVars0{:}] =  ...
30. model.AutoDiffBackend.initVariablesAD(p0, sW0, wellVars0{:});
31.  end
32. end
33. primaryVars = {'pressure', 'sW', wellVarNames{:}};
34. % We will solve for pressure, water saturation (oil saturation
35. %follows via the definition of saturations) and well rates + bhp.
36.
37. % Evaluate relative permeability
38. sO  = 1 - sW;
39. sO0 = 1 - sW0;
40. [krW, krO] = model.evaluateRelPerm({sW, sO});
41. % Multipliers for properties
42. [pvMult, transMult, mobMult, pvMult0] =  ...
43. getMultipliers(model.fluid, p, p0);
44. % Modifiy relperm by mobility multiplier (if any)
45. krW = mobMult.*krW; krO = mobMult.*krO;
46. % Compute transmissibility
47. T = s.T.*transMult;
48. % Gravity contribution
49. gdz = model.getGravityGradient();
50. % Evaluate water properties
51. [vW, bW, mobW, rhoW, pW, upcw] = ...
52. getFluxAndPropsWater_BO(model,p, sW, krW, T, gdz);
53. bW0 = model.fluid.bW(p0);
54.  % Evaluate oil properties
55. [vO, bO, mobO, rhoO, p, upco] = getFluxAndPropsOil_BO(model,...
56. p, sO, krO, T, gdz);
57. bO0 = getbO_BO(model, p0);
58.
```

Algoritmo 54 – Código do equationsOilWater

```
59.    if model.outputFluxes
60.     state = model.storeFluxes(state, vW, vO, []);
61.    end
62.    if model.extraStateOutput
63.     state = model.storebfactors(state, bW, bO, []);
64.     state = model.storeMobilities(state, mobW, mobO, []);
65.     state = model.storeUpstreamIndices(state, upcw, upco, []);
66.    end
67.
68.    % EQUATIONS ---------------------------------------------------------
69.    % Upstream weight b factors and multiply by interface fluxes to
       obtain the
70.    % fluxes at standard conditions.
71.    bOvO = s.faceUpstr(upco, bO).*vO;
72.    bWvW = s.faceUpstr(upcw, bW).*vW;
73.
74.    % Conservation of mass for water
75.    water = (s.pv/dt).*( pvMult.*bW.*sW - pvMult0.*bW0.*sW0 );
76.
77.    % Conservation of mass for oil
78.    oil = (s.pv/dt).*( pvMult.*bO.*sO - pvMult0.*bO0.*sO0 );
79.
80.    eqs = {water, oil};
81.    names = {'water', 'oil'};
82.    types = {'cell', 'cell'};
83.
84.    % Add in any fluxes / source terms prescribed as boundary
       conditions.
85.    rho = {rhoW, rhoO};
86.    mob = {mobW, mobO};
87.    sat = {sW, sO};
88.
89.    [eqs, state] = addBoundaryConditionsAndSources(model, eqs, ...
90.                                   names, types, state, ... {pW, p},
91.                                   sat, mob, rho, ...{}, {}, ...
92.                                   drivingForces);
93.    % Finally, add in and setup well equations
94.    [eqs, names, types, state.wellSol] =
95.    ...model.insertWellEquations(eqs, names, types, wellSol0, ...
96.    wellSol, wellVars, wellMap, p, mob, rho, {}, {}, dt, opt);
97.
98.    % Add in fluxes
99.    eqs{1} = eqs{1} + s.Div(bWvW);
100.   eqs{2} = eqs{2} + s.Div(bOvO);
101.
102.   problem = LinearizedProblem(eqs, types, names, primaryVars, ...
103.   state, dt);
104.   end
105.
106.
107.
108.
109.
```

Fonte: MRST (2018).

Figura 44 – Objeto da classe ADI referente à equação residual da água



*Ew – equação da água; Eo – equação do óleo; qws – vazão da água; qos – vazão de óleo; bhp – pressão fundo de poço; po – pressão óleo; sw – saturação da água.

Fonte: A autora (2020).

A Figura 44 traz um exemplo de como se apresenta o *cell array* "*eqs"* ao final da função *equationOilWater* para uma malha de dimensão [3x3x3] com um poço injetor completado nas duas camadas inferiores e o produtor nas duas superiores, tal como descrito na seção 3.10.

Note que ela armazena objetos da classe ADI, cada um representando uma equação residual. Por exemplo, a célula "*eqs{1}"* trata da equação residual da completa da água. Por pertencer à classe ADI, possui as propriedades `val` e `jac`. Note que no campo `jac`, estão presentes as derivadas com relação às variáveis primárias definidas (pressão, saturação da água, e variáveis de poço – qWs, qOs, e BHP). O mesmo formato se repete com as demais células do *cell array* "*eqs"*.

Finalizada a montagem da equação, o *cell array* "*eqs"* é armazenado no objeto *problem*, instância da classe *LinearizedProblem*. As propriedades da classe *LinearizedProblem* estão apresentadas no Quadro 24.

Quadro 26 – Classe LinearizedProblem

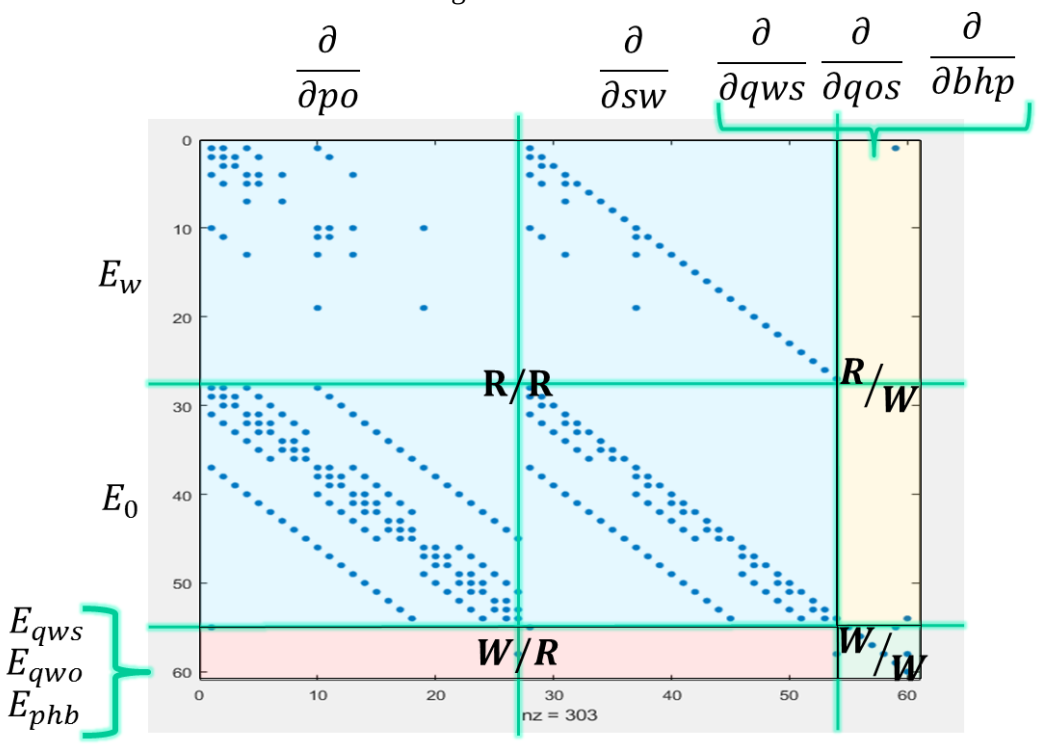
CLASSE – classdef LinearizedProblem		
	PROPRIEDADES	
	equations–	*Cell array* contendo objetos da classe ADI .
	types–	*Cell array* de comprimento igual ao número de equações indicando seus tipos. Ex: *(cell, cell, perf, perf, well)*
	equationNames–	*Cell array* de comprimento igual ao número de equações indicando nomes exclusivos. Ex: *(water, oil, waterWells, oilWells, closureWells)*
	primaryVariables–	*Cell array* com os nomes das varáveis primárias. Ex: *(pressure, sW, qWs, qOs, bhp)*

CLASSE – classdef LinearizedProblem	
A –	Sistema linear após a montagem de jacobianos. Ficará vazio até que a função de membro assembleSystem seja chamada.
b –	Lado direito para sistema linearizado.ver A.
state –	O estado do problema usado para produzir as equações.
dt –	A duração do intervalo de tempo.
iterationNo –	Número de iteração não linear correspondente ao problema.
drivingForces –	A estrutura de forças motrizes usada para gerar as equações. Ex: W.

Fonte: Adaptado de MRST (2018).

Note que é na instância da classe *LinearizedProblem* que o problema está definido. Observe também que ela contém todas as informações de interesse para a implementação do TPWL. É importante mencionar que a jacobiana completa só estará disponível após a chamada do método *assembleSystem* apresentado descrito no 4.2.1.1. Esse método concatena as equações, gerando a matriz jacobiana completa a partir dos sub-blocos, conforme indica a Figura 45.

Figura 45 – Matriz Jacobiana



Fonte: A autora (2020).

Essa matriz foi gerada para malha de 27 elementos, dimensão [3x3x3], com dois poços. Dessa forma, a parte correspondente a Reservatório/Reservatório (indicada por R/R) é uma matriz esparsa com dimensão 54x54, visto que corresponde às derivadas da equação da água (27x27) e do óleo (27x27) com relação às varáveis do campo, pressão e saturação.

As equações dos poços, bem como as derivadas de suas variáveis, apresentam uma linha ou coluna para cada poço. Dessa forma, sendo 2 poços e 3 equações, a parte indicada por Poço/Reservatório (W/R) é uma matriz esparsa de dimensão 6x54. Pelo mesmo raciocínio, a parte correspondente a Reservatório/Poço (R/W) tem dimensão de 54x6 e a Poço/Poço (W/W) tem dimensão de 6x6.

Dessa forma, a matriz A para a solução do problema linearizado tem dimensão de 60 x60 nesse exemplo. Cabe acrescentar que matriz jacobiana J utilizada no método do TPWL é a parte correspondente ao campo (Reservatório/Reservatório). Portanto, para o exemplo mencionado a dimensão de J seria de 54x54.

4.2.1.3 Exportação das derivadas

Diante do apresentado na seção anterior, é possível perceber que as derivadas J^{i+1}, $\partial A^{i+1} \big/ \partial x^{i+1}$, $\partial Q^{i+1} \big/ \partial u^{i+1}$, são calculadas por diferenciação automática na função *equationsOilWater.*

As equações residuais e os respectivos blocos das matrizes jacobianas são armazenados na instância da classe *LinearizedProblem*. Porém foi verificado, que essa instância de classe, bem como, a matriz jacobiana completa não estão definidos para *output do* simulador, possivelmente baseados nos princípios de encapsulamento e abstração, em que dados internos são protegidos e somente os resultados são apresentados ao usuário.

Diante disso, para a exportação das matrizes requeridas pelo TPWL, foram desenvolvidas duas metodologias, apresentadas a seguir. A primeira envolve alteração do código do MRST, enquanto a segunda consiste no recálculo das matrizes jacobianas a partir das funções disponíveis no MRST e dos *outputs* da simulação.

- **Metodologia 1**

Inicialmente, propôs-se acrescentar o objeto `problem`, ao relatório de saída da função `simulateScheduleAD`, correspondente a estrutura `report`. Para essa alteração, no método *stepFunction* da classe *PhysicalModel*, no comando `report = model.makeStepReport` foi acrescentado o campo `problem`, para exportar `problem.assembleSystem`. Cabe destacar que a função `assembleSystem` foi novamente chamada, pois antes de chegar ao final da função, onde o report é criado, a informação da matriz jacobiana completa é apagada. Além disso, ainda na classe *PhysicalMode*l, é necessário alterar o método *makeStepReport*, onde é padronizado o modelo do relatório da função-membro *stepFunction*, incorporando o campo `problem`. Essa alteração está indicada na Figura 46.

Figura 46 – Alteração para exportar objeto problem

PhysicalModel

methods

stepFunction

```
function [state, report] = stepFunction(model, state,
state0, dt, drivingForces, linsolver, nonlinsolver,
iteration, varargin)
report = model.makeStepReport(...
                        'LinearSolver', linearReport, ...
                        'UpdateState',  updateReport, ...
                        'Failure',      failure, ...
                        'FailureMsg',   failureMsg, ...
                        'Converged',    isConverged, ...
                        'AssemblyTime', t_assembly, ...
                        'Residuals',    values, ...
                        'StabilizeReport', stabilizeReport,...
                        'ResidualsConverged', convergence,...
Added: →                'problem', problem.assembleSystem())
```

makeStepReport

```
function report = makeStepReport(model, varargin)
    report = struct('LinearSolver', [], ...
                    'UpdateState',  [], ...
                    'Failure',      false, ...
                    'FailureMsg',   '', ...
                    'Converged',    false, ...
                    'FinalUpdate',  [],...
                    'Residuals',    [],...
                    'AssemblyTime',    0, ...
                    'StabilizeReport', [], ...
                    'ResidualsConverged', [],...
                    'problem', []);
```

Added:

Fonte: A autora (2020).

Com isso, pode-se ter acesso a matriz jacobiana completa e as demais propriedades da instância da classe *LinearizedProblem*, conforme indicado no Quadro 26.

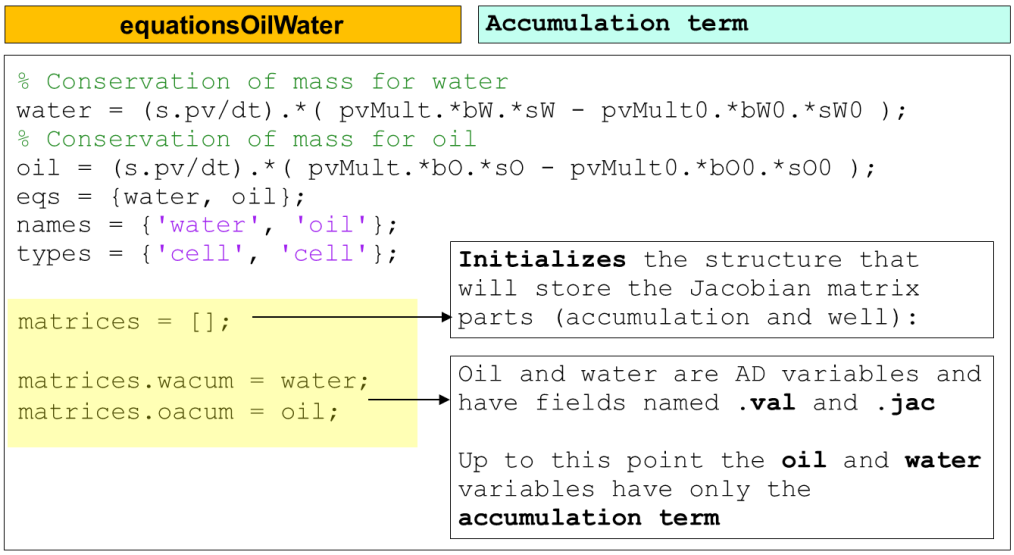
Além disso, para a implementação do TPWL, tem-se interesse na derivada do termo de acumulação ($\partial A^{i+1}/\partial x^{i+1}$) e dos poços ($\partial Q^{i+1}/\partial u^{i+1}$), uma vez que eles são calculados na função *equationsOilWater*, proprõe-se a criação de uma estrutura nessa função para armazenar essas variáveis de interesse.

Essa estrutura foi denominada por `matrices`. Inicialmente ela armazena o termo referente à acumulação, resultando nos objetos `matrices.wacum` e `matrices.oacum`, conforme indica a Figura 47. O campo `jac` desses objetos, quando contacatenados, resultará na derivada ($\partial A^{i+1}/\partial x^{i+1}$). Observe que essa derivada é obtida em relação aos estados do tempo atual. Para obtê-la em relação ao tempo anterior ($\partial A^{i+1}/\partial x^{i}$), conforme requerido pelo TPWL, deve-se empregar a relação expressa pela Eq. (8).

Em seguida, são armazenados os termos referentes aos poços, conforme apresenta a Figura 48, resultando nos objetos `matrices.wellW` e `matrices.wellO`, com apenas as contribuições dos poços. Como a derivada de interesse é em relação aos controles $\partial Q^{i+1}/\partial u^{i+1}$, a parte da matriz jacobiana necessária é Reservatório/Poço (ver Figura 45), especificamente as colunas referentes ao BHP, visto que nesse trabalho foi considerado apenas o controle por BHP.

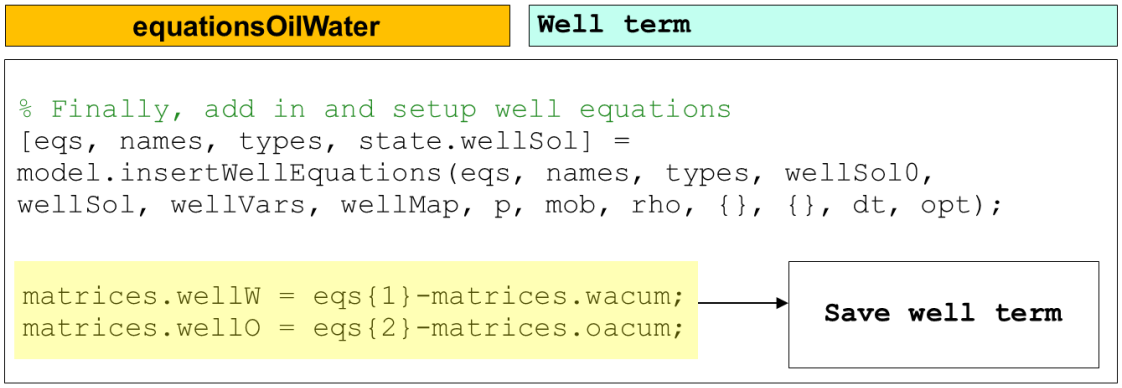
Por fim, para que essa estrutura também esteja presente no `report`, ela deve ser definida como uma propriedade da classe `LinearizedProblem`, conforme indica a Figura 49.

Figura 47 – salvar derivadas referentes ao termo de acumulação



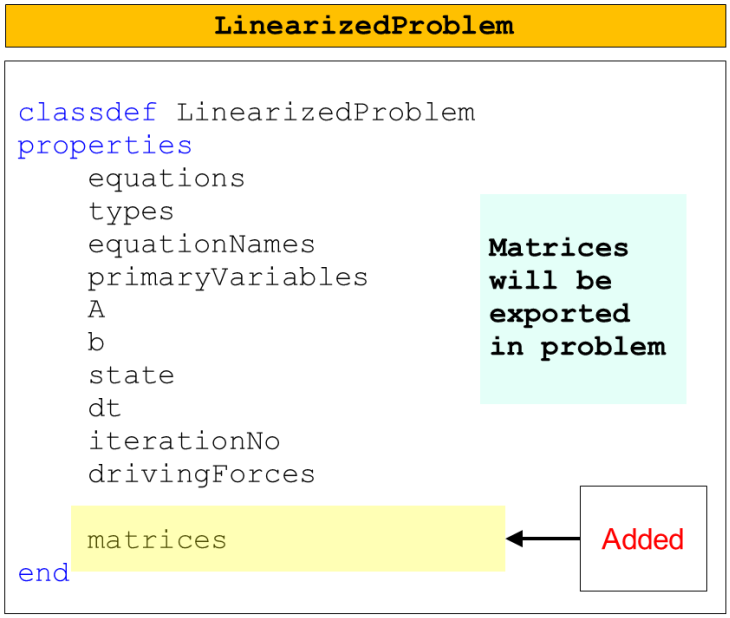
Fonte: A autora (2020).

Figura 48 – Salvar derivadas referentes ao termo de poço



Fonte: A autora (2020).

Figura 49 – Exportação das derivadas dos termos de acumulação e dos poços



Fonte: A autora (2020).

Dessa forma, todas as variáveis disponíveis internamente e requeridas para o método foram incorporadas na variável `report`. Para realizar os últimos ajustes, tais como, concatenar as equações, corrigir o termo de acumulação e salvar todas as variáveis requeridas no TPWL em uma única estrutura, é criada uma rotina auxiliar. Essa rotina extrai os resultados do report para obter as derivadas, e em conjunto com os estados, incremento de tempo e controle dos poços, os resultados são salvos em uma única estrutura, denominada `TPWL`. Essa estrutura será fornecida como entrada para a rotina `tpwlScript` (disponível no **APÊNDICE C – CÓDIGOS AUXILIARES**), onde o algoritmo do TPWL será executado. Cabe destacar que para recuperação das vazões foi utilizada uma função proveniente da rotina do MRST.

- **Metodologia 2**

Embora a forma explicitada anteriormente produza resultados satisfatórios, destaca-se que o MRST disponibiliza uma nova versão a cada 6 meses, com constantes alterações das estruturas de armazenamento do MRST AD-OO. Portanto, a cada nova versão seria necessário repetir o mesmo procedimento descrito na Metodologia 1, bem como, verificar se seu funcionamento está correto nessa versão atualizada.

Portanto, a fim de minimizar eventuais transtornos futuros, procurou-se desenvolver a.

Dessa forma, propõe-se a definição de uma função denominada `equationsOilWaterTPWL`. Essa função é totalmente similar a `equationsOilWater`,

porém a variável `matrices`, tal como descrito pela Figura 47 e Figura 48, é incorporada ao argumento de saída. A sintaxe dessa função será dada por:

Quadro 27 – Sintaxe equationsOilWaterTPWL

FUNÇÃO SINTAXE:
`[problem, state,matrices] = equationsOilWaterTPWL(state0, state, model, dt, drivingForces, varargin)`

Fonte: Adaptado do MRST (2018).

Para utilizar essa função, foi criada uma rotina, denominada exportTPWL, apresentada no Algoritmo 55. Essa rotina, inicialmente carrega os resultados da simulação de treinamento provenientes da função `simulateScheduleAD` além do modelo (`model`), estado inicial (`state0`) e *schedule*. Em seguida, ela define um *loop*, onde são fornecidos os argumentos de entrada para `equationsOilWaterTPWL`:

- `state0`, consiste no estado convergido do passo de tempo anterior `states(i-1);`
- `state`, consiste no estado convergido do passo de tempo atual;
- `model`, modelo definido na simulação;
- `dt`, passo de tempo armazenado em `schedule.step.val`;
- `drivingForces`, consiste no controle definido para o passo de tempo em questão, armazenado em `schedule.control`;
- `varargin`, são argumentos de entrada extra. É importante definir o número da iteração em que houve a convergência: `'iteration`, `report.Iterations(i).`

Dessa forma, é retornada a matriz jacobiana e os termos necessários à implementação do TPWL para cada passo de tempo. Em seguida, de forma análoga a metodologia anterior, serão realizados os últimos ajustes, tais como, concatenar as equações, corrigir o termo de acumulação e salvar todas as variáveis requeridas no TPWL em uma única estrutura, denominada `TPWL`. Essa estrutura será fornecida como entrada para a rotina `tpwlScript`. No **APÊNDICE C – CÓDIGOS AUXILIARES** é apresentado o código de exportação completo.

Embora essa metodologia envolva um custo adicional para recalcular as matrizes ela deverá ser empregada em virtude da não consolidação da estrutura do MRST AD-OO nas versões atualmente disponibilizadas.

Algoritmo 55 – Código do exportTPWL

