DESENVOLVIMENTO DE UMA INTERFACE GRÁFICA PARA UMA SOLUÇÃO DE IMAGEAMENTO SÍSMICO DE INVERSÃO FWI

Rodolfo Machado Brandão Costa

Projeto de Graduação apresentado ao Curso de Engenharia de Computação e Informação da Escola Politécnica, Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Engenheiro.

Orientador: Jorge Lopes de Souza Leão

Rio de Janeiro

Agosto de 2018

DESENVOLVIMENTO DE UMA INTERFACE GRÁFICA PARA UMA SOLUÇÃO DE IMAGEAMENTO SÍSMICO DE INVERSÃO FWI

Rodolfo Machado Brandão Costa

PROJETO DE GRADUAÇÃO SUBMETIDO AO CORPO DOCENTE DO CURSO DE ENGENHARIA DE COMPUTAÇÃO E INFORMAÇÃO DA ESCOLA POLITÉCNICA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE ENGENHEIRO DE COMPUTAÇÃO E INFORMAÇÃO

Autor:

Rodolfo Machado Brandão Costa

Rodolfo Machado Brandão Costa

Orientador:

Prof. Jorge Lopes de Souza Leão, Dr. Ing.

Examinador:

Prof. Heraldo Luís Silveira de Almeida, D. Sc.

Examinador:

Prof. Flávio Luis de Mello, D. Sc.

Rio de Janeiro – RJ, Brasil

Agosto de 2018

Costa, Rodolfo Machado Brandão

Desenvolvimento de uma Interface Gráfica para uma solução de Imageamento Sísmico de Inversão FWI / Rodolfo Machado Brandão Costa. – Rio de Janeiro: UFRJ/ Escola Politécnica, 2018.

XIII, 62 p.: il.; 29,7 cm.

Orientador: Jorge Lopes de Souza Leão

Projeto de Graduação – UFRJ/ Escola Politécnica/ Curso de Engenharia de Computação e Informação, 2018.

Referências Bibliográficas: p. 61-62.

1. Interface Gráfica 2. Imageamento Sísmico 3. Inversão FWI I. LEAO, J. L. S.. II. Universidade Federal do Rio de Janeiro, Escola Politécnica, Curso de Engenharia de Computação e Informação. III. Título.

AGRADECIMENTOS

Eu gostaria de agradecer aos meus pais Iracema e Antônio por todo apoio, carinho, inspiração, sabedoria, dedicação e amor que me deram ao longo da minha vida para que eu pudesse realizar esse sonho de poder me tornar Engenheiro de Computação.

Agradeço a minha irmã, Renata, por toda força, confiança e amor que me dá nos mais variados momentos de minha vida, e por me mostrar que com garra podemos conquistar tudo aquilo que almejamos.

Agradeço a minha família, ao Vinicius e sua família por todo carinho e amor que me dão incondicionalmente.

Agradeço a meus amigos Lygia Marina, Luciano Leite, Luiz Henrique Pinho, Thiago Vasconcelos e Pedro de Vasconcellos por toda colaboração, carinho e paciência, ajudando-me a enfrentar obstáculos com muito otimismo e transformarem a minha graduação em um momento inesquecível.

Agradeço a todo o corpo da PETREC por ser uma equipe que sempre busca se reinventar e criar experiências inovadoras, sendo fonte de inspiração para desenvolvimento de projetos desafiadores. Em especial agradeço Lívia Fernandes e Carlos Henrique pelo material e toda ajuda que me deram ao longo desse projeto.

Agradeço aos professores da UFRJ, por sempre fazerem do aluno alguém capaz de criar soluções com nível de excelência altíssimo, em especial ao meu orientador Jorge Lopes de Souza Leão por toda paciência e atenciosidade que teve comigo durante a construção desse projeto.

Por fim, agradeço a Nintendo, que por meio dos seus jogos criativos e inovadores, motivou-me a escolher esse curso e admirar de computação gráfica e interfaces gráficas.

Resumo do Projeto de Graduação apresentado à Escola Politécnica/ UFRJ como parte dos requisitos necessários para a obtenção do grau de Engenheiro de Computação e Informação.

Desenvolvimento de uma Interface Gráfica para uma solução de Imageamento Sísmico de Inversão FWI

Rodolfo Machado Brandão Costa

Agosto/2018

Orientador: Jorge Lopes de Souza Leão

Curso: Engenharia de Computação e Informação

O homem, em sua história, sempre buscou desenvolver ferramentas que o ajudassem a explorar o ambiente no qual habita. Diante de um resultado satisfatório com o uso, era esperado que esses instrumentos fizessem parte do seu cotidiano, porém, para que isso fosse possível, também era necessário que fossem amigáveis e bem adaptados às suas limitações sensoriais. Na atual conjuntura, a computação é a ferramenta mais importante e utilizada nos mais diversos campos da ciência e tecnologia, feito viabilizado por *softwares*, com interfaces gráficas, desenvolvidos para serem usáveis dentro do contexto em que forem aplicados. Tal cenário também pode ser observado na indústria de Óleo e Gás com *softwares* que propõem facilitar o processo de Imageamento Sísmico, mas que se encontram concentrados nas grandes operadoras de petróleo e são muito caros. Este trabalho aborda o desenvolvimento de uma interface gráfica para Inversão FWI por meio do *software* Qt que possa virar um produto de alto valor agregado. Nela o usuário informa valores dos dados de entrada necessários para executar a Inversão FWI e pode visualizar os resultados em tempo real, permitindo a continuidade ou interrupção do processo em função dos resultados parciais.

Palavras-chave: Interface Gráfica, Imageamento Sísmico, Inversão FWI, Qt, Óleo e Gás, Produto de Alto Valor Agregado, *Software*, Experiência do Usuário.

Abstract of Undergraduate Project presented to POLI/UFRJ as a partial fulfillment of the requirements for the degree of Computer and Information Engineer.

Development of a Graphical User Interface for a FWI Seismic Imaging solution

Rodolfo Machado Brandão Costa

Agosto/2018

Advisor: Jorge Lopes de Souza Leão

Course: Computer and Information Engineering

The man, in his story, always tried to develop tools that could help him explore the world in which he lives. Having a satisfactory result with their use, it was supposed that these tools would become part of his daily live, but in order for it to be possible, it was necessary that they were friendly and well adapted to his sensory limitations. In the current conjuncture, computing is the most important and used tool in the most diverse science and technology fields, a feat that was made possible due to softwares with graphical user interface that were developed to be usable inside the context in which they were applied. This scenery can be observed too in the Oil and Gas industry with softwares that propose to ease the Seismic Imaging process, but they are concentrated in the hands of big oil operators and are expensive. This project approaches the development of a graphical user interface for FWI by using Qt software, which can be turned into a high added value product. In this interface, the user inputs data values needed to run FWI and he can view the results in real time, allowing him to continue or abort the execution of the process based on the partial results.

Key-words: Graphical User Interface, Seismic Imaging, FWI, Qt, Oil and Gas, High Added Value Product, Software, User Experience.

Sumário

Capítulo 1
Introdução

1.1 Tema

O computador é um elemento que tem ganhado cada vez mais espaço em nosso cotidiano. Desde o ambiente de trabalho até em casa, ele é utilizado tanto para resolução de tarefas como para lazer. Em todos esses momentos, essa interação homem máquina é possível graças à existência das interfaces gráficas, que transformam as informações armazenadas sob a forma de linguagens de máquina em informações passíveis de interpretação humana.