```
clear all
load('SimulationResults_training.mat')

dt=schedule.step.val;
problem=cell(numel(dt),1);
matrices = cell(numel(dt),1);

for i=1:numel(dt)
 if i~=1
  state0=states{i-1};
 else
  ctrl = schedule.control(schedule.step.control(1));
  [forces, fstruct] = model.getDrivingForces(ctrl);
  model = model.validateModel(fstruct);
  state0 = model.validateState(state0);
 end
 j = schedule.step.control(i);
 [problem{i},~,matrices{i,1}] =
... equationsOilWaterTPWL(state0, states{i}, model, dt(i),
...schedule.control(j),'ResOnly',false,'ResOnly', ...
false,'iteration',report.Iterations(i));

 problem{i}=problem{i}.assembleSystem();

end

```

Fonte: A autora (2020).

5 RESULTADOS E DISCUSSÕES

Nessa seção, serão apresentados alguns estudos comparativos, com o objetivo de avaliar a qualidade da aproximação realizada pela técnica do TPWL.

5.1 CRITÉRIOS PARA COMPARAÇÃO DOS RESULTADOS

Para a comparação dos resultados, foram analisadas as vazões dos poços, o valor presente líquido (VPL) e o tempo de execução da simulação de referência e simulação TPWL, permitindo a análise de acurácia e eficácia do método implementado, conforme detalhado a seguir.

5.1.1 Quantificação dos erros das vazões

Para os casos estudados nas seções seguintes, é importante que haja uma quantificação dos erros com o intuito de mensurar a qualidade da informação. Essa quantificação de erros pode ser realizada de várias maneiras.

No presente trabalho, conforme Cardoso (2009) será aplicado um procedimento simples, baseado no erro médio das vazões óleo e água (injetado e produzido). Para cada poço produtor ou injetor *m*, a cada passo de tempo simulado *i*, define-se a vazão da fase α (água ou óleo) na simulação de referência por ($Q_{\alpha,hf}^{m,i}$) e em cada simulação TPWL como ($Q_{\alpha,tpwl}^{m,i}$). O erro médio de cada poço é designado por E^m, correspondente à média das diferenças absolutas, normalizada pela vazão média no tempo para o poço ($\bar{Q}_{\alpha,hf}^{m}$), conforme indica a Eq. (1). O erro médio total, designado por E, é dado pela Eq. (2).

$$\mathrm{E^m} = \frac{1}{\mathrm{n_t}\bar{\mathrm{Q}}_{\alpha,\mathrm{hf}}^{\mathrm{m}}} \sum \left| \mathrm{Q}_{\alpha,\mathrm{tpwl}}^{\mathrm{m,i}} - \mathrm{Q}_{\alpha,\mathrm{hf}}^{\mathrm{m,i}} \right| \tag{1}$$

Onde n_t é o número total de passos de tempo.

$$E = \frac{1}{n_w} \sum_{m=1}^{n_w} E^m \tag{2}$$

Onde n_w é o número de poços.

Dessa forma para obter o erro médio da vazão de óleo, primeiramente determina-se o erro médio de cada do poço produtor (através da diferença do valor absoluto entre as vazões de óleo da simulação de TPWL e simulação de alta fidelidade, normalizada pela vazão média de óleo no tempo dessa simulação) e, em seguida, realiza-se a média para todos os poços produtores do campo, conforme indica a Eq. (3). Para a vazão de água injetada e água produzida deve-se proceder da mesma forma.

$$E^m = \frac{1}{n_t \bar{Q}^m_{o,hf}} \sum \left| Q^{m,i}_{o,tpwl} - Q^{m,i}_{o,hf} \right| \qquad E = \frac{1}{n_w} \sum_{m=1}^{n_w} E^m \tag{3}$$

5.1.2 Comparação do VPL

Outro critério empregado é a comparação dos resultados em termos do VPL (Valor presente Líquido). Essa variável é de bastante interesse visto que essa é a função objetivo em problemas de otimização de reservatórios de petróleo. Se a simulação TPWL consegue predizer com precisão o VPL da simulação de alta fidelidade, o emprego desse método em problemas de otimização é justificado.

$$VPL = \sum_{t=0}^{n} \frac{F}{\left(1+i\right)^t} \text{ com } F = q_{op} \times P_o - q_{wp} \times C_{wp} - q_{wi} \times C_{wi} \tag{4}$$

Onde: F = fluxo de caixa;

i = taxa de desconto;

n = número de períodos no futuro que o fluxo de caixa está;

q_{op} = vazão de óleo produzido; q_{wp} = vazão de água produzido; q_{wi} = vazão de água injetada;

P_o = preço do óleo; C_{wp} = custo de água produzida; C_{wi} = custo de água injetada.

Como parâmetros teóricos foi estabelecido o preço do óleo é 100 $/BBL e os custos de injeção e produção de água são de 10 $/BBL e a taxa de desconto é de 10% a. a.

Para medir a qualidade da aproximação, foi utilizado o erro médio quadrático. Considerando VPL_{REF} e VPL_{TPWL}, respectivamente, o valor calculado com as vazões produzidas pelo simulador de referência e o calculado pela simulação TPWL. Seja N o numero de simulações em cada caso considerado, a Equação obtida é dada por:

$$RMSD = \sqrt{\frac{\sum_{i=1}^{N}\left(\frac{VPL_{TPWL} - VPL_{REF}}{VPL_{REF}}\right)^2}{N}} \times 100\% \quad (5)$$

5.1.3 Comparação do Tempo de Simulação

Outro importante parâmetro de comparação diz respeito aos tempos de simulação. A utilização da técnica TPWL é satisfatória por reproduzir os resultados do modelo de alta fidelidade consumindo menos tempo de simulação. Para quantificar essa diferença será empregado o *speedup*, ou seja, a razão entre os tempos de simulação de alta e baixa fidelidade.

5.2 MODELO DE RESERVATÓRIO I

- **Descrição do Modelo**

Inicialmente, a fim de verificar a implementação do TWPL, foi utilizado o modelo do micro reservatório descrito na seção 3.10. Dessa forma, o modelo consiste em uma malha cartesiana 3D, composta por [3 x 3 x 3] elementos, com dimensões de [50 m x 50 m x 5 m], com permeabilidade e porosidade constante nos valores de 100 mD e 0.2, respectivamente. O fluido incompressível bifásico, com densidade da água (ρ_w) 1000 kg/m^3 e densidade do óleo (ρ_o) 850 kg/m^3, ambos com viscosidade μ =1 cP e permeabilidade relativa quadrática. O estado inicial é determinado pelas condições de equilíbrio hidrostático considerando uma pressão de referência de 100 bar no topo do reservatório. São definidos dois poços: um injetor e outro produtor. O injetor é controlado por BHP com valor de 150 bar, está localizado no canto inferior esquerdo da malha, com completações nas duas últimas camadas. O produtor também terá o controle do tipo BHP no valor de 50 bar, estando localizado no canto superior direito, com completação nas duas primeiras camadas. O s*chedule* foi definido para um horizonte de simulação de 600 dias, com passos de tempo crescendo geometricamente até o intervalo constante de 30 dias. O estado inicial está representado na Figura 50.

- **Simulação de Treinamento**

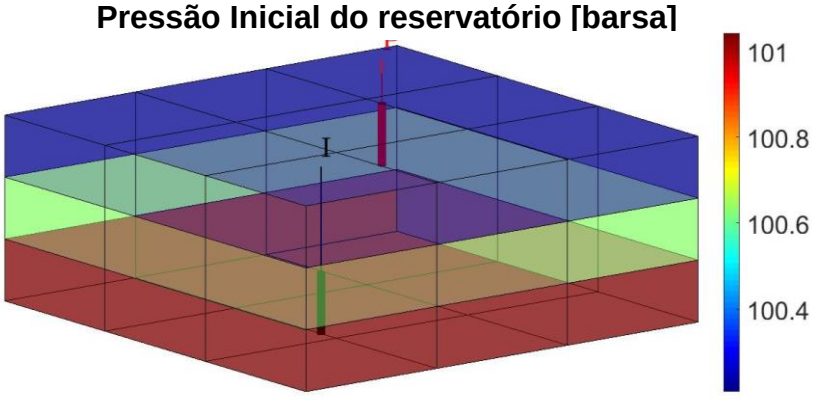
Para a simulação de treinamento, foi estabelecido um *schedule* randômico, com 10 intervalos de controles variando randomicamente entre 50 e 80 bar. Esse *schedule* será fornecido ao simulador *simulateschedule*AD, para ser obtida a vazão dos poços, estado do reservatório e as informações requeridas para a implementação do TPWL.

Inicialmente, como teste inicial, o *schedule* de treinamento foi utilizado no código do TPWL, para verificar se a correta implementação do código. Dessa forma, espera-se que os resultados do estado do reservatório e das vazões nos poços sejam idênticos ao da simulação de treinamento. Essas verificações garantem que o código do TPWL, assim como, cálculo das vazões nos poços estão corretamente implementados.

Em seguida são fornecidos novos controles, tanto para o simulador (simulação de alta fidelidade) como para o código do TPWL (criado os estados armazenados na simulação de treinamento, ou seja, com *schedul*e definido da Figura 50). Esses novos controles também foram definidos randomicamente no intervalo de 50 a 80 bar.

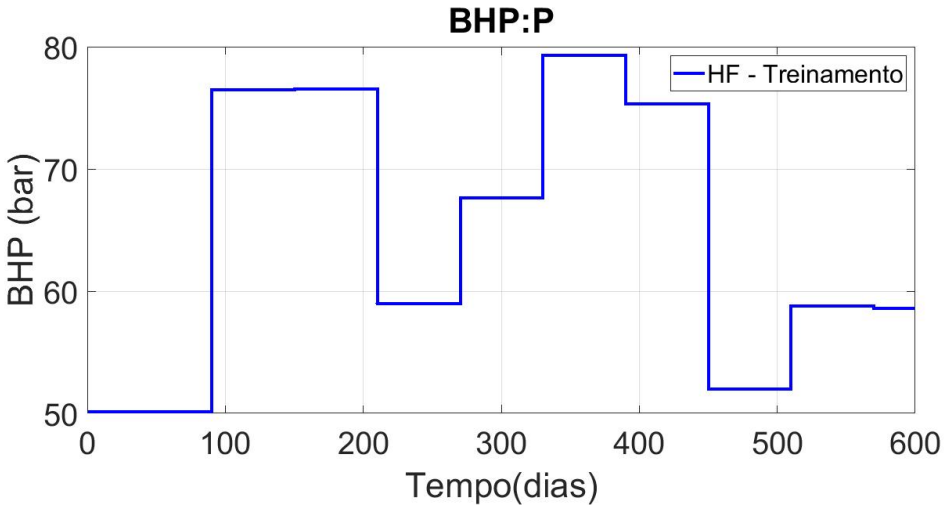
Os resultados desse teste podem ser observados na Figura 52, onde a linha azul contínua representa a simulação de treinamento, a linha azul tracejada representa a simulação TPWL para o mesmo *schedule* de treinamento, portanto, nesse caso, ocorreu o esperado: correspondência exata dos resultados. Já a linha vermelha contínua representa a simulação MRST para num novo *schedule* aleatório, enquanto a linha rosa tracejada representa a simulação TPWL para o novo *schedule*. Os erros médios dessa simulação estão expressos na Tabela 1.

Figura 50 – Modelo do microreservatório e *schedule* de treinamento



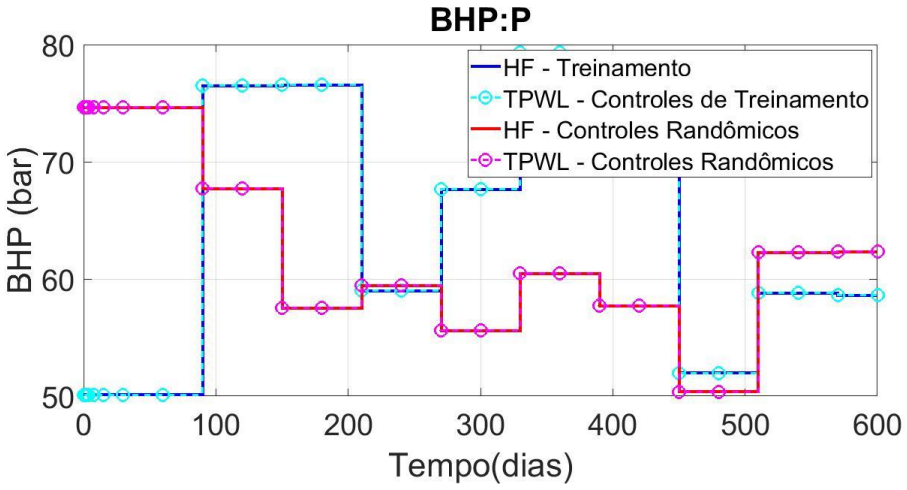
Fonte: A autora (2020).

Figura 51 – Schedule de Treinamento

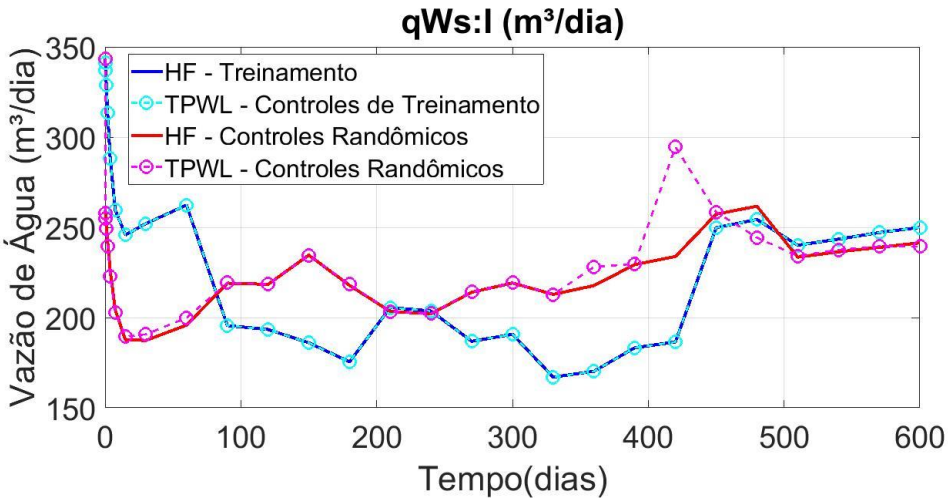


Fonte: A autora (2020).

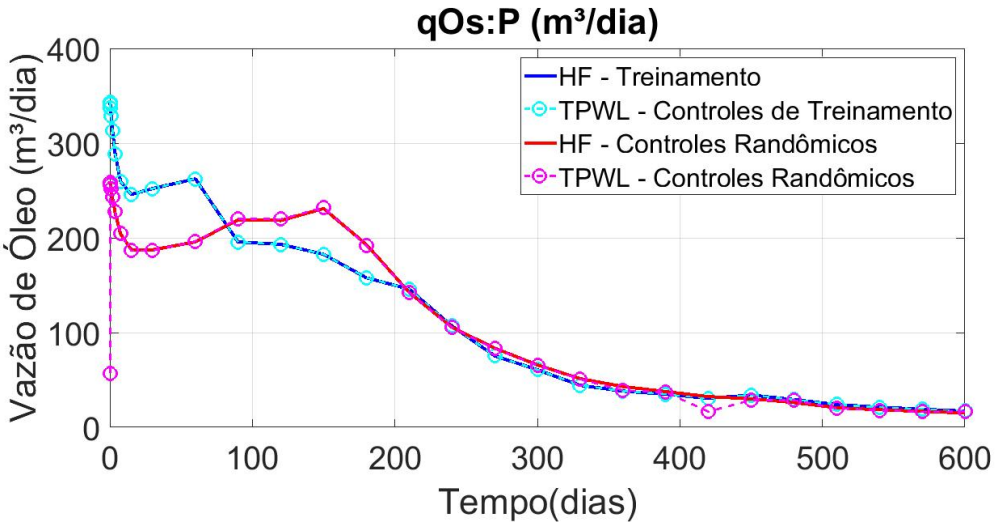
Figura 52 – Teste 1 – verificação da implementação do TPWL



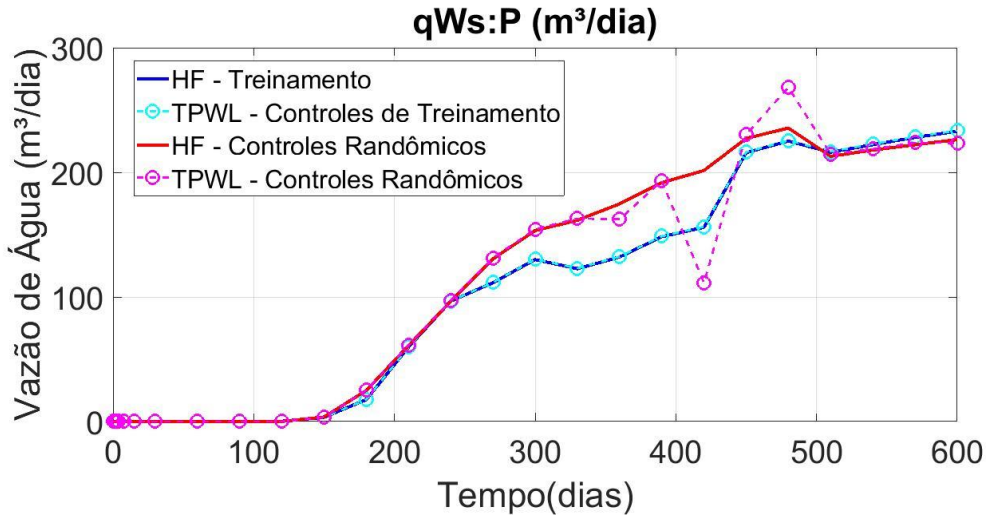
(a) Schedule de treinamento e simulação TWPL



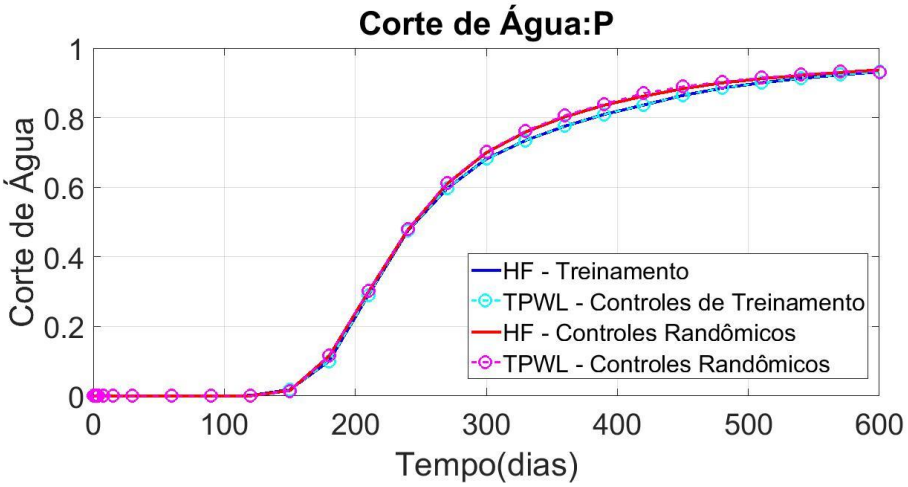
(b) Vazão de água no poço injetor



(c) Vazão de óleo no poço produtor



(d) Vazão de água no poço produtor



(e) Corte de Água

Fonte: A autora (2020).

Tabela 1 – Erro Médio

Vazões	**Erro médio E**
Água injetada ($qWs\ I$)	0,0314
Água Produzida($qWs\ P$)	0,0594
Óleo Produzido($qOs\ P$)	0,0635

Fonte: A autora (2020).

Observe que por se tratar de um micro reservatório, são gerados alguns pontos fora da curva, acentuando eventuais alterações, principalmente onde o *schedule* de treinamento difere do novo *schedule*. Em modelos completos de reservatórios, espera-se que esses efeitos sejam levemente atenuados.

5.3 MODELO DE RESERVATÓRIO II –SPE 10

Nessa seção, será descrito e estudado o The 10th SPE Comparative Solution Project (SPE10), modelo *benchmark* na indústria de petróleo. Dessa forma, será possível avaliar como a técnica implementada se comporta em um reservatório heterogêneo, em uma malha computacional de número considerável de elementos. Inicialmente, será realizada a descrição

do reservatório e validação dos resultados no MRST com um simulador comercial. Em sequida serão descritos os testes realizados.

5.3.1 Descrição do Modelo

O modelo proposto é uma porção modificada do SPE10. O SPE 10 foi desenvolvido por Christie e Blunt (2001) como benchmark para comparação do desempenho de diferentes simuladores ou algoritmos. O modelo possui geometria simples, descrita por uma malha cartesiana regular de 60 x 220 x 85 células com dimensões de 1200 ft x 2200 ft x170 ft. As 35 camadas superiores (70 ft) representam a formação Tarbert, similar ao ambiente costeiro; enquanto as demais (50 camadas, 100 ft) representam a formação fluvial Upper Ness.

A porção modificada a ser utilizada foi determinada em Cardoso (2009) como um dos modelos para aplicação da técnica TPWL, e também estudada por Machado (2014).

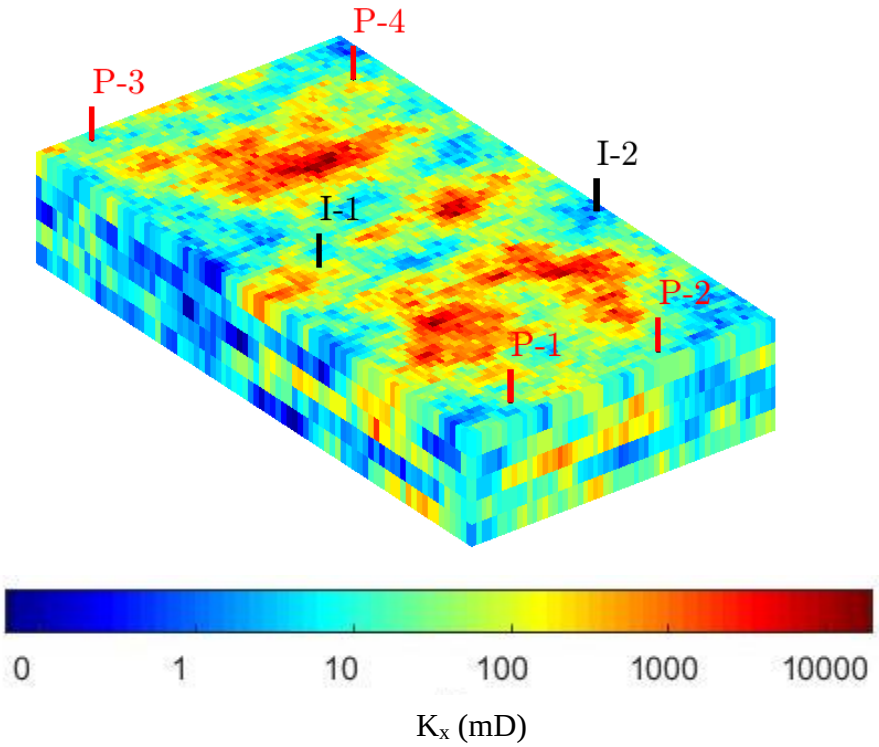
O modelo modificado é tridimensional, composto por 24000 células, dispostas na malha cartesiana de 60 x 220 x 5 células. Todas as células possuem a dimensão de 50 ft x 70 ft x20 ft. Existem quatro poços produtores (designados P1-P4) e dois poços injetores (designados I1 e I2).

A permeabilidade média na direção x é de $k_x = 418$ mD. A permeabilidade da direção y é igual a x ($k_x = k_y$). No que diz respeito à direção z, a permeabilidade vertical (k_z) é prescrita como $k_z = 0.3k_x$ nos canais e $k_z = 10^{-3}k_x$ nas demais regiões. A porosidade média é de 0,203.

Com relação às propriedades do fluido para o óleo define-se $\rho_o = 45\,\text{lb}/\text{ft}^3$ e $\mu_o = 3\,\text{cp}$, para água, tem-se $\rho_w = 60\,\text{lb}/\text{ft}^3$, $\mu_w = 0,3\,\text{cp}$. O sistema é pouco compressível $c_o = c_w = c_r = 10^{-6}\,\text{psi}^{-1}$ e os efeitos da pressão capilar são negligenciados.

As saturações iniciais de óleo e água são respectivamente de $S_{oi} = 0,8$ e $S_{wc} = 0,8$ e as saturações de óleo residual e água são $S_{or} = S_{wr} = 0,2$. As permeabilidades relativas são calculadas através do modelo de Corey, com os seguintes parâmetros $k_{ro}^0 = k_{rw}^0 = 1$ e $a = b = 2$. A Figura 53 apresenta a ilustração do modelo, criado com o uso do MRST.

Figura 53 – Modelo sintético (baseado no SPE10)



Fonte: A autora (2020).

5.3.2 Validação com simulador comercial

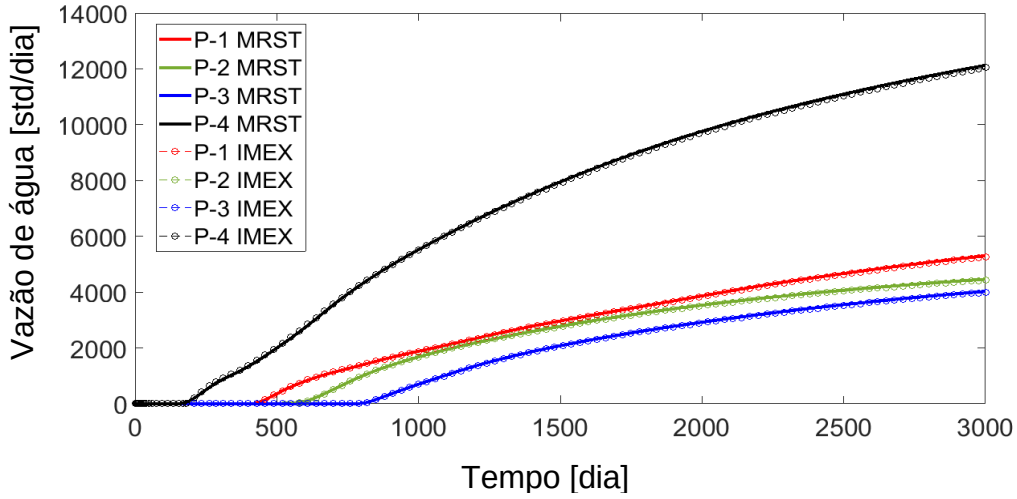
Como uma tentativa de validação do simulador, buscou-se comparar os resultados da simulação de um modelo de reservatórios obtida com o uso do MRST com o resultado obtido com algum simulador comercial, já consagrado pela indústria. O simulador comercial escolhido para essa comparação foi o IMEX da CMG.

Neste teste comparativo, todos os poços foram controlados por BHP, sendo para os produtores P_{bhp} = 4000 psi e para os injetores P_{bhp} = 11000 psi. O tempo da simulação é de 3000 dias.

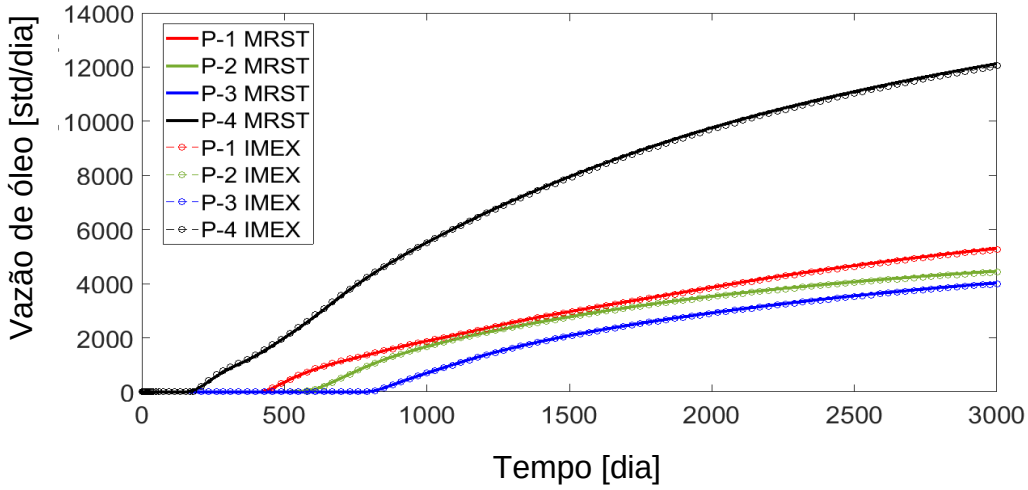
A Figura 54 apresenta os resultados obtidos para este modelo. A comparação foi feita considerando as vazões de óleo produzido (Figura 54 (a)), a vazão de água produzida (Figura 54(b)), e a vazão de água injetada (Figura 54(c)) por poço.

Estes mostraram quase perfeita aderência entre as vazões calculadas pelos dois simuladores. Portanto, as diferenças existentes entre os dois simuladores não possuem efeito significativo nos resultados, validando a utilização desse modelo no MRST.

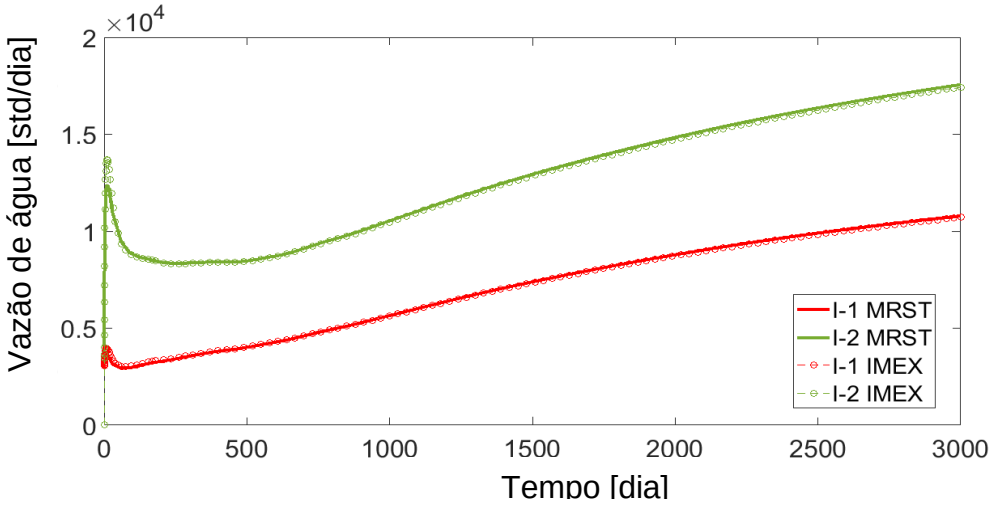
Figura 54 – Comparação de Resultados do MRST x IMEX - SPE10 Simplificado



(a) Vazão de óleo produzido por poço produtor



(b) Vazão de água produzido por poço produtor



(c) Vazão de água injetada

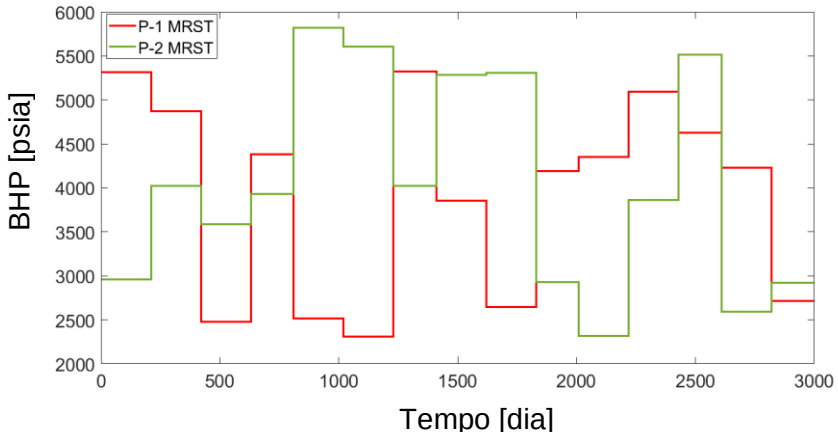
Fonte: A autora (2020).

5.3.3 Teste I – Modelo Incompressível (Treinamento 1)

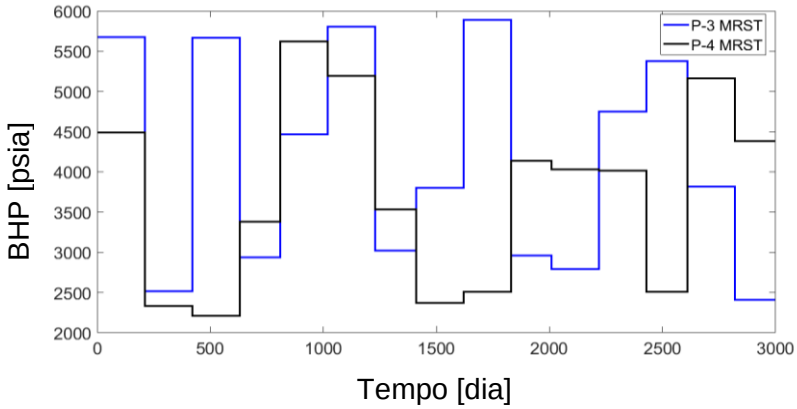
Com o objetivo de avaliar a qualidade da aproximação realizada pela técnica do TWPL, inicialmente, foram executados alguns estudos comparativos considerando algumas variações do modelo. O primeiro considera que as compressibilidades do óleo, água e rocha são nulas e as densidades dos fluidos são dadas por $\rho_o = 45\,\mathrm{lb}/\mathrm{ft}^3$ e $\rho_w = 60\,\mathrm{lb}/\mathrm{ft}^3$.

O tempo da simulação é de 3000 dias e o ciclo de controle é 200 dias, resultando em 15 ciclos de controle. Na simulação de treinamento, a pressão no fundo de poço para os injetores é fixada em P_{bhp} = 12000 psi. A sequência de controles para a simulação nos produtores é variável entre P_{bhp} = 2000 psi e P_{bhp} = 6000 psi, como apresentado na Figura 55.

Figura 55 – Sequência de controles de treinamento (BHP) aplicados aos poços produtores



(a) BHP nos poços P-1 e P-2



(b) BHP nos poços P-1 e P-2

Fonte: A autora (2020).

Em seguida, assim como em Machado (2014), são executadas no MRST simulações com 3 ciclos de controle a cada 1000 dias. São três valores avaliados em cada ciclo – 2000 psi (limite inferior), 4000 psi (meio da escala) e 6000 psi (limite superior). São admitidos os mesmos controles para os poços produtores. Dessa forma, são três valores e três ciclos que determinam a realização de 27 simulações, conforme apresentado no Quadro 28.

Por fim, executam-se as mesmas simulações com o TPWL utilizando como simulação de treinamento o *schedule* randômico da Figura 55.

Quadro 28 – *Schedule* para simulações

SIM	1º ciclo	2º ciclo	3ºciclo
1	MIN	MIN	MIN
2	MED	MIN	MIN
3	MAX	MIN	MIN
4	MIN	MED	MIN
5	MED	MED	MIN
6	MAX	MED	MIN
7	MIN	MAX	MIN
8	MED	MAX	MIN
9	MAX	MAX	MIN
10	MIN	MIN	MED
11	MED	MIN	MED
12	MAX	MIN	MED
13	MIN	MED	MED
14	MED	MED	MED
15	MAX	MED	MED
16	MIN	MAX	MED
17	MED	MAX	MED
18	MAX	MAX	MED
19	MIN	MIN	MAX
20	MED	MIN	MAX
21	MAX	MIN	MAX
22	MIN	MED	MAX
23	MED	MED	MAX
24	MAX	MED	MAX
25	MIN	MAX	MAX
26	MED	MAX	MAX
27	MAX	MAX	MAX

Fonte: A autora (2020).

A comparação dos resultados conforme os critérios apresentados na seção 5.1, resulta na Tabela 2.

Tabela 2 – Teste I - Resultados das simulações TPWL

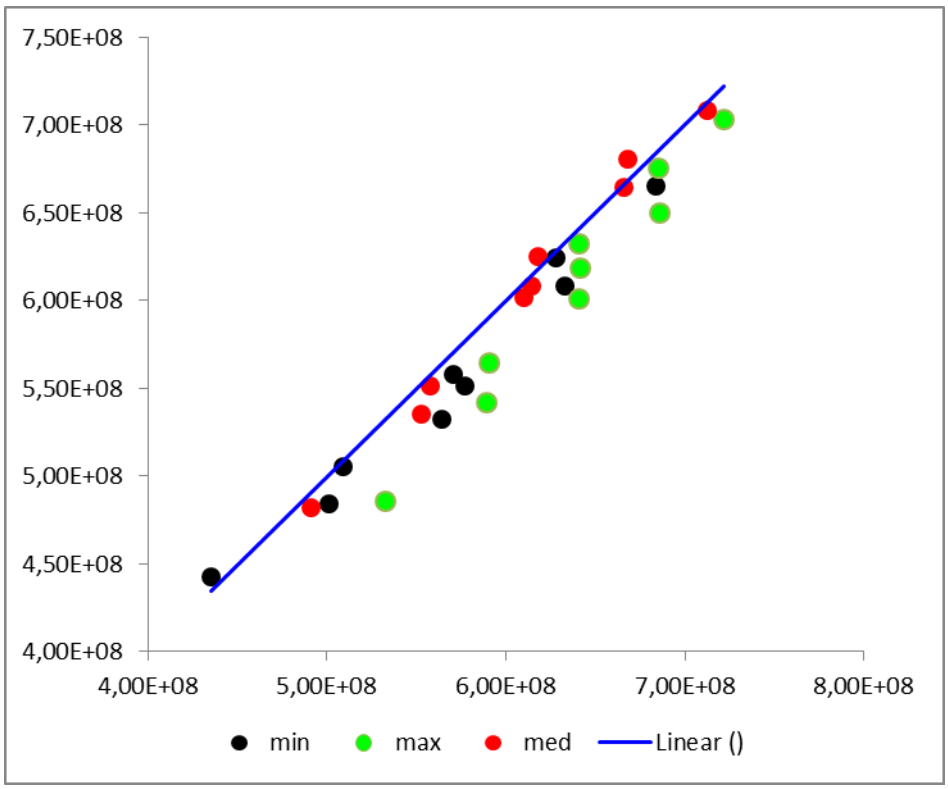
SIM	VPL HF	VPL TPWL	ERRO VPL	TEMPO HF	TEMPO TPWL	SPEEDUP	ERRO QWS PROD	ERRO QOS PROD
1	4,35E+08	4,43E+08	-0,02	511,60	161,01	3,18	0,31	0,22
2	4,91E+08	4,82E+08	0,02	498,62	160,42	3,11	0,23	0,10
3	5,32E+08	4,86E+08	0,09	503,70	156,23	3,22	0,15	0,20
4	5,09E+08	5,05E+08	0,01	513,82	161,28	3,19	0,29	0,22
5	5,58E+08	5,51E+08	0,01	509,35	159,70	3,19	0,09	0,03
6	5,91E+08	5,65E+08	0,04	503,38	156,61	3,21	0,21	0,26
7	5,77E+08	5,52E+08	0,04	519,84	161,44	3,22	0,17	0,18
8	6,18E+08	6,25E+08	-0,01	519,34	159,08	3,26	0,19	0,10
9	6,41E+08	6,32E+08	0,01	493,06	154,04	3,20	0,35	0,34
10	5,01E+08	4,84E+08	0,03	479,08	163,44	2,93	0,32	0,24
11	5,53E+08	5,35E+08	0,03	468,13	159,78	2,93	0,24	0,10
12	5,89E+08	5,42E+08	0,08	458,22	156,44	2,93	0,08	0,19
13	5,71E+08	5,58E+08	0,02	467,90	161,59	2,90	0,31	0,23
14	6,14E+08	6,08E+08	0,01	463,99	160,99	2,88	0,01	0,01
15	6,42E+08	6,19E+08	0,04	458,89	152,38	3,01	0,35	0,29
16	6,33E+08	6,08E+08	0,04	471,98	163,95	2,88	0,11	0,17
17	6,68E+08	6,80E+08	-0,02	472,61	157,71	3,00	0,32	0,13
18	6,85E+08	6,76E+08	0,01	476,82	154,68	3,08	0,42	0,36
19	5,64E+08	5,32E+08	0,06	478,05	162,72	2,94	0,36	0,25
20	6,10E+08	6,01E+08	0,01	467,47	161,12	2,90	0,21	0,09
21	6,40E+08	6,01E+08	0,06	463,61	155,68	2,98	0,21	0,22
22	6,28E+08	6,24E+08	0,01	474,01	164,11	2,89	0,30	0,22
23	6,66E+08	6,65E+08	0,00	468,11	155,08	3,02	0,15	0,05
24	6,86E+08	6,50E+08	0,05	460,14	152,27	3,02	0,46	0,34
25	6,84E+08	6,65E+08	0,03	462,24	157,44	2,94	0,27	0,21
26	7,13E+08	7,08E+08	0,01	453,84	154,27	2,94	0,42	0,16
27	7,22E+08	7,03E+08	0,03	451,96	154,53	2,92	0,43	0,36
			RMSD:	**MÉDIAS:**				
			3,68%	480,38	158,44	3,03	0,26	0,19

Fonte: A autora (2020).

Note que o erro médio quadrático do VPL (RMSD) resultou em 3,68%, resultado de mesma ordem de grandeza é encontrado em Machado (2014) para análise semelhante. Em Machado (2014) para outro *schedule* randômico do TPWL, no caso sem reitreinamento, incompressível e com diferença de densidade, o RMSD resulta em 8,83%.

Para poder ter uma melhor visualização dos resultados, é construído um *crossplot* no qual o eixo x representa o VPL calculado pelo MRST, que seria a medida real, e o eixo y representa a aproximação pelo modelo TPWL. Sendo assim, se o TPWL fosse perfeito, todos os pontos cairiam sobre a reta identidade em azul. No *crossplot* apresentado, os pontos verdes representam simulações cujo primeiro controle é o valor máximo para a pressão de fundo, os pontos pretos têm o primeiro controle no valor mínimo e os vermelhos o têm primeiro controle no valor intermediário.

Figura 56 – Comparação do VPL calculado pelo MRST e pelo TPWL



Fonte: A autora (2020).

Observa-se certa estratificação, onde o valor utilizado para o primeiro controle divide a nuvem em três grupos de pontos. Isto se deve à grande força do primeiro ciclo de controle na formação do VPL por causa da taxa de desconto, que maximiza a influência de erros do primeiro ciclo no erro total.

Outro importante resultado a se comentar diz respeito aos tempos de simulação. Vale a pena utilizar o TPWL, visto que, este reproduz os resultados do modelo de alta fidelidade consumindo menos tempo de simulação. O tempo médio para a simulação de alta fidelidade foi de 480,38 segundos, enquanto para o modelo TPWL, o tempo médio foi de 158,44 segundos,

aproximadamente 3 vezes mais rápido que a simulação no MRST. Esse resultado representa um ganho modesto com a implementação do TPWL, que deverá ser potencializado com o acoplamento do POD.

Para uma melhor comparação da ordem de grandeza, pode-se observar o estudo realizado por Hewson (2015). No referido estudo, foi simulado o modelo Egg, para o qual a simulação de alta fidelidade no MRST foi de 1100 segundos, enquanto a simulação empregando apenas TPWL teve um tempo de 100 segundos, de forma que o ganho do TWPL foi de 11 vezes. Esse resultado, a despeito da diferença do modelo, demostra similar ordem de grandeza para o tempo de execução.

5.3.4 Teste II – Modelo Compressível (Treinamento 1)

Para o segundo teste, o modelo é considerado compressível, sendo a compressibilidade do óleo $c_o = 3 \times 10^{-6} psi^{-1}$, da água $c_w = 1 \times 10^{-6} psi^{-1}$ e da rocha $c_r = 1 \times 10^{-7} psi^{-1}$. A comparação dos resultados conforme os critérios apresentados na seção 5.1, resulta na Tabela 3 e na elaboração do *crossplot* apresentado na Figura 57.

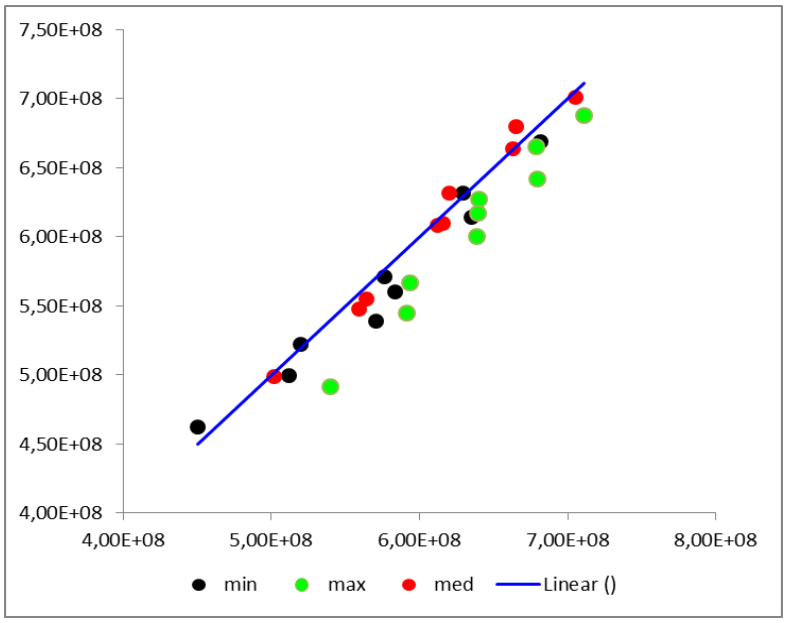
Tabela 3 – Teste II - Resultados das simulações TPWL

SIM	VPL HF	VPL TPWL	ERRO VPL	TEMPO HF	TEMPO TPWL	SPEEDUP	ERRO QWS PROD	ERRO QOS PROD
1	4,50E+08	4,62E+08	-0,03	2027,79	603,95	3,36	0,30	0,21
2	5,02E+08	4,99E+08	0,01	2210,05	659,13	3,35	0,22	0,09
3	5,40E+08	4,91E+08	0,09	2515,27	732,71	3,43	0,15	0,21
4	5,20E+08	5,22E+08	0,00	2063,94	608,42	3,39	0,29	0,21
5	5,64E+08	5,55E+08	0,02	2233,91	664,65	3,36	0,08	0,03
6	5,94E+08	5,67E+08	0,05	2573,88	751,03	3,43	0,22	0,28
7	5,84E+08	5,60E+08	0,04	2097,17	625,50	3,35	0,17	0,18
8	6,20E+08	6,31E+08	-0,02	2307,19	695,43	3,32	0,21	0,11
9	6,40E+08	6,27E+08	0,02	2587,33	758,39	3,41	0,36	0,36
10	5,12E+08	4,99E+08	0,02	2011,35	599,86	3,35	0,32	0,23
11	5,59E+08	5,47E+08	0,02	2193,97	649,61	3,38	0,22	0,10
12	5,91E+08	5,44E+08	0,08	2498,94	724,62	3,45	0,09	0,20
13	5,76E+08	5,71E+08	0,01	2046,19	614,97	3,33	0,31	0,22

SIM	VPL HF	VPL TPWL	ERRO VPL	TEMPO HF	TEMPO TPWL	SPEEDUP	ERRO QWS PROD	ERRO QOS PROD
14	6,16E+08	6,10E+08	0,01	2228,63	678,00	3,29	0,01	0,01
15	6,39E+08	6,17E+08	0,03	2517,71	744,04	3,38	0,35	0,31
16	6,35E+08	6,14E+08	0,03	2090,09	623,74	3,35	0,12	0,16
17	6,65E+08	6,79E+08	-0,02	2268,33	688,46	3,29	0,31	0,13
18	6,79E+08	6,65E+08	0,02	2574,37	758,19	3,40	0,43	0,38
19	5,71E+08	5,39E+08	0,05	2001,26	608,30	3,29	0,35	0,24
20	6,12E+08	6,08E+08	0,01	2160,69	676,68	3,19	0,22	0,08
21	6,39E+08	6,00E+08	0,06	2463,68	740,62	3,33	0,21	0,23
22	6,29E+08	6,31E+08	0,00	1993,45	615,99	3,24	0,31	0,21
23	6,63E+08	6,64E+08	0,00	2130,51	682,05	3,12	0,14	0,04
24	6,80E+08	6,42E+08	0,06	2448,26	750,26	3,26	0,46	0,35
25	6,82E+08	6,69E+08	0,02	1956,21	664,26	2,94	0,27	0,20
26	7,06E+08	7,01E+08	0,01	2168,28	693,56	3,13	0,40	0,16
27	7,11E+08	6,88E+08	0,03	2481,61	776,78	3,19	0,44	0,39
RMSD:				**MÉDIAS:**				
3,63%				2253,71	681,08	3,31	0,26	0,20

Fonte: A autora (2020).

Figura 57 – Comparação do VPL calculado pelo MRST e pelo TPWL



Fonte: A autora (2020).

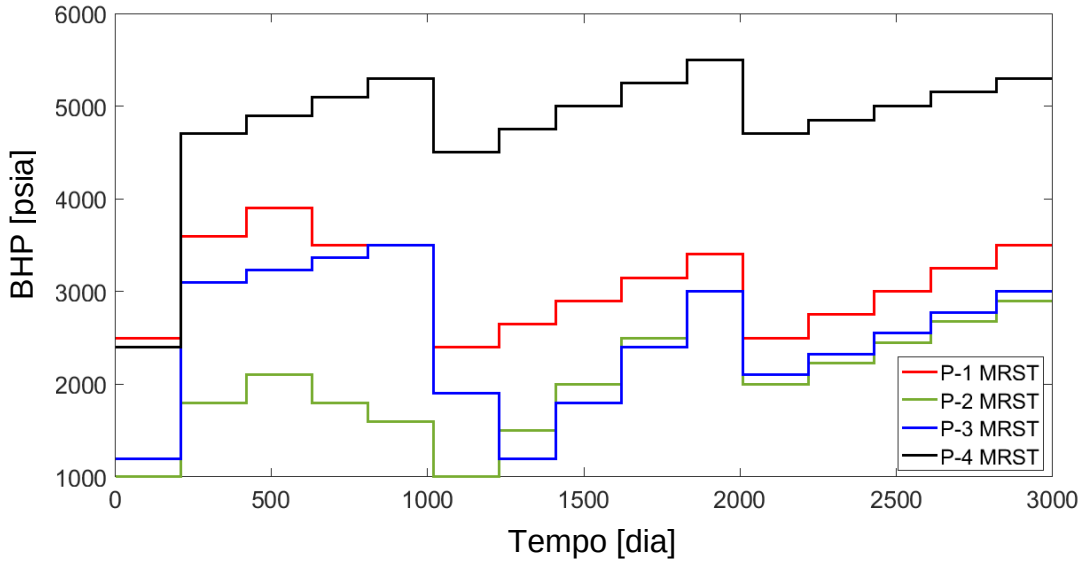
Os resultados indicam que a inclusão da compressibilidade no modelo SPE10 também produz resultados satisfatórios para o TPWL em comparação ao MRST. Os valores obtidos para os erros da vazão de óleo e vazão de água, bem como do erro médio quadrático do VPL (RMSD), são equiparáveis ao caso incompressível. Este último é dado pelo valor de 3,63%. Contudo o tempo médio de simulação do MRST foi de 2253,71 segundos, já para o TPWL ele foi de 681,08 segundos. Note que, embora o tempo de simulação tenha aumento, o *speedup* se manteve próximo aos 3, com uma redução de 70% no tempo da simulação. Cabe destacar que o tempo contabilizado para a simulação TPWL inclui a execução da Equação (7), determinando os estados do reservatório, assim como, o cálculo da vazão nos poços, obtendo, portanto, os mesmos *outputs* da simulação MRST.

Em estudo similar para o modelo compressível, Machado (2014) na simulação de alta fidelidade utilizando o SIMPAR levou em média 1873 segundos, considerando-se as 27 simulações realizadas no estudo. Por outro lado, as simulações correspondentes do TPWL/POD custaram em média 4,8 s, o que resulta em um *speedup* de 390. Esse valor é bastante superior ao mencionado, em virtude do acoplamento do TPWL ao POD. Dessa forma, espera-se que com a implementação do POD esse ganho esteja na faixa de 100-500 vezes, conforme apresentado em Cardoso (2009) e He (2013) e Machado (2014).

5.3.5 Teste III – Modelo Compressível (Treinamento 2)