Em função desse contexto, a demanda pelo desenvolvimento de aplicações baseadas em interface gráfica vem aumentando. Ao mesmo tempo, conhecer as necessidades do usuário final tornou-se um pré-requisito fundamental, pois são elas que delimitam o processo de desenvolvimento e as funcionalidades finais do produto. Junto a isso, o aumento na demanda por aplicações gráficas estimula, paralelamente, o crescimento e investimento em diversas áreas da ciência e do conhecimento humano.

A área de sísmica, dentro de um contexto tecnológico, é fortemente caracterizada por softwares científicos, que utilizam muitos recursos computacionais e servem, basicamente, para fazer cálculos, não sendo projetados com um usuário final em mente. Dessa forma, tornam-se aplicações cujo uso é limitado apenas a usuários altamente especializados, reduzindo drasticamente a quantidade de possíveis utilizadores.

Este trabalho se encaixa nesse âmbito, abordando o desenvolvimento de uma interface gráfica como uma forma de transformar um software de sísmica existente em um produto comercial que possa ser utilizado por a maior quantidade de usuários possível.

1.2 Delimitação

O objeto de estudo é o desenvolvimento de um *software* para *desktop Linux* dentro de uma *startup*, abrangendo características do processo de desenvolvimento do mesmo. Esta aplicação foi toda construída com base nos dados de entrada e saída de um código pronto de *FWI* proprietário dessa *startup*.

1.3 Justificativa

Atualmente há uma grande demanda e foco em projetos que tem como objetivo principal a satisfação do usuário. A preocupação com a experiência do usuário vem ganhando cada vez mais destaque dado que serve como uma ferramenta poderosa de *marketing* e valorização de produto, elementos que influenciam diretamente na expansão e maior reconhecimento de uma marca e de uma empresa.

O mercado de Petróleo e Gás é um dos segmentos mais lucrativos no Brasil e extremamente importante na economia mundial que se encontra concentrado nas mãos das maiores operadoras de petróleo nacionais e internacionais. Apesar de estas operadoras oferecerem diversos serviços relacionados a Exploração e Produção, muitos deles são caríssimos e de baixa acessibilidade para pequenas e médias empresas desse segmento. Por conta disso, existe uma carência de serviços que sejam de baixo custo e que garanta resultados precisos e de alta qualidade, configurando um nicho de mercado que, se explorado, pode ser altamente lucrativo e ao mesmo tempo inovador para a indústria de petróleo como um todo por facilitar a entrada de novos empreendimentos para a área.

O desenvolvimento desse trabalho está intimamente ligado à exploração desse nicho de mercado. Ser um dos primeiros a desenvolver uma tecnologia de origem nacional significa poder conquistar uma grande fatia desse mercado, garantindo um forte reconhecimento da empresa por muitos outros empreendimentos da área, o que representa um ganho tanto para a PETREC quanto para a indústria de Petróleo e Gás.

1.4 Objetivos

O objetivo deste trabalho é construir um produto que seja utilizado como um "cartão de visitas" da PETREC para atrair investimentos e novos clientes interessados em serviços de *FWI* e também de imageamento sísmico de qualidade e custo competitivo. Para isso, uma interface gráfica se mostrou uma excelente opção, pois mantém o aspecto de serviço do imageamento sísmico ao ser usada internamente pela empresa e, ao mesmo tempo, serve como um produto para clientes, servindo também como uma identidade visual para a empresa.

Para poder criar essa situação de uso interno e externo da interface, foi necessário um desenvolvimento rápido no qual a prioridade era obter um protótipo que fosse o mais fiel possível a versão final que a empresa planejava. Assim, sempre que surgisse alguma chance de fazer uma demonstração, uma versão funcional da interface estaria pronta para ser usada e demonstrada para potenciais clientes.

1.5 Metodologia

O trabalho proposto é uma aplicação *desktop* que se comunica com o código *FWI* existente e que fornece informações de maneira visual para o usuário. A versão deste trabalho deveria ser capaz de executar as seguintes tarefas:

- Construir um diretório de pastas necessário para a correta execução do código *FWI*;
- Construir o arquivo texto com todos os dados de entrada ordenados e formatados para que pudesse ser lido pelo código *FWI*;
- Definir os limites de valor para cada parâmetro de acordo com o que o código *FWI* poderia aceitar;
- Ler o arquivo de saída do código *FWI* demonstrando o passo-a-passo da execução do algoritmo e seu tempo de execução;
- Ler as imagens geradas pelo código *FWI* contendo os modelos de velocidade resultantes de cada etapa do algoritmo;
- Ler arquivo contendo os valores indicadores de convergência do algoritmo;

- Dispor as imagens dos modelos de velocidades na interface de forma a viabilizar a comparação direta entre elas em tempo de execução;
- Dispor o passo-a-passo da execução do algoritmo e seu tempo de execução na interface em tempo de execução;
- Dispor os valores indicadores de convergência do algoritmo sob a forma de um gráfico de pontos em tempo de execução;
- Executar o código *FWI* a partir de um comando dentro da interface.

Por fim, a interface deveria apresentar todas as informações de forma fácil, intuitiva e possuir uma aparência similar a outros programas de sísmica presentes no mercado.

1.6 Descrição

Este texto foi dividido nos seguintes capítulos:

- Capítulo 2: Aborda conceitos básicos de Engenharia de *Software* que embasaram a construção deste trabalho;
- Capítulo 3: Aborda conceitos básicos de Interface Homem Máquina que influenciaram no processo de desenvolvimento da aplicação;
- Capítulo 4: Apresentação resumida sobre *FWI* e explicação comparativa das vantagens e melhorias trazidas pelo acoplamento da interface gráfica ao *software* de *FWI*;
- Capítulo 5: Apresentação e explicação das ferramentas utilizadas para implementação deste trabalho;
- Capítulo 6: Conclui o trabalho e ilustra futuras melhorias que podem ser aplicadas.

Capítulo 2

Modelagem do *Software*

O ser humano, no seu dia-a-dia, está sempre buscando soluções para os problemas aos quais é exposto. A partir das diversas ferramentas criadas por ele mesmo ou fornecidas pela própria natureza, ele estuda a melhor solução possível, avaliando o custo-benefício dos recursos a serem usados versus o impacto do resultado sobre ele e o ambiente ao seu redor. No contexto de desenvolvimento de *software*, existem diversas regras criadas com o intuito de padronizar e viabilizar o processo de modelagem e desenvolvimento de um sistema, que são descritas e explicadas pela Engenharia de *Software*.

Um problema possui diversas características responsáveis por determinar a sua ocorrência e seu comportamento. Por isso, faz-se necessária uma análise cautelosa para entender primeiro a sua natureza, com o objetivo de definir qual será a metodologia empregada para construir a solução. Para o caso de um problema em que a computação é utilizada, uma variedade de técnicas, ferramentas, procedimentos e paradigmas são empregados para obter-se uma abordagem eficiente e produtiva que gere soluções efetivas. Para facilitar a compreensão ao longo da leitura deste trabalho, é importante saber as definições de alguns termos dentro da ótica da Engenharia de *Software*: (PFLEEGER, 2004, p. 3)

- **Técnica** é "um procedimento formal para produzir algum resultado.".
- **Ferramenta** é "um instrumento ou sistema automatizado utilizado para realizar uma tarefa da melhor maneira. Essa 'melhor maneira' pode significar que a ferramenta nos torna mais precisos, eficientes e produtivos ou que melhora a qualidade do produto resultante.".
- **Procedimento** é "como uma receita: a combinação de ferramentas e técnicas que, em harmonia, produzem um resultado específico.".
- **Paradigma** é "como um estilo de cozinhar; ele representa uma abordagem ou filosofia em particular para a construção do software".