Nesse caso, utiliza-se o mesmo modelo compressível, porém foi estabelecido outro *schedule* para o treinamento. O tempo da simulação é de 3000 dias e o ciclo de controle é 200 dias, resultando em 15 ciclos. Com relação aos controles, eles variam entre P_{bhp} = 1000 psi e P_{bhp} = 6000 psi, similar ao definido em Cardoso (2009).

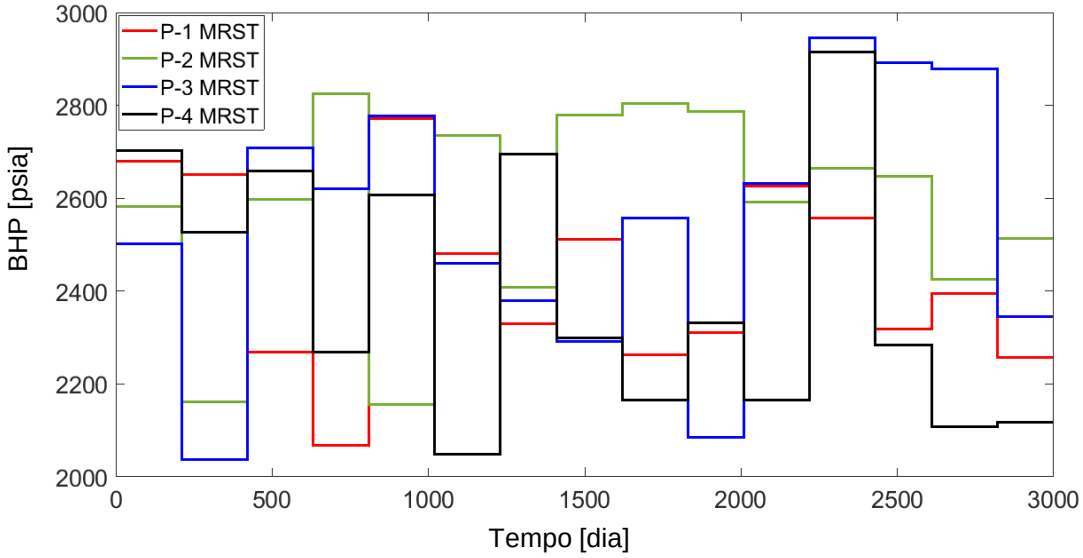
Figura 58 – Sequência de controles de treinamento (BHP) aplicados aos poços produtores



Fonte: A autora (2020).

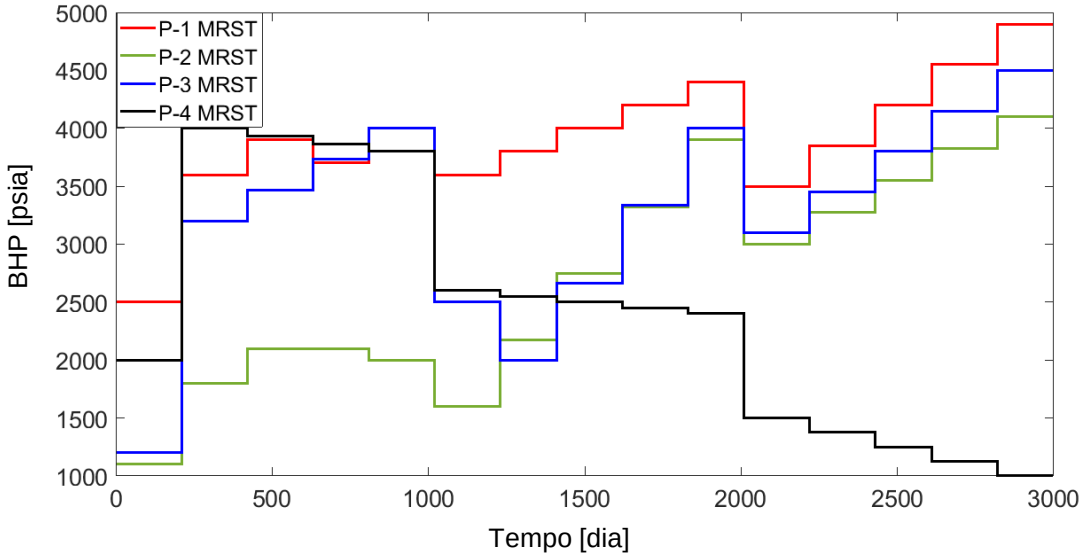
Para testar a simulação TPWL foram definidos dois novos *schedules*. Para a simulação A, a sequência de controles (pressão do fundo de poço) varia randomicamente com 15 ciclos de controle, entre 2000 e 3000 psia, conforme apresentado na Figura 59. Na simulação B a sequência de controles também varia aleatoriamente, também com 15 ciclos de controle, conforme indica a Figura 60.

Figura 59 – Sequência de controles aplicados aos poços produtores na simulação A



Fonte: A autora (2020).

Figura 60 – Sequência de controles aplicados aos poços produtores na simulação B



Fonte: A autora (2020).

Após realizar a simulação do MRST para ambos *schedules* e realizar a simulação TPWL, é possível realizar o estudo comparativo para o emprego do TPWL. A Tabela 4 apresenta os resultados. É possível observar que o erro médio na vazão de óleo e água são relativamente pequenos, bem como, no valor do VPL. Portando, o desempenho do TPWL para as simulações de teste foi satisfatório.

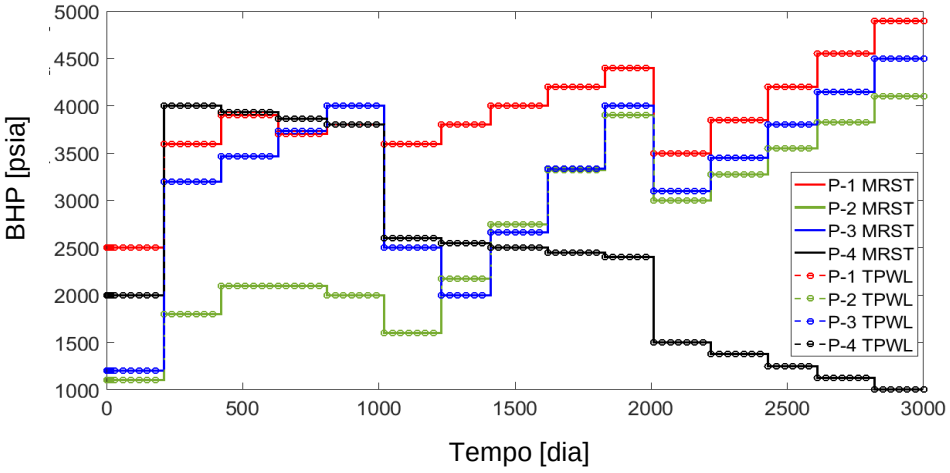
Tabela 4 – Teste III - Resultados das simulações TPWL

SIM	VPL HF	VPL TPWL	ERRO VPL	TEMPO HF	TEMPO TPWL	SPEEDUP	ERRO QWS PROD	ERRO QOS PROD
A	4,91E+08	4,71E+08	0,04	2259,60	600,25	3,76	0,08	0,04
B	5,19E+08	5,03E+08	0,03	2112,20	609,59	3,46	0,04	0,01
RMSD:				**MÉDIAS:**				
3,63%				2185,90	604,92	3,61	0,06	0,03

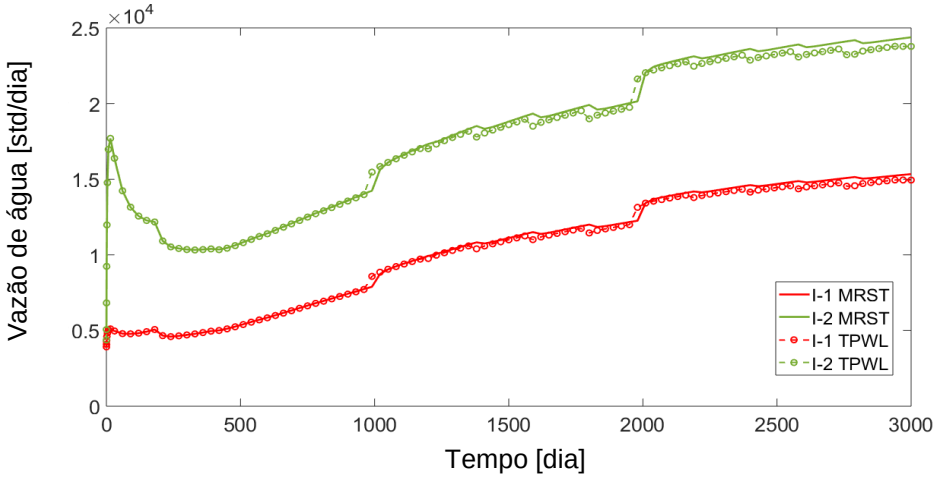
Fonte: A autora (2020).

Analisando com mais detalhes o teste B, nota-se que o Poço P-4 apresenta um grande desvio comparado à simulação de treinamento, sendo, portanto, um teste mais desafiador para o modelo TPWL. A Figura 61 apresenta a comparação da simulação de alta fidelidade e simulação TPWL para esse teste.

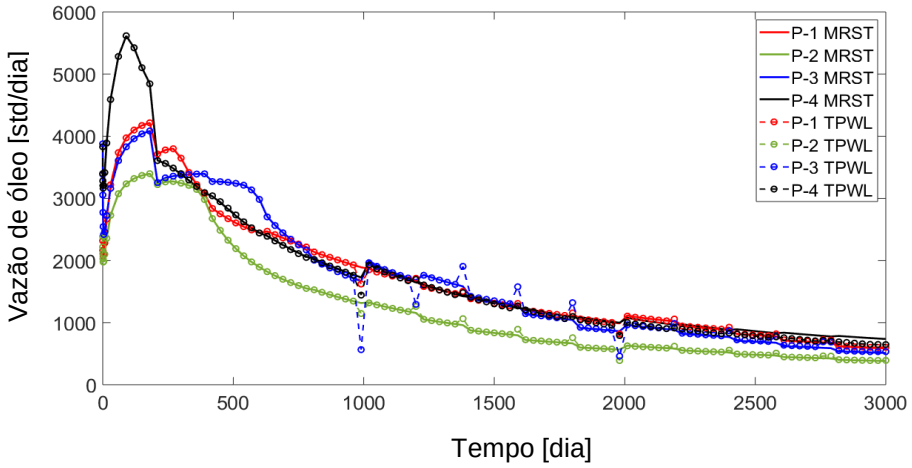
Figura 61 – Resultados Gráficos da Simulação B



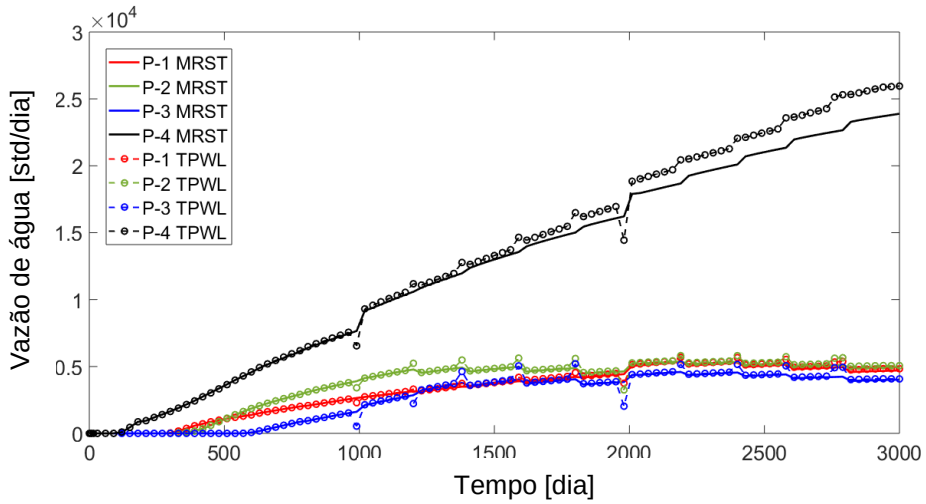
(a) Schedule de treinamento e simulação TWPL



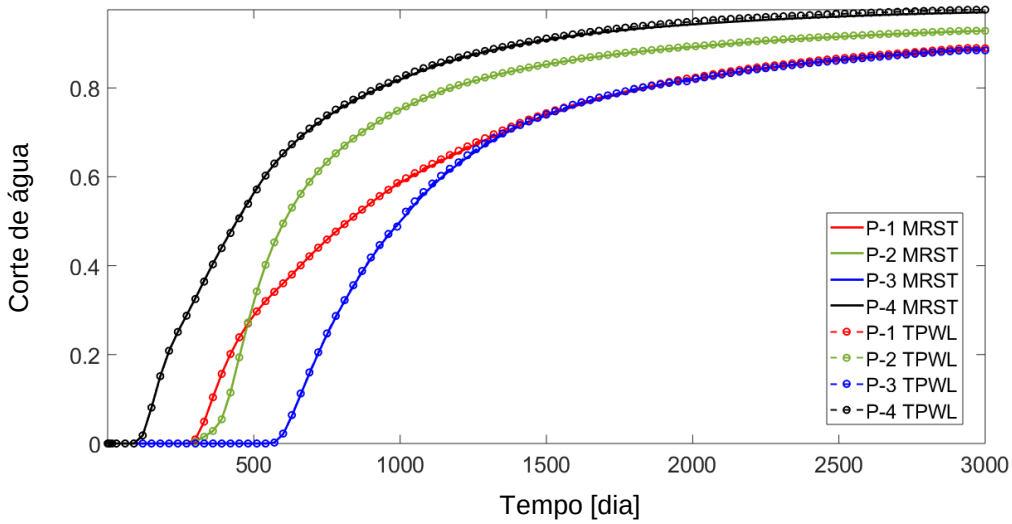
(b) Vazão de água no poço injetor



(c) Vazão de óleo no poço produtor



(d) Vazão de água no poço produtor



(e) Corte de Água

Fonte: A autora (2020).

Note que no instante da mudança de controle, a vazão de água e óleo apresentam pontualmente um valor fora da curva, porém, esse desvio é rapidamente corrigido. O poço P-4, em virtude da sua discrepância do seu controle com o estabelecido pela simulação de treinamento apresentou um maior erro, comparado aos demais. Porém esse valor não é destoante do conjunto. O erro médio da produção de água no poço P-4 é de 0.0594, enquanto no poço P-1 esse valor é de 0.0307. Na produção de óleo o poço P-4 apresenta um erro médio 0,0228 de e o poço P-1 apresenta um erro no valor de 0,0062, conforme indicam os resultados da Tabela 5.

Tabela 5 – Teste B - Erro Médio da vazão por poço

Poço Produtor	ERRO QWS PROD	ERRO QOS PROD	Poço Injetor	ERRO QWS INJ
P-1	0,0307	0,0062	**I -1**	0,0148
P-2	0,0292	0,0080	**I -2**	0,0139
P-3	0,0489	0,0207	-	
P-4	0,0594	0,0228	-	
Média	**0,04205**	**0,014425**	**Média**	**0,01435**

Fonte: A autora (2020).

Em teste similar, realizado em Cardoso (2009), o erro médio da vazão de água produzida foi de 0,0382; da vazão de óleo produzido foi de 0,0166; e da taxa de água injetada foi de 0,0149. Note que esses valores estão bastante próximos ao estudo comparativo aqui realizado.

É importante destacar que o resultado da simulação TWPL foi satisfatório, os valores de erro relativamente pequenos corroboram a eficácia e acurácia desse método para estimar a produção de óleo e água em um reservatório de petróleo.

Com relação ao tempo, analisando a média da simulação de teste A e teste B, verifica-se que ele reduz em 72% o tempo da simulação, o *speedup* é aproximadamente 4. O tempo de execução médio do MRST foi de 2185,90 segundos, na simulação TPWL esse tempo foi de 604,92 segundos.

Em caso similar, no estudo de Cardoso (2009), a simulação de alta fidelidade no General Purpose Research Simulator (GPRS) requer cerca de 430 segundos de tempo de CPU, enquanto o POD-TPWL é executado consome apenas cerca de 0,85 segundos sem cálculo dos erros de balanço de materiais a cada passo do tempo. Isso representa um aumento de velocidade de execução de cerca de 500. Novamente, é importante ressaltar que com a redução de dimensionalidade pelo POD espera-se uma potencialização no valor do *speedup*.

6 CONCLUSÕES E TRABALHOS FUTUROS

Este trabalho desenvolveu um manual básico para a compreensão do *Matlab Reservoir Simulation Toolbox* (MRST) e estudou a implementação e utilização da técnica *Trajectory Piecewise Linearization* (TPWL). O TPWL concebe um modelo substituto físico através da linearização da equação de fluxo e torno de estados convergidos em uma simulação de treinamento, transformando a simulação de reservatórios em uma sequência de soluções de sistemas lineares. Os resultados indicaram que a técnica é satisfatória em acurácia e reduz o tempo empregado na simulação.

A seguir é apresentado um resumo dos principais resultados.

- Estudo da simulação de um modelo *black-oil* no *Matlab Reservoir Simulation Toolbox* (MRST), conhecendo as estruturas, objetos e funções requeridas para a utilização de *toolbox*.
- Estudo da programação orientada objeto e diferenciação automática, bem como, a aplicação desses no MRST. Dessa forma, desenvolveu-se a compreensão da família MRST AD-OO, nova estrutura de programação que vem sendo aperfeiçoada no MRST.
- Estudo interno da simulação pelo MRST AD-OO para o modelo *black-oil*. Especificamente, se fez necessário compreender a função *simulateScheduleAD,* e todas as classes, métodos e objetos utilizados para a simulação.
- Desenvolvimento da exportação dos mapas de estados e diversas derivadas dos resíduos, requeridas para a implementação do TPWL. Foram elaboradas duas metodologias. Na primeira, as matrizes jacobianas foram diretamente extraídas do simulador o que exigiu altarações em algumas funções do MRST; na segunda, buscou-se a não interfência da estrutura do simulador. Embora essa metodologia envolva um custo adicional para recalcular as matrizes, ela foi proposta em decorrência da constante alteração que vem sofrendo a estrutura do MRST AD-OO.
- Desenvolvida a implementação da técnica *Trajectory Piecewise Linearization (TPWL)*, com mapas e derivadas exportadas do MRST.
- Construção do modelo de reservatório baseado no corte do SPE10 e sua validação através do simulador comercial IMEX.

- Análise do desempenho e acurácia da técnica *Trajectory Piecewise Linearization (TPWL).* A técnica se mostrou satisfatória em acurácia, com erros da ordem de 3% e aproximadamente 3-4 vezes mais rápida que a simulação do MRST.

Esse trabalho é apenas o ponto de partida para diversos trabalhos futuros. A seguir estão listados alguns trabalhos de continuidade a essa pesquisa.

- Realizar a implementação do *Proper Orthogonal Decomposition* – POD para redução da dimensionalidade do problema, o que deverá potencializar a redução do tempo e trazer *speedups* da ordem de 100-500.
- Construir um algoritmo de otimização controlando as pressões de fundo dos poços com objetivo de maximizar o VPL, com a utilização da técnica TPWL/POD.
- Aplicar o critério de retreinamento (estudado em Fragroso (2015)), assim como, realizar a análise do erro TPWL/POD para aperfeiçoar o critério de reitreinamento.
- Realizar possibilidade de controlar os poços por vazão no modelo TPWL/POD.
- Considerar, além dos controles de poços, a variação de parâmetros geológicos na formulação do TPWL/POD (em He (2010) e He et al. (2013) encontra-se um estudo para transmissibilidades), implementando assim um modelo substituto a ser utilizado em estudos de análise de incertezas geológicas e ajuste de histórico.
- Desenvolver técnicas não intrusivas para modelos substitutos.

REFERÊNCIAS

ALHUTHALI, A. H.; DATTA-GUPTA, A.; YUEN, B.; e FONTANILLA, J. P. Field applications of waterflooding optimization via rate control with smart wells. **SPE Reservoir Simulation Symposium**, 2009.

AZIZ, K.; SETTARI, A. Fundamentals of Reservoir Simulation. **Elsevier Applied Science Publishers,** 1986.

BAO, K.; LIE, K. A.; MØYNER, O.; LIU, M. Fully implicit simulation of polymer flooding with MRST. **Computacional Geosciences**., [s.l.], vol. 21, n. 5-6, p. 1219-1244, 2017.

BEAR, J. **Dynamics of Fluids in Porous Media**. Dover, New York, 1972.

BP Statistical review of world energy. Disponível em: <https://www.bp.com/en/global/corporate/energy-economics/statistical-review-of-world-energy/chief-economist-analysis.html.>. Acesso em: 8 de janeiro de 2020.

BROWER D. R. ; JANSEN J. D. Dynamic optimization of waterflooding with smart wells using optimal control theory. **SPE Journal**, 2004.

CARLBERG, K.; C, BOU-MOSLEH; E FARHAT, C. Effcient nonlinear model reduction via a leastsquares petrov-galerkin projection and compressive tensor approximations. **International Journal for Numerical Methods in Engineering**, 2009.

CARDOSO, M. A.; DURLOFSKY, L. J. Linearized reduced-order models for subsurface flow simulation. **Journal of Computational Physics**, Vol. 229(No. 3):pp. 681–700, 2010.

CARDOSO, M. **Development and Application of Reduced-Order Modeling Procedures for Reservoir Simulation**, Dissertation (Doctor of Philosophy) – Stanford University, 2009.

CARVALHO, V. A.; TEIXEIRA, G. F. **Programação Orientada a Objetos: Curso técnico de informática**. Colatina: IFES, 2012.

CHATURANTABUT, S. e SORENSEN, D. C. **Discrete empirical interpolation for nonlinear model reduction**. In 48th IEEE Conference, 2009.

CHEN, Z.; HUAN, G.; MA, Y. **Computational Methods for Multiphase Flows in Porous Media.** Southern Methodist University Dallas, Texas. SIAM, 2006.

CHEN C.; WANG Y.; LI G., e A. C. Reynolds. Closed-loop reservoir management on the brugge test case. **Computational Geosciences**, 2010.

CHEN, Yan; OLIVER, Dean S. Ensemble-Based Closed-Loop Optimization Applied to Brugge Field, SPE 118926-PA, **SPE Reservoir Evaluation and Engineering**, Vol. 13, n. 1, p. 56-71, 2010.

CHEN, C.; LI, G.; e REYNOLDS, A. C. Robust constrained optimization of short and long-term net present value for closed-loop reservoir management. **SPE Journal**, 2012.

CHRISTIE, M. A.; BLUNT M. J., Tenth SPE comparative solution project: A comparison of upscaling techniques, **SPE Reservoir Evaluation & Engineering**, v.4, p. 308-317, 2001.

CMOST. **CMOST - User's Guide**. Computer Modeling Group Ltd., Calgary, Alberta, Canada, 2012.

CONN, A. R.; SCHEINBERG, K.; VICENTE, L. N. **Introduction to Derivative-Free Optimization**, SIAM, Philadelphia, USA, 2009.

DARMSTADT, Technische Universität Darmstadt**. Automatic Differentiation for Matlab (ADiMat)**, 2008. Disponível em: <http://www.adimat.de/>. Acesso em: 11 de novembro de 2019.

DEITEL, Harvey M.; DEITEL, Paul J. **Java: como programar**. 8. ed. São Paulo: Pearson/ Prentice-Hall, 2010.

EIA. **US. Energy Information Administration**. Disponível em: https://www.eia.gov/todayinenergy/detail.php?id=42415. Acesso em: 10 janeiro de 2020.

ERTEKIN, T.; ABOU-KASSEM, J. H.; KING, G. R. **Basic Applied Reservoir Simulation**. Richardson, TX, Society of Petroleum Engineers (SPE), 2001.

FINK, M. **Automatic Differentiation for Matlab.** MATLAB Central. 2007. Disponível em:< https://tinyurl.com/ycvp6n8a>. Acesso em: 20 de novembro de 2019.

FRAGOSO, M., HOROWITZ, B. e RODRIGUEs, J., Retraining Criteria for TPWL/POD Surrogate Based Waterflooding Optimization, **SPE Reservoir Simulation Symposium**, Houston, USA, 2015.

GILDIN E.; GHASEMI M.; PROTASOV, A.; EFENDIEV, Y. Nonlinear complexity reduction for fast simulation of flow in heterogeneous porous media. **SPE Reservoir Simulation Symposium**, 2013.

GOVINDJEE. **Object Oriented Programming and Classes in MATLAB.** University of California Berkeley, 2013.

HE. J. **Enhanced linearized reduced-order models for subsurface flow simulation**. Master's thesis, Stanford University, 2010.

HE, J.; SAETROM, J.; DURLOFSKY, L. J. Enhanced linearized reduced-order models for subsurface flow simulation. **Journal of Computational Physics**, Vol. 230(No. 23):pp. 8313–8341, 2011.

HE, J. **Reduced-Order Modeling For Oil-Water and Compositional Systems, with Applications to Data Assimilation and Production Optimization**, Stanford, USA, 2013.

HE, J.; DURLOFSKY, L. J. Reduced-order modelling for compositional simulation using trajectory piecewise linearization. **Reservoir Simulation Symposium,** The Woodlands, Texas, USA, 2014.

HE, J.; DURLOFSKY, L. J. Constraint reduction procedures for reduced-order subsurface flow models based on POD-TPWL. In**ternational Journal for Numerical Methods in Engineering**, 103:1–30, 2015.

HEWSON C.W. **Reduced-Order Modelling for Production Optimisation**. Department of Geoscience & Engineering Delft University of Technology. 2015

JANSEN, J. D.; DURLOFSKY, L.J. Use of reduced-order models in well control optimization. **Optimization and Engineering**. Springer US. Volume 18, Issue 1, pp 105–132, 2017.

JANSEN, J. D. Adjoint-based optimization of multiphase flow through porous media - a review. **Computational Fluids**, 2011.

KROGSTAD, S.; LIE, K–A; MØYNER, O.; NILSEN, H. M.; RAYNAUD, X., SKAFLESTAD B. MRST-AD – an Open-Source Framework for Rapid Prototyping and Evaluation of Reservoir Simulation Problems Stein SINTEF ICT, **Society of Petroleum Engineers**, 2015.

LIE, K. A. **An Introduction to Reservoir Simulation Using MATLAB/ User Guide for the MATLAB Reservoir Simulation Toolbox (MRST).** Oslo (Norway): SINTEF ICT, Departement of Applied Mathematics, 392 p., 2016.

MACHADO, M. F. J. **Aplicações de um modelo substituto de ordem reduzida a estudos de gerenciamento de reservatórios de petróleo**. Dissertação (Mestrado em Engenharia Civil), Universidade Federal de Pernambuco, Recife, 2014.

MARKOVINOVIC R. **System-theoretical model reduction for reservoir simulation and optimization.** PhD thesis, Technische Universiteit Delft, 2003.

MATHWORKS. **MATLAB Documentation**. Disponível em: <https://www.mathworks.com/help/matlab/index.html>. Acesso em: 10 de fevereiro de 2019.

MATHWORKS. **MATLAB - Object-Oriented Programming.** The MathWorks, Inc, 2015.

MRST. **The MATLAB Reservoir Simulation Toolbox**, version 2018b. Disponível em:<https://www.sintef.no/projectweb/mrst/. >. Acesso em: 1 de dezembro de 2018.

NEIDINGER, R. **Introduction to automatic differentiation and MATLAB objectoriented programming**. SIAM Review, 52(3), 545–563, 2010.

OLIVEIRA, Diego F. B., **Técnicas de Otimização da Produção para Reservatórios de Petróleo**, Dissertação de Mestrado, UFPE, Recife, 2006.

PEACEMAN, D.W. **Presentation of a horizontal well in numerical reservoir simulation**, SPE 21217, The 11th SPE Symposium on Reservoir Simulation, Ananheim, CA., 1991.

QUEIPO, N. V.; GOICOCHEA, L. V.; PINTOS P., Surrogate Modeling-Based Optimization of SAGD Process, **Journal of Petroleum Science & Engineering**, Vol. 35, p 83-93, 2002.

PET-TELE. **Tutorial de Programação Orientada a Objeto,** Versão: 2k9. Universidade Federal Fluminense, Niterói – RJ, 2009.

REGISTER, A. H. **A Guide to MATLAB Object-Oriented Programming**. Atlanta, Georgia, USA: SciThech Publishing Inc., 2007.

REWIENSK, M. J. i. **A trajectory piecewise-linear approach to model order reduction of nonlinear dynamical systems.** PhD thesis, Massachusets Institute of Technology, 2003.

REWIENSKI M.; WHITE J. **A trajectory piecewise-linear approach to model order reduction and fast simulation of nonlinear circuits and micromachined devices.** IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 22:155–170, 2003.

ROSA, A. J, **Engenharia de reservatórios de petróleo.** Rio de Janeiro, Ed. Interciência; Petrobrás, 2006.

SAVITCH, W. J. **C++ absoluto**. São Paulo, Brasil: Pearson Education do Brasil, 612 p., 2004.

SARMA, P.; CHEN, W. H.; DURLOFSKY L. J.; AZIZ K. Production optimization with adjoint models under nonlinear control-state path inequality constraints. **SPE Reservoir Evaluation Engineering**, 2008.

SOUZA, S. A.; AFONSO, S. M. B.; HOROWITZ, B., **Optimal Management of Oil Production Using the Particle Swarm Algorithm and Adaptive Surrogate Mode**l, XXXI Iberian Latin-American Congress on Computational Methods in Engineering, Buenos Aires, Argentina, 15-18 November, 2010.

SCHIUMBERGER. **ECLIPSE reservoir simulation software**. Reference Manual, 2014.

THOMAS, J.E.; TRIGGIA,A.A.; CORREIA,C.A.; FILHO,C.V; XAVIER,J.A.D.; MACHADO, J.C.V; FILHO, J.E.S.; PAULA, J..L.; ROSSI, N.C.M; PITOMBO, N.E.S; GOUVE, P.C.V.M; CARVALHO; R. S; BARRAGAN, R.V. **Fundamentos de Engenharia de Petróleo**. Editora Interciência. Rio de Janeiro, 2001.

TOMLAB **Optimization Inc. Matlab Automatic Differentiation (MAD).** [2008?] Disponível em:<http://matlabad.com/>. Acesso em: 20 de novembro de 2020.

VERMA, A. A**DMAT: Automatic differentiation in MATLAB using object oriented methods**. Henderson, M. E., Anderson, C. R., and Lyons, S. L., 1999.

WANG, C.; LI, G.; REYNOLDS, Albert C., Production Optimization in Closed-Loop Reservoir Management, SPE 109805, **SPE Journal**, Vol. 14, n. 3, p 506-523, 2009.

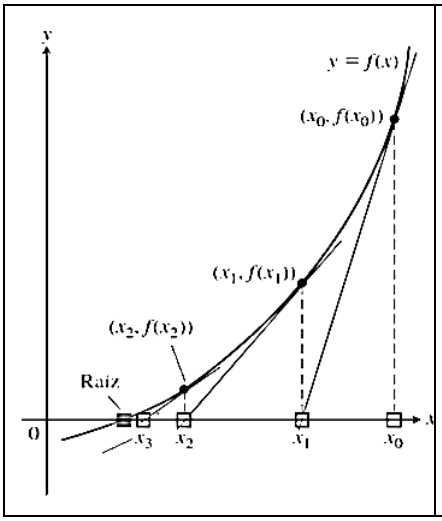
XIA, Y.; REYNOLDS, Albert C., An Optimization Algorithm Based on Combining Finite-Difference Approximations and Stochastic Gradients, SPE 163613, **SPE Reservoir Simulation Symposium**, The Woodlands, Texas, USA, 18-20 February, 2013.

ZHAO, Hui; CHEN, Chaohui ; DO, Sy; LI, Gaoming; REYNOLDS, Albert C., Maximization of a Dynamic Quadratic Interpolation Model for Production Optimization**, SPE Reservoir Simulation Symposium**, Woodlands, Texas, USA, 21-23 February, 2011.

APÊNDICE A – MÉTODO DE NEWTON-RAPHSON

Para uma equação unidimensional $f:\mathbb{R}\to\mathbb{R}$ utilizando polinômios de Taylor e expandindo a função em torno de x_0, obtém-se a Eq. (1). Mantendo os dois primeiros termos da série, a Eq. (2) representa a equação de uma reta que passa pelo ponto x_0 com inclinação $f'(x_0)$, ou seja, tangente à curva no ponto x_0. Na intersecção da curva com o eixo, por definição, $f(x)=0$. Logo, x é obtido pela Eq. (3). O valor obtido por x é utilizado como novo valor inicial x_0 até que obtenha uma melhor aproximação para raiz da função. Logo, a expressão do método de Newton é dada pela Eq. (4).

Figura A1 – Método de Newton-Raphson para Equação unidimensional



$$f(x)=f(x_0)+(x-x_0)f'(x_0)+\frac{(x-x_0)^2}{2!}f''(x_0)+\ldots \quad (1)$$

$$f(x)\approx f(x_0)+(x-x_0)f'(x_0) \quad (2)$$

$$x=x_0-\frac{f(x_0)}{f'(x_0)} \quad (3)$$

$$x_{i+1}=x_i-\frac{f(x_i)}{f'(x_i)} \quad (4)$$

Fonte: Autora (2020).

Para um sistema de equações com k equações e n variáveis, tem-se que:

$$\begin{aligned}
f_{1,i+1} &= f_{1,i}+(x_{1,i+1}-x_{1,i})\frac{\partial f_{1,i}}{\partial x_1}+(x_{2,i+1}-x_{2,i})\frac{\partial f_{1,i}}{\partial x_2}+\ldots+(x_{n,i+1}-x_{n,i})\frac{\partial f_{1,i}}{\partial x_n}=0\\
f_{2,i+1} &= f_{2,i}+(x_{1,i+1}-x_{1,i})\frac{\partial f_{2,i}}{\partial x_1}+(x_{2,i+1}-x_{2,i})\frac{\partial f_{2,i}}{\partial x_2}+\ldots+(x_{n,i+1}-x_{n,i})\frac{\partial f_{2,i}}{\partial x_n}=0 \quad (5)\\
&\text{M}\\
f_{k,i+1} &= f_{k,i}+(x_{1,i+1}-x_{1,i})\frac{\partial f_{k,i}}{\partial x_1}+(x_{2,i+1}-x_{2,i})\frac{\partial f_{k,i}}{\partial x_2}+\ldots+(x_{n,i+1}-x_{n,i})\frac{\partial f_{k,i}}{\partial x_n}=0
\end{aligned}$$

Reorganizando a Eq. (5) e isolando os termos com $x_{1,i+1}, x_{2,i+1}, \text{K}\ x_{n,i+1}$ no primeiro membro, teremos:

$$\begin{aligned}
&\frac{\partial f_{1,i}}{\partial x_1}x_{1,i+1}+\frac{\partial f_{1,i}}{\partial x_2}x_{2,i+1}+\ldots+\frac{\partial f_{1,i}}{\partial x_n}x_{n,i+1}=-f_{1,i}+\frac{\partial f_{1,i}}{\partial x_1}x_{1,i}+\frac{\partial f_{1,i}}{\partial x_2}x_{2,i}+\ldots+\frac{\partial f_{1,i}}{\partial x_n}x_{n,i}\\
&\frac{\partial f_{2,i}}{\partial x_1}x_{1,i+1}+\frac{\partial f_{2,i}}{\partial x_2}x_{2,i+1}+\ldots+\frac{\partial f_{2,i}}{\partial x_n}x_{n,i+1}=-f_{2,i}+\frac{\partial f_{2,i}}{\partial x_1}x_{1,i}+\frac{\partial f_{2,i}}{\partial x_2}x_{2,i}+\ldots+\frac{\partial f_{2,i}}{\partial x_n}x_{n,i}\\
&\qquad\qquad\qquad\qquad\text{M}\\
&\frac{\partial f_{k,i}}{\partial x_1}x_{1,i+1}+\frac{\partial f_{k,i}}{\partial x_2}x_{2,i+1}+\ldots+\frac{\partial f_{k,i}}{\partial x_n}x_{n,i+1}=-f_{k,i}+\frac{\partial f_{k,i}}{\partial x_1}x_{1,i}+\frac{\partial f_{k,i}}{\partial x_2}x_{2,i}+\ldots+\frac{\partial f_{k,i}}{\partial x_n}x_{n,i}
\end{aligned} \tag{6}$$

Reorganizando a Eq. (6):

$$\begin{bmatrix}
\frac{\partial f_{1,i}}{\partial x_1} & \frac{\partial f_{1,i}}{\partial x_2} & \text{L} & \frac{\partial f_{1,i}}{\partial x_n}\\
\frac{\partial f_{2,i}}{\partial x_1} & \frac{\partial f_{2,i}}{\partial x_2} & \text{L} & \frac{\partial f_{2,i}}{\partial x_n}\\
\text{L} & \text{L} & \text{L} & \text{L}\\
\frac{\partial f_{k,i}}{\partial x_1} & \frac{\partial f_{k,i}}{\partial x_2} & \text{L} & \frac{\partial f_{k,i}}{\partial x_n}
\end{bmatrix}
\begin{Bmatrix} x_{1,i+1}\\ x_{2,i+1}\\ \text{L}\\ x_{n,i+1}\end{Bmatrix}
=-\begin{Bmatrix} f_{1,i}\\ f_{2,i}\\ \text{L}\\ f_{k,i}\end{Bmatrix}+
\begin{bmatrix}
\frac{\partial f_{1,i}}{\partial x_1} & \frac{\partial f_{1,i}}{\partial x_2} & \text{L} & \frac{\partial f_{1,i}}{\partial x_n}\\
\frac{\partial f_{2,i}}{\partial x_1} & \frac{\partial f_{2,i}}{\partial x_2} & \text{L} & \frac{\partial f_{2,i}}{\partial x_n}\\
\text{L} & \text{L} & \text{L} & \text{L}\\
\frac{\partial f_{k,i}}{\partial x_1} & \frac{\partial f_{k,i}}{\partial x_2} & \text{L} & \frac{\partial f_{k,i}}{\partial x_n}
\end{bmatrix}
\begin{Bmatrix} x_{1,i}\\ x_{2,i}\\ \text{L}\\ x_{n,i}\end{Bmatrix} \tag{7}$$

Onde:

$\left[J\right]=\begin{bmatrix}
\frac{\partial f_{1,i}}{\partial x_1} & \frac{\partial f_{1,i}}{\partial x_2} & \text{L} & \frac{\partial f_{1,i}}{\partial x_n}\\
\frac{\partial f_{2,i}}{\partial x_1} & \frac{\partial f_{2,i}}{\partial x_2} & \text{L} & \frac{\partial f_{2,i}}{\partial x_n}\\
\text{L} & \text{L} & \text{L} & \text{L}\\
\frac{\partial f_{k,i}}{\partial x_1} & \frac{\partial f_{k,i}}{\partial x_2} & \text{L} & \frac{\partial f_{k,i}}{\partial x_n}
\end{bmatrix}$ é a matriz jacobiana das derivadas parciais;

Portanto, tem-se que:

$$\left[J\right]\left\{x_{i+1}\right\}=-f_i+\left[J\right]\left\{x_i\right\} \tag{8}$$

Pré-multiplicando ambos os membros pelo inverso da matriz Jacobiana:

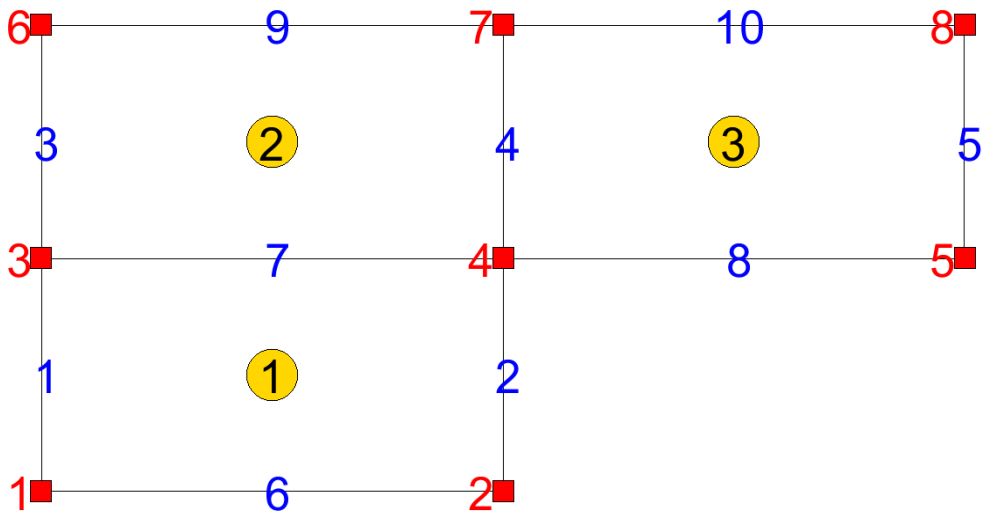
$$\left[J\right]^{-1}\left[J\right]\left\{x_{i+1}\right\}=-\left[J\right]^{-1}f_i+\left[J\right]^{-1}\left[J\right]\left\{x_i\right\} \tag{9}$$

$$\left\{x_{i+1}\right\}=\left\{x_i\right\}-\left[J\right]^{-1}f_i \tag{10}$$

A Eq. (10) representa a solução do sistema de equações não lineares pelo método de Newton.

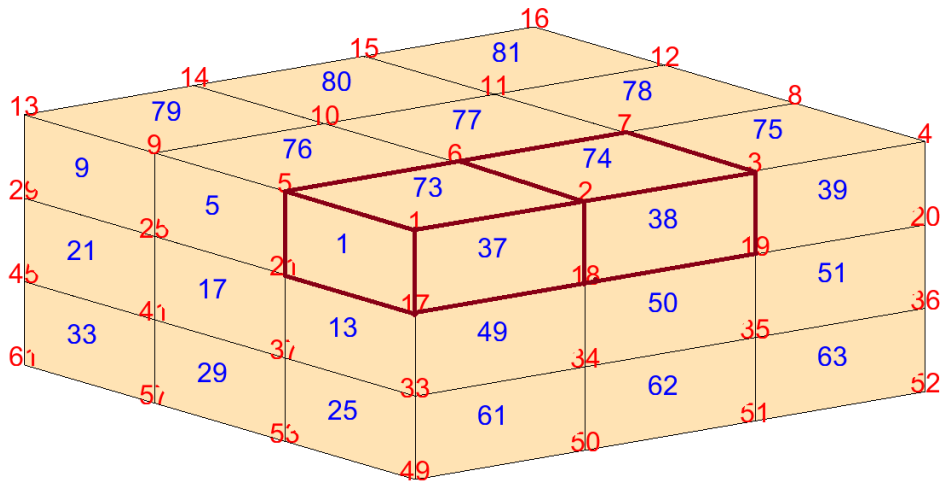
APÊNDICE B – CRIAÇÃO DAS FIGURAS

- **Figura 18 – Mapeamento** 2D



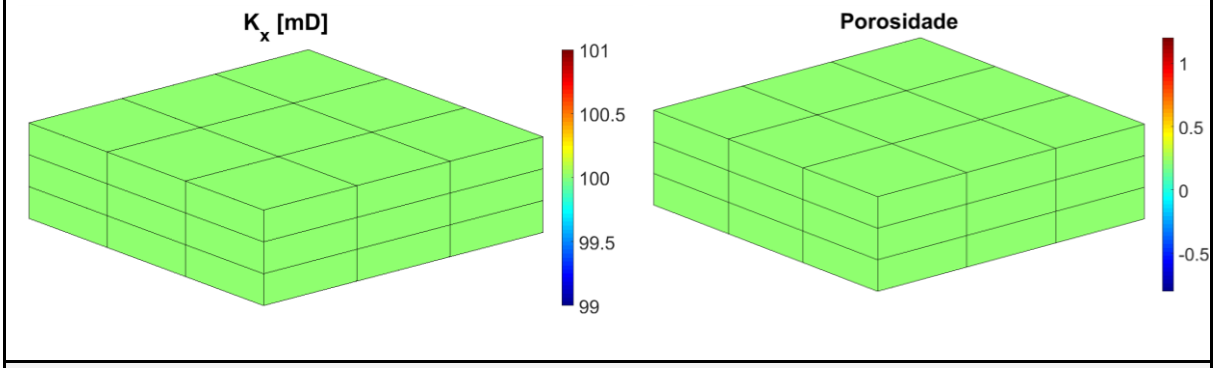