Outro fator muito influente na construção e planejamento da solução de um problema é a complexidade do mesmo. Quanto maior a complexidade, mais difícil será a solução, demandando uma melhor organização e mapeamento dos processos e recursos necessários para que se possa atingir a concretização do produto. Trazendo

essa problematização para a esfera da codificação deste trabalho, utilizam-se algumas técnicas, ferramentas, procedimentos e paradigmas que tornam o processo de escrita do código mais rápido e seguro.

2.1 Orientação a Objetos

Orientação a Objetos é um paradigma amplamente utilizado no desenvolvimento de *software*. Sua abordagem se baseia na divisão dos problemas e soluções do *software* em diversas unidades distintas denominadas **Objetos**. Esses objetos podem se relacionar e interagir entre si, possuindo comportamentos específicos, definindo, assim, a estrutura e o comportamento dos dados do *software* (PFLEEGER, 2004, p. 210). Esse paradigma se utiliza de vários conceitos, porém apenas alguns deles serão definidos abaixo em função do seu grau de importância para a construção deste trabalho. São eles: (PFLEEGER, 2004, p. 211-214)

- **Objeto:** Unidade que possui estados e comportamentos predefinidos, capaz de se comunicar e relacionar com outros Objetos.
- **Classe:** Representação de um conjunto de Objetos que possuem comportamentos e atributos semelhantes.
- **Instância:** Objeto representativo de uma Classe, que possui seus próprios valores de atributos e compartilha os mesmos comportamentos das outras Instâncias da mesma Classe.
- **Atributo:** Característica de uma Classe. Uma Classe pode possuir um ou mais atributos, que são responsáveis por definir a sua estrutura de dados.
- **Método:** Operação implementada de forma específica para uma Classe. Uma Classe pode possuir um ou mais métodos.
- **Comportamento:** Transformação ou ação que um objeto pode realizar ou ser levado a realizar.
- **Mensagem:** Invocação de um dos métodos de um Objeto (Fonte: http://www.dca.ufrn.br/~anderson/FTP/dca0120/P2_Aula1.pdf, acessado em 22/01/2017 às 16:00).
- **Herança (Generalização):** Vínculo entre classes, em que uma Subclasse pode reaproveitar os atributos e métodos da Superclasse, criando características comuns entre elas e, ao mesmo tempo, a Subclasse pode possuir atributos e métodos exclusivos, ou seja, características específicas dela.

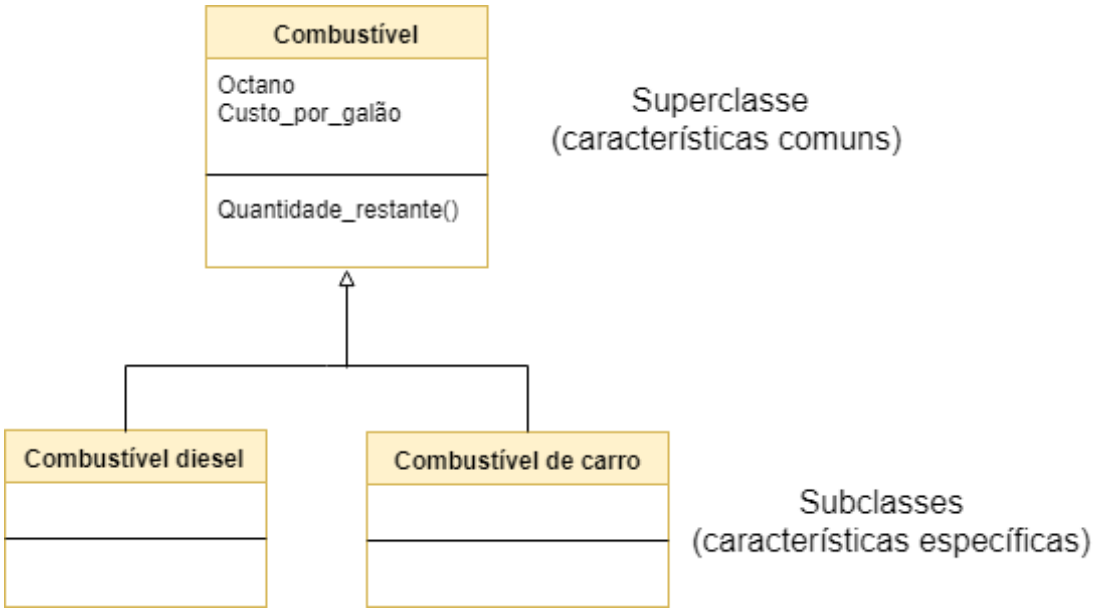


Figura 2-1: Exemplo de Herança demonstrando Superclasse e Subclasses

Fonte: Adaptado de (PFLEEGER, 2004, p.213)

- **Composição:** Vínculo entre classes com relação Todo-Parte, em que o Objeto-Parte só pode existir se o Objeto-Todo existir. No exemplo abaixo, um pedido pode possuir um ou mais itens pedidos, porém não é coerente existir um ou mais itens pedidos se não há um pedido (PFLEEGER, 2004, p. 224).

Figura 2-2: Exemplo de Composição

Fonte: Adaptado de (PFLEEGER, 2004, p.224)

- **Agregação:** Vínculo entre classes com relação Todo-Parte, em que o Objeto-Parte pode existir independente se o Objeto-Todo existir ou não. No exemplo abaixo, forma-se um time a partir de atletas, porém é coerente existir atletas, independente da existência de um time.

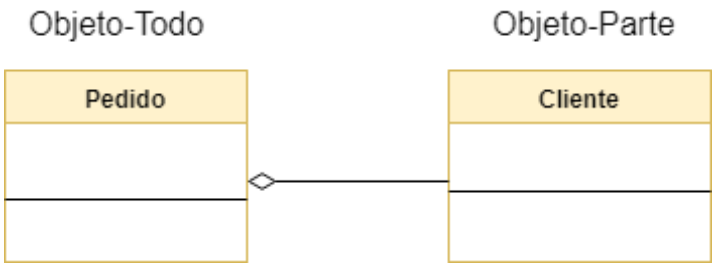


Figura 2-3: Exemplo de Agregação

Fonte: Adaptado de (PFLEEGER, 2004, p.224)

- **Associação:** Vínculo entre classes, sem relação Todo-Parte, que indica quando uma classe está conectada a outra classe e que a conexão deve ser mantida temporariamente. No exemplo abaixo, há uma associação entre Vendedor e Pedido, em que um vendedor pode ter um ou mais pedidos e, até serem concluídos, os pedidos devem ser vinculados ao vendedor (PFLEEGER, 2004, p. 224).

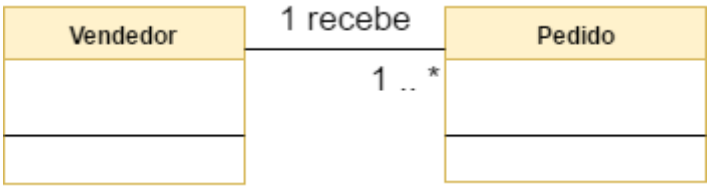


Figura 2-4: Exemplo de Associação

Fonte: Adaptado de (PFLEEGER, 2004, p. 224)

- **Encapsulamento:** Mecanismo de proteção de um Objeto em que os valores de seus atributos só podem ser alterados por meio de seus próprios métodos (PFLEEGER, 2004, p. 130).
- **Abstração:** Mecanismo de simplificar a representação de um contexto qualquer por meio da utilização dos aspectos mais relevantes em detrimento dos menos relevantes (Fonte: http://www.dca.ufrn.br/~anderson/FTP/dca0120/P2_Aula1.pdf, acessado em 22/01/2017 às 16:00).
- **Polimorfismo:** Mecanismo em que diferentes classes ou subclasses possuem uma mesma assinatura de método representando um mesmo comportamento exibido de diferentes formas (PFLEEGER, 2004, p. 213).

Utilizando toda essa gama de definições, abrem-se muitos caminhos para se desenvolver a solução computacional. Assim, o planejamento da implementação do projeto se torna uma etapa fundamental e responsável por otimizar o processo de implementação como um todo, tanto em questão de tempo quanto em questão de organização e esforço necessário para a conclusão do projeto. Com a possibilidade de

divisão do problema em partes menores, pode-se definir de que forma serão os relacionamentos entre os Objetos para que se projete a solução mais efetiva, aquela que apresenta a funcionalidade que mais se aproxima do resultado esperado.

A implementação deste trabalho foi realizada utilizando o Qt, um framework para desenvolvimento multiplataforma de interfaces gráficas em C++ (BLANCHETTE E SUMMERFIELD, 2008, p. 8). Toda a estrutura e ambiente de desenvolvimento dessa ferramenta se baseia nessas definições de Orientação a Objetos. Portanto, para que um desenvolvedor que venha a utilizar o Qt possa ter um aproveitamento máximo de todos os recursos que o programa oferece, é importante que ele compreenda e consiga aplicar esses conceitos no código que for implementar, sempre atento para a manutenibilidade do que produz. Isso estará melhor descrito no capítulo 5 deste trabalho.

No desenvolvimento, a delimitação da estrutura de dados e da hierarquia das Classes são tarefas importantes para a estruturação do código, característica muito influente no grau de escalabilidade e manutenibilidade de um *software*. Esse grau varia de acordo com o projeto a ser realizado e permite determinar se a implementação a ser executada atenderá todos os requisitos e recursos necessários para o projeto. Essa avaliação também é importante para prever e evitar possíveis retrabalhos ao longo do desenvolvimento do *software*, tornando mais seguro o processo como um todo. No caso de um *software* comercial, evitar retrabalho significa entregar o produto dentro do prazo combinado com o cliente e com um custo mínimo, situação ideal tanto para o responsável pelo desenvolvimento quanto para o cliente.

2.2 Orientação a Eventos

Também conhecida como Programação Dirigida a Evento, a Orientação a Eventos é um paradigma em que o fluxo de execução do *software* é definido por meio de ações externas, denominadas **eventos**. Em computação, um **evento** é a resposta para uma dada ação, que pode ser executada por um *software* externo, uma pessoa ou sensores. A ocorrência desses eventos é detectada por uma rotina de monitoramento capaz de invocar um ou mais códigos especializados para responder ao evento ocorrido. Existem diversos tipos de eventos e eles variam em função do dispositivo ou ambiente no qual o software/sistema será executado. Apesar de ser mais usado no desenvolvimento de

interfaces gráficas (GUIs), a Orientação a Eventos também é usada em banco de dados, sistemas de arquivos, aplicações comerciais e científicas, entre outros.

O Qt é uma ferramenta que possui um sistema robusto de detecção e tratamento de eventos que permite que o desenvolvedor reaproveite ou crie novos eventos com base nos que estão implementados no próprio Qt. Esse sistema de eventos foi desenvolvido de um modo que os objetos criados em um projeto qualquer possam se comunicar entre si sem a necessidade de haver um vínculo direto entre eles. Essa característica é muito importante, pois ela ajuda a reduzir o grau de **acoplamento** de um projeto, ou seja, ajuda a diminuir o grau de dependência entre as partes que compõem o projeto.

Para ilustrar de forma resumida o funcionamento geral da interface desenvolvida neste trabalho, desenhou-se um fluxograma simples, como pode ser visto abaixo, demonstrando como ocorre o fluxo de execução do *software* e de que modo os eventos influenciam nesse projeto.

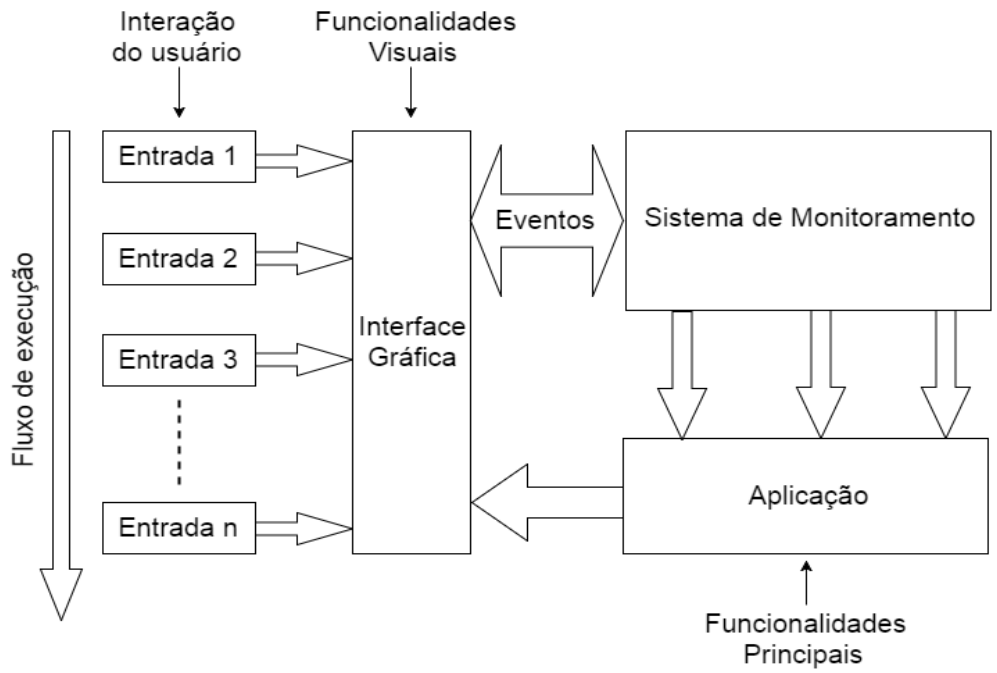


Figura 2-5: Fluxo de execução representativo da interface gráfica desenvolvida neste trabalho.

A interação do usuário com a interface gráfica é representada sob a forma das entradas 1, 2, 3, …, n. Essas entradas disparam eventos que serão detectados pelo sistema de monitoramento, desengatilhando funcionalidades visuais e principais, cujos resultados podem ser vistos pelo usuário na interface. Dessa forma, novas possibilidades

de entradas são criadas e, assim, promove-se uma continuidade no uso do sistema, caracterizando um fluxo de execução do mesmo.