```
G = cartGrid([2,2]);
G = computeGeometry(G);
G = removeCells( G,2);
newplot;
plotGrid(G,'FaceColor','white'); axis off;hold on;

plot(G.cells.centroids(:,1),G.cells.centroids(:,2),'ok',...
 'MarkerSize',44,'color','black','MarkerFaceColor',[1 0.8398
0]);
text(G.cells.centroids(:,1)-0.03,G.cells.centroids(:,2),...
 num2str((1:G.cells.num)'),'FontSize',40,'color','black');
text(G.faces.centroids(:,1)-0.045, G.faces.centroids(:,2),...
 num2str((1:G.faces.num)'),'FontSize',40,'color','blue');
text(G.nodes.coords(:,1)-0.075,G.nodes.coords(:,2), ...
 num2str((1:G.nodes.num)'),'FontSize',40,'color','red');
plot(G.nodes.coords(:,1), G.nodes.coords(:,2),'sk',...
 'MarkerSize',24,'MarkerFaceColor','red');
hold off;
```

- **Figura 20 – Numeração do cubo MRST – Exemplo** Base



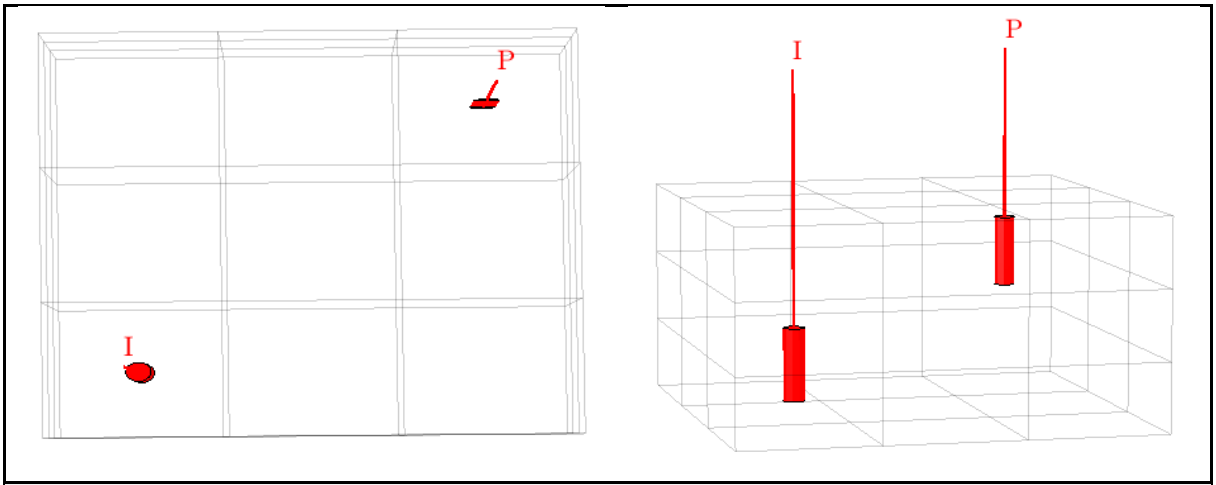