As funcionalidades visuais são operações necessárias para que o usuário possa interagir com a interface de forma intuitiva, de modo que ele possa compreender e visualizar os resultados de suas ações como, por exemplo, troca de ícones ou uma animação de botão pressionado com o *click* do *mouse*. As funcionalidades principais são operações necessárias para executar uma determinada atividade considerada mais "baixo nível", isto é, que lida com, por exemplo, leitura e escrita de arquivos e cálculos, ficando "invisíveis" para o usuário.

A presença da Orientação a Objetos e da Orientação a Eventos, portanto, pode ser observada ao longo de toda a cadeia de operações da interface gráfica. Tanto as funcionalidades visuais como as funcionalidades principais e os eventos são compostos por um conjunto de objetos que foram estruturados de tal maneira para atender a todos os requisitos deste *software*.

O conceito de Orientação a Eventos surgiu no contexto em que o homem passou a ver o computador não apenas como uma máquina de fazer contas e sim como também uma ferramenta poderosa capaz de resolver diversos outros tipos de problema além da área matemática. A partir disso, o processo de desenvolvimento de *software* precisa garantir a funcionalidade principal e também que o usuário possa interagir com o mesmo de maneira mais simples, rápida e satisfatória possível.

Ao falar de possibilidades de interação é necessário então dispor de *softwares* capazes de compreender todas essas possíveis interações. Operações de Entrada e Saída (*Input*/*Output*) de *mouse*, teclado, *touch*, entre outros, precisam estar implementados para poder se adaptar às diferentes situações e dispositivos em que o usuário estiver interagindo com o *software*. Isso significa que o responsável pelo desenvolvimento também deverá entender o comportamento dos eventos e quando ocorrem para poder implementar todas as funcionalidades requeridas com mais eficácia.

Por conta de ser uma atividade imprescindível, a interação entre humanos e máquinas foi transformada em um campo de estudo, conhecida como Interação Homem-Máquina (IHM). Caracterizada por sua interdisciplinaridade, a IHM utiliza conhecimentos de diversas áreas como computação, *design*, psicologia, semiótica, entre outras, para poder compreender o ser humano e conceber um produto que mais se adeque às necessidades daquele. Na computação, esses conhecimentos são de extrema importância, pois constituem os requisitos de *software*, que são elementos fundamentais

para estabelecer uma ponte entre o cliente e os responsáveis pelo desenvolvimento do *software*, garantindo que o processo de criação e construção do produto seja executado de forma efetiva e satisfatória para ambos.

2.3 Requisitos

Os **requisitos** são um dos elementos mais cruciais no processo de desenvolvimento de um *software*. Eles descrevem uma funcionalidade ou característica na qual o sistema poderá desempenhar para que possa cumprir o seu propósito (PFLEEGER, 2004, p. 111). Eles são responsáveis por definir de que maneira a implementação do *software* será realizada, influenciando diretamente na qualidade e no tempo de construção do produto final.

A quantidade e a forma dos requisitos a serem elaborados variam de projeto a projeto, em função do escopo do projeto e ao longo do desenvolvimento do mesmo. No caso deste trabalho, inicialmente quando se teve a ideia de construir a interface gráfica, não se tinha uma noção exata do escopo do projeto. A princípio, um grupo de requisitos foi construído com o objetivo de se obter um protótipo base funcional para que, a partir da sua utilização, novas necessidades, ideias e limitações ficassem mais facilmente aparentes.

O processo de documentação e implementação dos requisitos é realizado, em conjunto, pelos desenvolvedores e pelo cliente. Nesse processo, os requisitos podem ser elaborados de diversas formas como, por exemplo, por meio de imagens, texto, documentos, entre outros; e podem ser classificados em 02 (dois) tipos: os **requisitos funcionais** e os **requisitos não-funcionais** (PFLEEGER, 2004, p. 115).

2.3.1 Requisitos Funcionais

São os requisitos responsáveis por descrever explicitamente como o sistema deve se comportar diante das possíveis interações que terá com o ambiente em que estiver inserido e com os seus utilizadores (PFLEEGER, 2004, p. 115). Eles devem ser elaborados da maneira mais precisa e esclarecida possível para evitar ambiguidade de interpretações por parte dos responsáveis pelo desenvolvimento do projeto. Qualquer

dúvida ou ausência de informação pode culminar em retrabalho e perda de prazos de entrega de funcionalidades ou de partes do projeto.

Um fator que influencia diretamente na qualidade e eficiência da documentação e implementação dos requisitos é o contato constante do cliente com o projeto ao longo do seu desenvolvimento. Isso se deve ao fato de que quando houver necessidade de revisão, reformulação ou mesmo reimplementação de requisitos, o tempo gasto para verificação e validação deles é menor do que comparado a projetos em que o cliente tem pouca participação no processo de desenvolvimento.

Neste trabalho, o processo de validação e verificação dos requisitos era realizado por meio de reuniões quinzenais, nas quais se observava o que estava satisfatório e o que poderia melhorar, além de discutir e organizar novas ideias. Não havia documentos formais para documentar os requisitos, então todos os pontos discutidos nas reuniões eram escritos em *emails* e enviados entre os membros da equipe responsável pelo projeto.

Para exemplificar a presença dos requisitos funcionais neste trabalho, segue abaixo uma tabela com alguns desses requisitos feitos para a interface gráfica.

Tabela 2-1: Tabela com exemplos de requisitos funcionais da Interface Gráfica deste trabalho.

Nº Requisito	**Requisito**	**Descrição**
01	Exibir logo da empresa	A interface gráfica deverá exibir a logo da empresa
02	Exibir modelo de velocidades original	O modelo de velocidades inicial escolhido pelo usuário deverá ser exibido
03	Exibir modelo de velocidades atualizado	O modelo de velocidades atualizado deverá ser exibido
04	Exibir gráfico de simulação	A interface gráfica deverá exibir o gráfico de simulação
05	Estudar eficácia da exibição em abas ou em múltiplas janelas	Definir qual abordagem trará mais vantagens e praticidade na utilização da interface
06	Construir botão para executar o FWI	Implementar botão com funcionalidade para executar o FWI

07	Construir terminal de saída de texto	Implementar janela com saída de texto e aparência semelhante a um terminal de um Linux
08	Construir menu principal	Implementar uma barra de menu semelhante as barras de menu das aplicações para Windows e Linux

A lista acima contém alguns requisitos funcionais elaborados durante os primeiros passos de desenvolvimento da interface gráfica. Por conta de ser o primeiro projeto de desenvolvimento de *software* voltado para cliente da PETREC, fez-se necessário realizar estudos e treinamentos em tecnologias e metodologias que permitissem a combinação da melhor navegabilidade e praticidade possível no uso da interface gráfica com a disposição de grandes quantidades de informações como imagens, textos, tabelas, entre outras. No início, o requisito 5, por exemplo, resumiu-se a um estudo com o objetivo de obter a melhor forma de dispor todas as informações das quais o usuário precisa ter acesso no antes e depois da execução do FWI. Apesar de não impedir o progresso da implementação dos outros requisitos funcionais, após a resolução do requisito 5, todos os outros requisitos que envolvem algum tipo de exibição terão que ser atualizados para se adequar a metodologia escolhida no novo requisito 5.

2.3.2 Requisitos Não-Funcionais

São os requisitos responsáveis por estabelecerem limites necessários no sistema como um todo ou em partes de forma a criar um ambiente que solucione o problema (PFLEEGER, 2004, p. 115). Por conta disso, esses requisitos são mais delicados, pois se algum deles não for satisfeito, o sistema pode não funcionar como um todo e se tornar inútil para o cliente. Por consequência, eles devem ser constantemente avaliados para verificar se estão coerentes com o propósito do *software*.

Diferente dos requisitos funcionais, os requisitos não-funcionais foram os que menos sofreram alterações durante o desenvolvimento deste trabalho. Desde o começo já se sabia quais eram, por exemplo, os sistemas operacionais em que a interface teria que funcionar. Outros requisitos foram sendo adicionados ou modificados à medida que novas ideias eram incorporadas ao projeto.

Também, para exemplificar, segue abaixo uma tabela com alguns requisitos não-funcionais da interface gráfica.

Tabela 2-2: Tabela com exemplos de requisitos não-funcionais da Interface Gráfica deste trabalho.

Nº Requisito	Requisito	Descrição
01	Compatibilidade com Windows e Linux	A interface gráfica deverá funcionar da mesma forma nos sistemas operacionais Windows e Linux
02	Linguagem de programação C++	A interface gráfica deverá ser desenvolvida em C++
03	Exibir modelo de velocidades original durante execução do FWI	O modelo de velocidades original deverá ser exibido somente após o início da execução do FWI
04	Ambiente de desenvolvimento Qt	A interface gráfica deverá ser desenvolvida utilizando o Qt
05	Estudar bibliotecas de visualização 2D e 3D compatíveis com o Qt	Pesquisar bibliotecas de visualização 2D e 3D que possam ser incorporadas ao projeto no Qt

A lista acima contém alguns requisitos não-funcionais elaborados durante os primeiros passos de desenvolvimento da interface gráfica. Como pode ser observado, os requisitos não-funcionais, também conhecidos como **restrições**, são voltados para a parte mais técnica do projeto, ou seja, para a programação do sistema e para o ambiente no qual esse sistema ficará inserido.

Muitas vezes pode acontecer de o cumprimento de um requisito não-funcional exigir a eliminação ou modificação de um ou mais requisitos funcionais. Por isso, é importante estar sempre revisando todos os requisitos do projeto para garantir a coesão e coerência das partes que compõem o produto final. Para que esse processo de revisão seja efetivo, é importante que os requisitos elaborados sejam de alta qualidade, dado que tanto o cliente quanto os responsáveis pelo desenvolvimento do projeto tem que ser capazes de entender e utilizar tais requisitos de um modo adequado.

2.4 Prototipação e Testes

Prototipação é o processo de construção de um ou mais **protótipos**, que são produtos parcialmente elaborados ao longo do processo de desenvolvimento de um projeto. Esses produtos permitem que os desenvolvedores e o cliente consigam observar ideias propostas para o sistema e avaliar se elas são adequadas para a versão final do produto (PFLEEGER, 2004, p. 41).

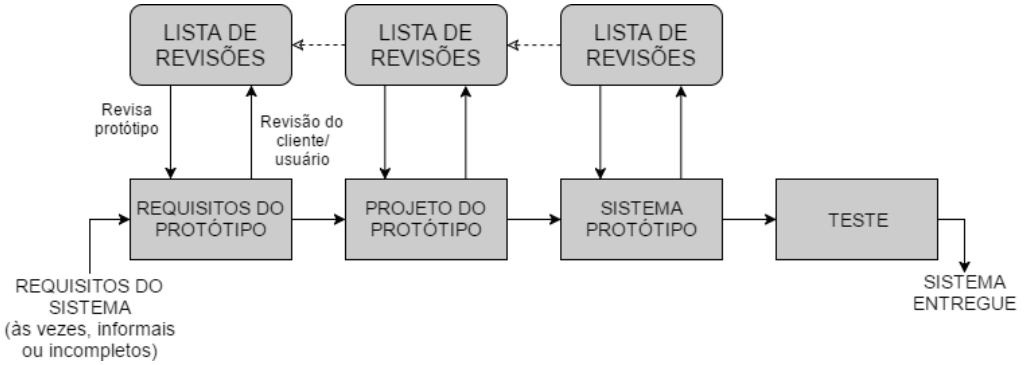


Figura 2-6: Modelo de prototipação.

Fonte: PFLEEGER, 2004, p. 43.

A construção de protótipo é um procedimento que possibilita o estudo de estratégias de desenvolvimento. Diferentes protótipos para uma mesma parte do sistema podem ser desenvolvidos a fim de utilizar aquele que apresentar as melhores propriedades e desempenho, sempre respeitando o que foi imposto pelos requisitos do sistema. Além disso, esses produtos viabilizam a observação de problemas e erros de implementação com antecedência, de modo que possam ser corrigidos antes de um teste formal do sistema.

Quando se fala de protótipo dentro do processo de desenvolvimento de *software*, logo se pensa em um *software* parcial e funcional. Entretanto, o protótipo pode ser também elaborado por meio de outras ferramentas como, por exemplo, o papel. O protótipo em papel é uma atividade que pode ser executada rapidamente, gasta poucos recursos e é muito útil para avaliar questões como *layout*, agrupamento de elementos e navegação (PREECE *et al.*, 2005, p. 260).

Neste trabalho, a prototipação em papel foi muito utilizada em função do desenvolvimento ser, especificamente, sobre uma interface gráfica, que requer uma

forte preocupação com a experiência do usuário. Segue abaixo um protótipo inicial da interface feito em papel e, posteriormente, digitalizado.

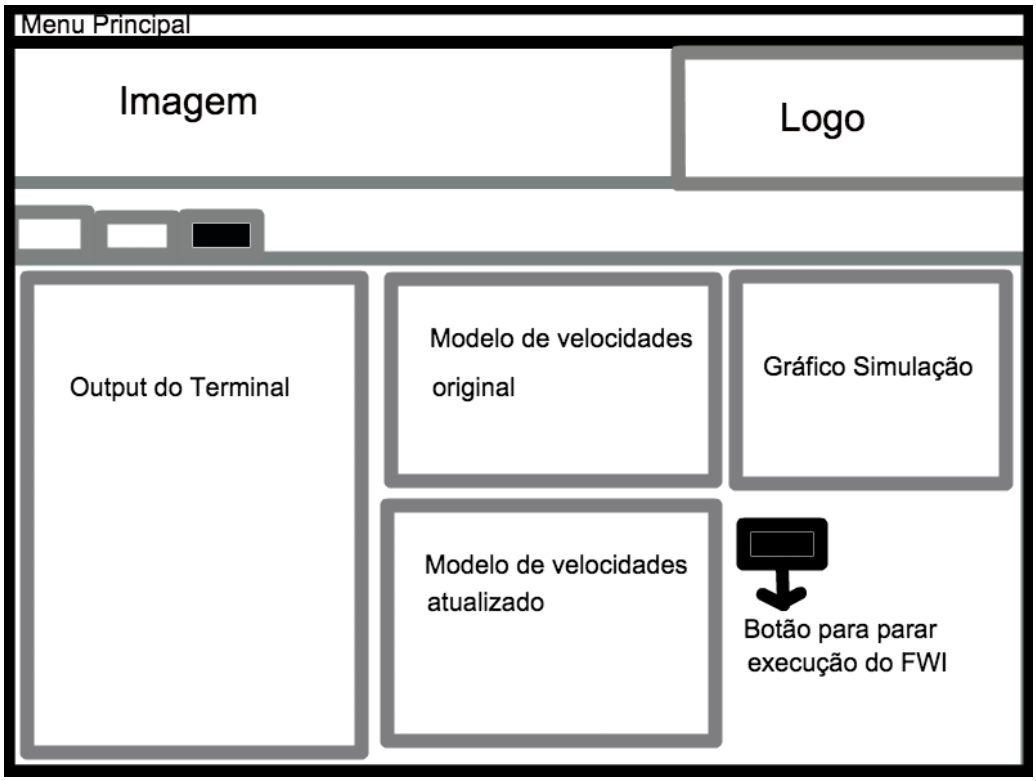


Figura 2-7: Protótipo inicial de tela da Interface Gráfica para FWI.