```
G = cartGrid([3,3,3],[1,1,1]);G = computeGeometry(G);
h = plotGrid(G,'FaceColor',[1 0.8906 0.7070]);
axis off;view(3);

text(G.nodes.coords(:,1)-0.03,G.nodes.coords(:,2),...
 G.nodes.coords(:,3)-0.05,num2str((1:G.nodes.num)'),...
 'FontSize',24,'color','red');
text(G.faces.centroids(:,1)-0.06, G.faces.centroids(:,2),...
 G.faces.centroids(:,3)-0.06,num2str((1:G.faces.num)'),...
 'FontSize',24,'color','blue');
text(G.cells.centroids(:,1)-0.04, G.cells.centroids(:,2),...

G.cells.centroids(:,3),num2str((1:G.cells.num)'),'FontSize',24);
```

- Figura 37 – Definição do micro reservatório

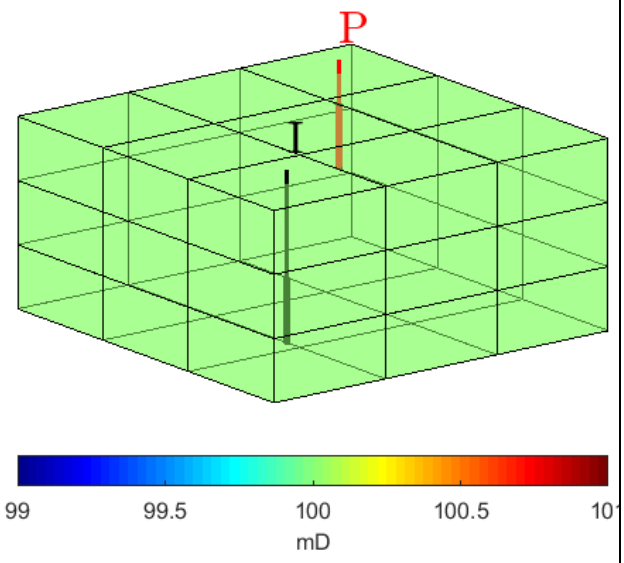


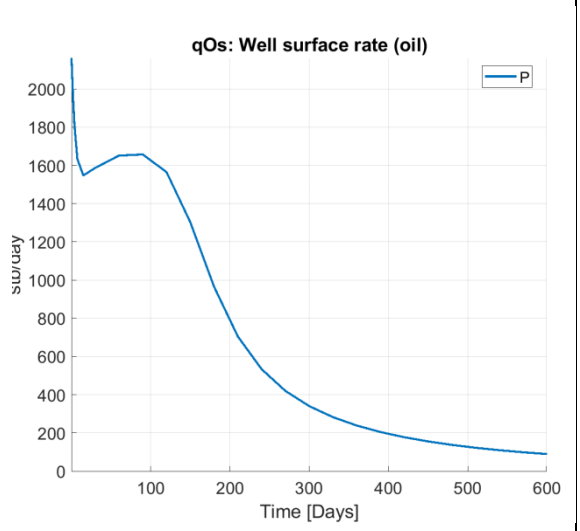

```
% Plot horizontal permeability

figure; clf
plotCellData(G, rock.perm(:, 1)./(milli*darcy))
colormap('jet');
set(gca,'FontSize',30)
view(50, 50), axis off
colorbar
title('K_x [mD]')

% Plot porosity
figure; clf
plotCellData(G, rock.poro)
colormap('jet');
view(50, 50), axis off
set(gca,'FontSize',30)
colorbar
grid on
title('Porosidade')
```

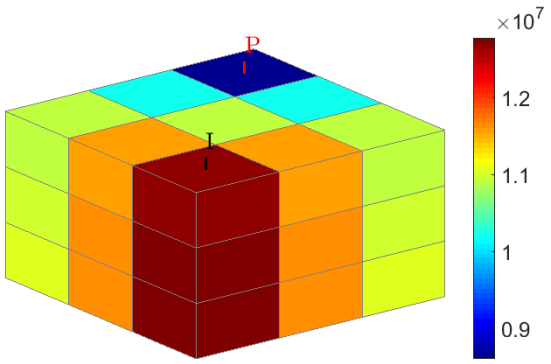
- **Figura 26 – Definição de poços**

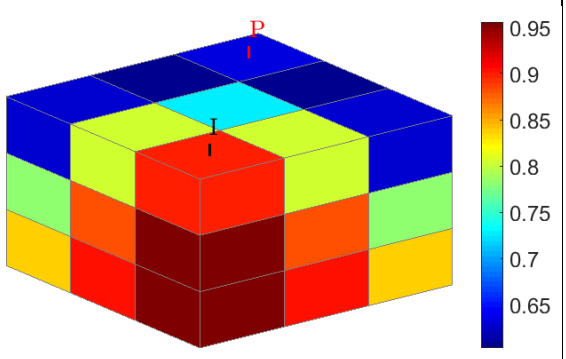



```
subplot(2,2,1), pos = get(gca,'Position'); clf
%Plotar Malha
plotGrid(G, 'FaceColor', 'none', 'EdgeAlpha',.1);

view(3), camproj perspective, axis tight off, camlight
headlight
%Plotar Poços
[ht, htxt, hs] = plotWell(G, W, 'radius', .3, 'height', 2);
set(htxt, 'FontSize', 16);
```










```
%% Plot Grid and Wells
figure
g=plotCellData(G, rock.perm(:, 1)/(milli*darcy),'FaceAlpha', 0.5);
view(3)
plotWell(G, W(1), 'radius',  .1, 'height',1,'fontsize',24,...
  'color', 'k');
plotWell(G, W(2), 'radius', .1, 'height',1,'fontsize',24, ...
  'color', 'r');
colormap('jet');
axis off
h = colorbar ('horiz');
h.FontSize = 12;
h.Label.String = 'mD';
set(gca,'FontSize',24)
 %% Plot the pressure and water saturation through the simulation
plotResults = 1;

if plotResults == 1
 inj = find(vertcat(W.sign) > 0);
 prod = find(vertcat(W.sign) < 0);
```

```
 df = get(0, 'DefaultFigurePosition');
 h = figure('Position', df.*[1, 1, 2.25, 1]);
 colormap('jet')
 fontSize = 20;
 edge = [0.5 0.5 0.5];
 for i = 1:numel(states)
  % Plot the pressure
  timestr = formatTimeRange(sum(schedule.step.val(1:i)));
  figure(h); clf
  subplot(1, 2, 1)
  plotCellData(G, states{i}.pressure/barsa, 'EdgeColor',edge);
  plotWell(G, W(inj), 'fontsize', fontSize, 'color', 'k')
  plotWell(G, W(prod), 'fontsize', fontSize, 'color', 'r')
  title(['Reservoir pressure after ', timestr]);
  view(-50, 70);
  axis tight off
  colorbar
  set(gca,'FontSize',fontSize)
% Plot water saturation
  subplot(1, 2, 2)
  plotCellData(G, states{i}.s(:, 1), 'EdgeColor',edge);
  plotWell(G, W(inj), 'fontsize', fontSize, 'color', 'k')
  plotWell(G, W(prod), 'fontsize', fontSize, 'color', 'r')
  title(['Water saturation after ', timestr]);
  view(-50, 70);
  axis tight off
  colorbar
  caxis([0, 1])
  drawnow
  set(gca,'FontSize',fontSize)
 end
end
%%  Plotting the results (interactive interface)
 if plotResults == 1
 plotWellSols(wellSols, cumsum(schedule.step.val), ...
  'datasetnames', 'Constant pressure')
 set(gca,'FontSize',fontSize)
 figure;
 plotToolbar(G, states,'EdgeColor',edge);
 view(3)
 axis off
 plotWell(G, W,'height', 1, 'fontsize', fontSize, , ...
    'color', 'k', 'color2', 'r');
 set(gca,'FontSize',fontSize)
 colormap('jet')
end
```

APÊNDICE C – CÓDIGOS AUXILIARES

- **Algoritmo 51**

Schedule randômico

```
function schedule
=controlSchedule(ncontrol,W,schedule,T,dt)

changeControl= T/ncontrol;                          %tempo de
sControl=changeControl;                             trocar o
                                                    controle
ts = cell(1,numel(ncontrol));

count =1;
a=1;
 for i=1:numel(dt)
        if sum(dt(1:i))>sControl
        ts{1,count}=dt(a:i-1);                      %Define o
        sControl=sControl+changeControl;            agrupamento
        count=count+1;                              de steps
        a=i;                                        para cada
        end                                         controle
 end

  ts{1,count}=dt(a:numel(dt));

  numCnt = numel(ts);
  [schedule.control(1:numCnt).W] = deal(W);
  schedule.step.control = rldecode((1:ncontrol)',   %Cria a
  cellfun(@numel, ts));                             estrutura
  schedule.step.val  = vertcat(ts{:});              schedule
end
```

- **Algoritmo 51**

Schedule randômico

```
function schedule = randSchedule
(a,b,W,schedule,ncontrol,numrand)


%Controle varia randomicamente nos POÇOS
PRODUTORES

s = rng(numrand);
 for j= 1:numel(W)
 r = (b-a).*rand(ncontrol,1) + a;
 if W(j).sign == -1
for i= 1:ncontrol
 schedule.control(i).W(j).val=r(i);
end
  end
 end
rng(s);

end
```

- **TPWLscript**

TPWL script

```
mrstModule add ad-core ad-blackoil ad-props spe10 mrst-gui ad-fi
optimization

 close all
 localread='C:\Users\mrst-2018b\examples\Model';
 cd(localread)
 localsave=strcat(localread,'\TREINAMENTO');

 %% TPWL-POD SIMULATION SCHEDULE
 % change schedule (true/false)?
 changeschedule = 1;

 if changeschedule == 1
  % Schedule from the new Simulation (HF)
  namesaved=['MODEL_SimResults_V1','.mat'];
  load(namesaved);
  statesHF=states;
  timeHF=tempo;
  G = model.G;
  clear report states W wellSols
 end

 %% TPWL VECTORS INITIALIZATION
 % Load stored states and derivatives from training simulation
 load('MODEL_TPWL_MATRICES_TRAINING.mat')

 fluid = TPWL.fluid;% Fluid properties
 states = TPWL.states; % Saved states (p, s, wellSols)
 W = TPWL.W;  % Well struct
 J = TPWL.J;  % Jacobian matrices
 A = TPWL.dA; % Accumulation terms derivative matrices
 Q = TPWL.dQ; % Well terms derivative matrices
 T = TPWL.dt; % Time steps

 nT = numel(schedule.step.val);% Number of timesteps
 nG = length(states{1,1}.pressure);  % Number of cells
 nW = length(W);% Number of wells

 ntTPWL=length(TPWL.dt);

 X_i = zeros(2*nG, nT);  % Saved states from Training Run
 X_n = zeros(2*nG, nT);  % States obtained from TPWL
 U_n = zeros(nW, nT); % Well controls applied at TPWL
 statesTPWL{nT,1} = struct();
 wellSolsTPWL{nT,1} = struct();

 % Putting the snapshots in the following format:
 % Each column represents the states saved in a time step.
 % The 'nG' first rows represent the PRESSURE in each cell
 % while 'nG' last rows represent the WATER SATURATION in each cell.
 X = zeros(2*nG,ntTPWL);
 U = zeros(nW, ntTPWL);
 for i = 1:ntTPWL
  X(:, i) = TPWL.X{i}(:,1);
  U(:, i) = cell2mat(TPWL.u{i}(:,1));
```

TPWL script

```
 end

 W_n = W;
 wc = vertcat(W(:).cells);
 injInx = find(vertcat(W(:).sign) == 1);
 prodInx = find(vertcat(W(:).sign) == -1);

 %% WELL CONTROLS INITIALIZATION

 for i = 1:nT
  inx = schedule.step.control(i);
  countinj = 0;
  countprod = 0;

  % Load control values for TPWL
  for j = 1:length(states{i}.wellSol)
% INJECTOR WELLS
if states{i}.wellSol(j).sign == 1
 countinj = countinj + 1;
 if i == 1
  W_n(j).val = U_n(j,i);
  W_n(j).bhpLimit = schedule.control(inx).W(j).lims;
 end
 if strcmp(states{i}.wellSol(j).type,'bhp') % BHP Control
  U_n(j,i) = schedule.control(inx).W(j).val;
  statesTPWL{i}.wellSol(j).type = 'bhp';
 else
  U_n(j,i) = schedule.control(inx).W(j).val;
  statesTPWL{i}.wellSol(j).type = 'wrat';
 end
 statesTPWL{i}.wellSol(j).mixs = [1,0];
 statesTPWL{i}.wellSol(j).name = states{i}.wellSol(j).name;
 statesTPWL{i}.wellSol(j).status = true;
 statesTPWL{i}.wellSol(j).val = U_n(j,i);
 statesTPWL{i}.wellSol(j).sign = states{i}.wellSol(j).sign;
 statesTPWL{i}.wellSol(j).cqs = ...
 zeros(length(W(j).cells),2);
 statesTPWL{i}.wellSol(j).cdp =zeros(length(W(j).cells),1);
 statesTPWL{i}.wellSol(j).cstatus = ...
 ones(length(W(j).cells),1);
 if (strcmp(statesTPWL{i}.wellSol(j).type,'bhp'))
  statesTPWL{i}.wellSol(j).bhp = U_n(j,i);
  statesTPWL{i}.wellSol(j).qTs = 0;
  statesTPWL{i}.wellSol(j).qWs = 0;
  statesTPWL{i}.wellSol(j).qOs = 0;
  statesTPWL{i}.wellSol(j).qGs =...
 states{i}.wellSol(j).qGs;
  statesTPWL{i}.wellSol(j).qs = states{i}.wellSol(j).qs;
 else
  statesTPWL{i}.wellSol(j).bhp = 0;
  statesTPWL{i}.wellSol(j).qTs = 0;
  statesTPWL{i}.wellSol(j).qWs = U_n(j,i);
  statesTPWL{i}.wellSol(j).qOs = 0;
  statesTPWL{i}.wellSol(j).qGs =...
 states{i}.wellSol(j).qGs;
  statesTPWL{i}.wellSol(j).qs = states{i}.wellSol(j).qs;
 end
```

TPWL script

```
 statesTPWL{i}.wellSol(j).mixs = states{i}.wellSol(j).mixs;
 statesTPWL{i}.wellSol(j).cstatus =...
 states{i}.wellSol(j).cstatus;
end

% PRODUCER WELLS
if states{i}.wellSol(j).sign == -1
 countprod = countprod + 1;
 if i == 1
  W_n(j).val = U_n(j,i);
  W_n(j).bhpLimit = schedule.control(inx).W(j).lims;
 end
 if strcmp(states{i}.wellSol(j).type,'bhp') % BHP Control
  U_n(j,i) = schedule.control(inx).W(j).val;
  statesTPWL{i}.wellSol(j).type = 'bhp';
 elseif strcmp(states{i}.wellSol(j).type,'lrat')%LiquidRate
  U_n(j,i) = schedule.control(inx).W(j).val;
  statesTPWL{i}.wellSol(j).type = 'lrat';
 end
 statesTPWL{i}.wellSol(j).mixs = [0,1];

 statesTPWL{i}.wellSol(j).name = states{i}.wellSol(j).name;
 statesTPWL{i}.wellSol(j).status = true;
 statesTPWL{i}.wellSol(j).val = U_n(j,i);
 statesTPWL{i}.wellSol(j).sign = states{i}.wellSol(j).sign;
 statesTPWL{i}.wellSol(j).cqs = ...
 zeros(length(W(j).cells),2);
 statesTPWL{i}.wellSol(j).cdp =...
  zeros(length(W(j).cells),1);
 statesTPWL{i}.wellSol(j).cstatus = ...
 ones(length(W(j).cells),1);
 if (strcmp(statesTPWL{i}.wellSol(j).type,'bhp'))
  statesTPWL{i}.wellSol(j).bhp = U_n(j,i);
  statesTPWL{i}.wellSol(j).qTs = 0;
  statesTPWL{i}.wellSol(j).qWs = 0;
  statesTPWL{i}.wellSol(j).qOs = 0;
  statesTPWL{i}.wellSol(j).qGs = ...
   states{i}.wellSol(j).qGs;
  statesTPWL{i}.wellSol(j).qs = states{i}.wellSol(j).qs;
 end

 statesTPWL{i}.wellSol(j).mixs = states{i}.wellSol(j).mixs;
 statesTPWL{i}.wellSol(j).cstatus = ...
 states{i}.wellSol(j).cstatus;
end
  end
 end

  %% TPWL PERFORMANCE
  tic;
  for i = 1:nT-1

 if i == 1
  X_n(:, i) = X(:,i);
  end
```

TPWL script

```
% Manuel Fragoso (2014)
% The distance from the TPWL state for each saved snapshot
% is calculated considering ONLY WATER SATURATIONS
d_t = zeros(ntTPWL,1);
for j = 1:ntTPWL
 d_t(j) = norm(abs(X(nG+1:2*nG,j) - X_n(nG+1:2*nG,i)));
end
% closest saved state index
[~,indSs] = min(d_t);
X_i(:, i) =  X(:, indSs);

if indSs == ntTPWL
 indSs = ntTPWL-1;
end

X_i(:, i+1) = X(:, indSs+1);
J = cell2mat(TPWL.J(indSs+1));
A = cell2mat(TPWL.dA(indSs+1));
Q = cell2mat(TPWL.dQ(indSs+1));

if i == 1
 X_n(:,i)= X_i(:, i);
end

X_n(:,i+1) = X_i(:,i+1) - J\(A*(X_n(:,i) - X_i(:,i)) ...
 + Q*(U_n(:,i+1) - U(:,i+1)));

disp(['Simulating TPWL ', num2str(i), '/', ...
 num2str(nT-1)]);

clear J A Q
if i == 1
 % TPWL pressure states
 statesTPWL{1}.pressure = X_n(1:nG,1);

 % TPWL water saturation states
 statesTPWL{1}.s(:,1) = X_n(nG+1:2*nG,1);

 % TPWL oil saturation states
 statesTPWL{1}.s(:,2) = 1 - X_n(nG+1:2*nG,1);

 % TPWL wellSols states
 [model,statesTPWL{i}] = ...
WellContributions(statesTPWL{i}, W,schedule,model);
 wellSolsTPWL{i} = statesTPWL{i}.wellSol;
end
% Assign values to the TPWL state structure
statesTPWL{i+1}.pressure = X_n(1:nG,i+1);
statesTPWL{i+1}.s(:,1) = X_n(nG+1:2*nG,i+1);
statesTPWL{i+1}.s(:,2) = 1 - X_n(nG+1:2*nG,i+1);
 end
 time = toc;
```

- **exportTPWL**

exportTPWL

```
% Rotina destinada a exportar as matrizes necessárias ao TPWL
clear all
local='C:\Users\mrst-2018b\examples\Model';
cd(local)

load('MODEL4_SimResults.mat')
if ~isfield(schedule.step.control(1),'src')
src = [];
bc=[];
[schedule.control.src] = deal(src);
[schedule.control.bc] = deal(bc);
end

dt=schedule.step.val;
problem=cell(numel(dt),1);
matrices = cell(numel(dt),1);

for i=1:numel(dt)
 if i~=1
  state0=states{i-1};
 else
  ctrl = schedule.control(schedule.step.control(1));
  [forces, fstruct] = model.getDrivingForces(ctrl);
  model = model.validateModel(fstruct);
  state0 = model.validateState(state0);
 end
 j = schedule.step.control(i);
 [problem{i},~,matrices{i,1}] = equationsOilWaterTPWL(state0, ...
states{i}, model, dt(i), schedule.control(j),'ResOnly',false, ...
'ResOnly',false,'iteration',report.Iterations(i));
 problem{i}=problem{i}.assembleSystem();

end

%% Store result at each step
n=2*model.G.cells.num;
for i = 1:numel(dt)
 % salva separadamente o termo referente a acumulacao (val e jac)
 % acumulacao referente a equacao da agua:
 A{1} = matrices{i}.wacum;
 % acumulacao referente a equacao do oleo:
 A{2} = matrices{i}.oacum;
 % concatena a equacao da agua e do oleo:
 iseq = cellfun(@(x) ~isempty(x), A);
 Acum {i,1}= vertcat(A{iseq});
 % salva a matriz com as derivadas em relacao a saturacao e pressao:
 AcumM{i,1} = [Acum{i,1}.jac{1,1},Acum{i,1}.jac{1,2}];
 % obtem acumulacao referente ao passo de tempo anterior
 if i==1
  rt (i) = 0;
  dAi1dxi{i,1}=sparse(n,n);
 else
  rt(i) = -dt(i-1)/dt(i);
  dAi1dxi{i,1}=rt(i).*AcumM{i-1,1};
 end
```

exportTPWL

```
 % matriz jacobiana
 % jacobiana referente a equacao da agua
 J{1} = matrices{i,1}.Ew;
 % jacobiana referente a equacao do oleo
 J{2} = matrices{i,1}.Eo;
 % concatena a equacao da agua e do oleo:
 iseq = cellfun(@(x) ~isempty(x), J);
 Jac{i,1}= vertcat(J{iseq});
 % salva a matriz com as derivadas em relacao a saturacao e pressao:
 JacM{i,1}=[Jac{i,1}.jac{1,1},Jac{i,1}.jac{1,2}];

 % salva separadamente o termo referente ao fluxo (val e jac)
 % não necessario ao TPWL colocada apenas para ficar complet0
 % fluxo referente a equacao da agua:
 F{1} = matrices{i,1}.wflux;
 % fluxo referente a equacao do oleo:
 F{2} = matrices{i,1}.oflux;
 % concatena a equacao da agua e do oleo:
 iseq = cellfun(@(x) ~isempty(x), F);
 Flow {i,1}= vertcat(F{iseq});
 % salva a matriz com as derivadas em relacao a saturacao e pressao:
 FlowM{i,1}=[Flow{i,1}.jac{1,1},Flow{i,1}.jac{1,2}];

 % matriz dos poços
 % pocos referente a equacao da agua
 Qu{1} = matrices{i,1}.wellW;
 % pocos referente a equacao do oleo
 Qu{2} = matrices{i,1}.wellO;
 % concatena a equacao da agua e do oleo:
 iseq = cellfun(@(x) ~isempty(x), Qu);
 dQu{i,1}= vertcat(Qu{iseq});
 % salva a matriz com as derivadas em relacao a qWs, qOs eBHP:
 dQuM{i,1}=[dQu{i,1}.jac{1,3},dQu{i,1}.jac{1,4},dQu{i,1}.jac{1,5}];
 % salva a matriz com as derivadas em relacao a BHP:
 dQuB{i,1}=[dQu{i,1}.jac{1,5}];
 % salva a matriz com as derivadas em relacao a p, s, qWs, qOs eBHP:
 dQuT{i,1}=[dQu{i,1}.jac{1,1},dQu{i,1}.jac{1,2},...
dQu{i,1}.jac{1,3},dQu{i,1}.jac{1,4},dQu{i,1}.jac{1,5}];

 % estados convergidos
 % pressao
 XI{1} = states{i,1}.pressure;
 % saturacao
 XI{2} = states{i,1}.s(:,1);
 % concatenar pressao e saturacao
 iseq = cellfun(@(x) ~isempty(x), XI);
 XIn{i,1}= vertcat(XI{iseq});

 % controles dos pocos
 u{i,1} = ...
 num2cell([wellSols{i,1}.qWs,wellSols{i,1}.qOs,wellSols{i,1}.bhp]');
 % controle dos pocos - somente bhp
 uB{i,1} = num2cell([wellSols{i,1}.bhp]');
 % Estado convergido - vetor completo
 Xtotal{i,1}=vertcat(XIn{i,1},cell2mat(u{i,1}));

end
```

exportTPWL

```
%% Save structs to TPWL

TPWL = [];
TPWL.dA = dAi1dxi;
TPWL.dQ = dQuB;
TPWL.X =XIn;
TPWL.J = JacM;
TPWL.u = uB;
TPWL.dt = dt;
TPWL.schedule = schedule;
TPWL.states = states;
TPWL.wellSols = wellSols;
TPWL.fluid = model.fluid;
TPWL.W = schedule.control(1).W;

save('MODEL_TPWL_MATRICES _TRAINING_V1.mat','TPWL')
```