Esse protótipo foi muito importante, pois ele ajudou a entender os requisitos iniciais do projeto, bem como visualizar de que forma um protótipo funcional deveria ser construído e definir um passo a passo que o usuário deveria fazer para poder executar o FWI a partir da interface gráfica.

Ao longo do desenvolvimento deste trabalho, diversos protótipos em papel, em *software*, entre outros; foram elaborados. Eles, em sua grande maioria, não representavam fielmente a versão final do produto, mas contribuíram com ideias e características para o mesmo. Por conta disso, eles são classificados como **protótipos de baixa-fidelidade**, pois serviram como base para o desenvolvimento da interface gráfica, mas não possuem semelhança com a versão final. Já os **protótipos de alta-fidelidade** eram as partes de *software* produzidas individualmente e agrupadas para formar uma interface bem próxima da versão final desejada, dado que já eram confeccionadas com o mesmo material esperado para confeccionar o produto final (PREECE *et al.*, 2005, p. 262-266).

Em conjunto com a prototipação, os testes também estiveram muito presentes ao longo do desenvolvimento desta interface gráfica. Visando encontrar possíveis defeitos ou mesmo dificuldades de navegação, foram realizados diversos testes para garantir que uma versão funcional da interface gráfica sempre estivesse pronta para ser demonstrada.

Existem diversos tipos de defeitos em um sistema, assim como vários tipos de testes que podem ser feitos em um sistema. Explicar todas as variedades de defeitos e testes não é necessário, pois não se utilizou todos os tipos de testes ao longo do desenvolvimento deste trabalho. Também não se usou nenhuma ferramenta específica para detecção e controle de erros ou execução automática de testes de *software*. Os testes e inspeções eram feitos durante as reuniões ou enquanto se programava a interface gráfica.

Desenvolvimento de *software* é um processo de alto grau de complexidade que envolve a elaboração e organização de diversas tarefas que devem ser executadas a fim de se obter o produto final. Em função disso, os defeitos podem ser observados sob muitas formas e circunstâncias, pois além de dependerem do *software* em si, também dependem das expectativas dos clientes e dos usuários. Mesmo sabendo que cada projeto possui um conjunto de características e de requisitos único que pode levar a ocorrência de defeitos inesperados e diferentes, existe uma classificação ortogonal de defeitos, desenvolvida por (CHILLAREGE *et al*., 1992), que pode ser utilizada em qualquer projeto de desenvolvimento de *software*.

Tabela 2-3: Classificação ortogonal de defeitos, da IBM.

Tipo de defeito	Significado
Função	Defeito que afeta a capacidade, as interfaces com o usuário final, as interfaces do produto, a interface com a arquitetura de hardware ou as estruturas globais de dados.
Interface	Defeito na interação com outros componentes ou drivers, por meio de chamadas, macros, blocos de controle ou listas de parâmetros.
Verificação	Defeito na lógica do programa, que falha ao validar dados e valores apropriadamente, antes de utilizá-los.
Atribuição	Defeito na estrutura de dados ou na inicialização do bloco de código.
Sincronia/sequência	Defeito que envolve a sincronia de recursos compartilhados e em tempo real.
Versão/empacotamento	Defeito que ocorre devido a problemas nos repositórios, de mudanças de gerenciamento ou no controle de versões.
Documentação	Defeito que afeta as informações nas publicações ou sobre a manutenção.

Algoritmo	Defeito que envolve a eficiência ou a correção do algoritmo ou da estrutura de dados, mas não do projeto.

Fonte: PFLEEGER, 2004, p. 273.

O termo ortogonal indica que esses erros são independentes entre si, que cada um pertence a uma única categoria. Por exemplo, não existe um erro que seja, ao mesmo tempo, classificado como erro de Algoritmo e de Função. O que pode acontecer é que o erro de Algoritmo desencadeie um ou mais erros de Função, mas a nível de usuário parece ser tudo apenas um único defeito.

Essa classificação é produtiva, pois ela ajuda a perceber quais podem ser os motivos que causam a ocorrência do(s) defeito(s), além de ajudar na previsão de possíveis novos erros com base nos defeitos anteriores. O acontecimento de defeitos e a frequência com que acontecem também pode servir como um sinal indicando a necessidade de reavaliação do projeto ou, até mesmo, dos requisitos. Todas essas informações também são fundamentais para guiar o processo de elaboração e execução de testes de *software*.

Os testes de *software* tem como objetivo identificar defeitos no *software* para que, posteriormente, possam ser eliminados. Neste trabalho, a execução de testes não era realizada formalmente, com descrição do passo a passo para reprodução dos defeitos nem documentação. Apenas se escrevia ou falava o defeito ocorrido em qual parte do software e, a partir disso, realizavam-se testes até que os defeitos fossem encontrados. Utilizando o critério de frequência dos defeitos ocorridos ao longo do processo de desenvolvimento como um todo, apenas serão abordados os testes que mais foram realizados.

- **Teste de Unidade:** é uma verificação isolada do funcionamento de uma **unidade** (menor parte) do sistema para observar se diante de todas as entradas esperadas ela produz um resultado esperado. Testar o funcionamento de uma classe é um exemplo de teste de unidade.
- **Teste de Integração:** é uma verificação do funcionamento em grupo de um conjunto de unidades para observar a comunicação entre elas e se produzem os resultados esperados a partir das entradas esperadas. Testar um grupo de classes que trocam mensagens entre si é um exemplo de teste de integração.
- **Teste de Funcionalidade:** é uma verificação do funcionamento dos componentes da interface gráfica em comparação com os requisitos e expectativas do cliente. Clicar em um botão, que abre janelas, para abrir uma janela é um exemplo de teste de funcionalidade.

- **Teste de Sistema:** é uma verificação do funcionamento do sistema como um todo em um ambiente de produção simulado com o objetivo de validar funcionalidades e requisitos funcionais e não-funcionais.

A forma como se reportava a existência de defeitos e a forma como se realizavam testes funcionou no caso deste trabalho, pois a equipe envolvida no projeto desta interface era pequena, com um único desenvolvedor. Então, questões de comunicação entre membros da equipe eram rapidamente resolvidas e a verificação da eliminação dos defeitos era feita, em grande parte, ao mesmo tempo que se desenvolvia esta interface gráfica.

Saber de antemão os tipos de defeitos e testes que podem ser observados e realizados dentro do processo de desenvolvimento de um *software* é importante, pois permite um melhor planejamento e organização das tarefas necessárias para concretizar o produto final desejado. Assim, a equipe envolvida no projeto consegue com clareza verificar e validar os requisitos do projeto e antever outras necessidades do(s) cliente(s) e usuário(s).

Capítulo 3

Interação Homem Máquina

Interação Homem-Máquina (IHM), também conhecido como Interação Homem-Computador (IHC), é o estudo do modo como o homem se comunica com o computador e dispositivos similares. A compreensão do ser humano e suas características servem como base para o desenvolvimento de teorias e técnicas que visam simplificar essa comunicação. A partir dessas teorias e técnicas, constroem-se máquinas e dispositivos que sejam mais adaptados possíveis ao ser humano em questões comportamentais e estéticas para viabilizar o estabelecimento da comunicação entre eles.

Atualmente, diversas tarefas do dia-a-dia já são realizadas por meio do uso de dispositivos digitais, que também transformam a forma como o homem interage e vê o mundo ao seu redor. O ritmo de vida, relacionamentos interpessoais, trabalho, lazer e outras atividades estão permeados por diferentes tecnologias que estimulam os sentidos e a percepção, canais pelos quais todas as possíveis comunicações humanas podem ocorrer.

A comunicação pode ser definida, resumidamente, como uma troca de informações entre um emissor e um receptor. Tanto o emissor quanto receptor pode ser um humano ou máquina, e o mais importante é que ambos saibam interpretar as informações trocadas entre eles. Essa troca promove a aquisição de conhecimento, um processo fundamental denominado **Cognição** (PREECE, 1994).

3.1 Cognição

A cognição é um processo de aquisição de conhecimento que se realiza por meio do raciocínio, atenção, imaginação (criação de ideias), entendimento, obtenção de habilidades e lembrança (PREECE, 1994). A Ciência Cognitiva, área que tem a cognição como objeto de estudo, fornece uma base teórica valiosa para os estudos da IHM, visto que a compreensão do comportamento das pessoas por meio de suas características inerentes permite entender e construir canais de comunicação que sejam efetivos para o homem, ou seja, facilmente interpretados por ele (GENTIL, 2008).

A obtenção de conhecimento de uma pessoa ocorre através do processamento de informação. Essa atividade pode ser representada em um modelo no qual a informação entra e sai da mente humana unidirecionalmente, passando por uma sequência de 4 estágios ordenados de processamento, nos quais cada um possui um tempo de processamento que varia de acordo com a quantidade e complexidade de procedimentos a serem realizados (LINDSAY E NORMAN, 1977).

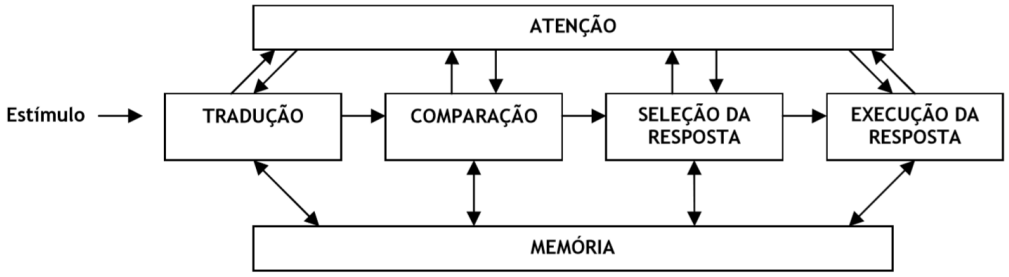


Figura 3-1: Modelo de processamento humano de informação (adaptado de PREECE, 1994)

Nesse modelo, pode-se observar as seguintes etapas:

- **Tradução:** Estágio em que ocorre a tradução da informação externa (estímulo) para um formato que possa ser compreendido internamente;
- **Comparação:** Estágio em que ocorre a comparação entre o formato gerado na **Tradução** e os modelos armazenados na memória;
- **Seleção da resposta:** Estágio em que ocorre a seleção da resposta mais satisfatória para o estímulo;
- **Execução da resposta:** Estágio em que ocorre a estruturação e cumprimento dos procedimentos para executar a resposta.

Simultaneamente a elas, ocorrem os seguintes processos:

- **Atenção:** Processo presente em todos os estágios responsável por definir a percepção do estímulo e de que modo ela ocorrerá;
- **Memória:** Processo presente em todos os estágios responsável por armazenar e dar acesso a todos os conhecimentos adquiridos a partir de estímulos anteriores.

Todo esse modelo de processamento humano de informação é viabilizado pela existência dos 5 sentidos humanos: visão, audição, olfato, tato e paladar. Cada sentido possui um conjunto de características únicas que definem a forma como ocorrerá a atenção e a memorização de um conhecimento. A partir da combinação desses sentidos

é que se estabelecem os mais diversos canais de comunicação entre o homem e o ambiente externo a ele.

Os canais de comunicação representam uma etapa fundamental no processo de obtenção de conhecimento, pois eles influenciam diretamente no grau de facilidade de interpretação do estímulo, podendo simplificar ou dificultar a elaboração de uma resposta a ele. Dessa forma, esses canais podem ser caracterizados de acordo com a eficiência que conseguem promover a aquisição de conhecimento. Dentro da IHM, existe uma área que estuda a eficiência desses canais e maneiras de melhorá-la, conhecida como **usabilidade**.

3.2 Usabilidade

Um elemento de extrema importância tanto para o desenvolvimento de produto como para o desenvolvimento de *software*, a usabilidade tem como propósito definir um conjunto de condições a serem satisfeitas para que um sistema seja considerado fácil de aprender e usar (PREECE, 1994).

As condições são definidas de acordo com as características do projeto a ser realizado e todas elas se baseiam em três (03) noções que ajudam na elaboração de um sistema com boa usabilidade (GENTIL, 2008). São elas:

- Observar de que forma os mais variados aspectos como, por exemplo, psicológicos, sociais, ergonômicos e organizacionais influenciam no modo como as pessoas utilizam máquinas/computadores em seu cotidiano (GENTIL, 2008);
- Converter as informações coletadas dessa observação em ferramentas e procedimentos que auxiliem no projeto (GENTIL, 2008);
- Atingir de que forma os mais variados aspectos como, por exemplo, psicológicos, sociais, ergonômicos e organizacionais influenciam no modo como as pessoas utilizam máquinas/computadores em seu cotidiano (GENTIL, 2008);
- Converter as informações coletadas dessa observação em ferramentas e procedimentos que auxiliem no projeto (GENTIL, 2008);

Para o caso de projetos de *software*, essas noções são importantes, pois influenciam na elaboração dos requisitos funcionais e não-funcionais, contribuindo para um melhor planejamento do desenvolvimento e do ciclo de vida do *software*.

Segundo a norma ISO 9241-11, usabilidade é uma medida baseada na efetividade, eficiência e satisfação no uso de um produto por um usuário que busca um resultado

específico. A efetividade mede a finalização do objetivo do usuário, bem como a qualidade do resultado final obtido. A eficiência mede o esforço e a quantidade de recursos usados para o usuário alcançar seu objetivo. A satisfação mede a agradabilidade no uso de um sistema. A norma ISO 9241-11 também discrimina outras medidas possíveis para usabilidade, que são mais específicas e que estão dentro do mesmo escopo de atuação da efetividade, eficiência e satisfação.

A efetividade e eficiência são 2 métricas objetivas, podendo ser medidas a partir dos erros e caminhos que são realizados pelo usuário para que ele consiga alcançar o seu objetivo. Já a satisfação é uma métrica subjetiva, pois ela depende da percepção e expectativa do usuário com relação ao produto, seu uso e seus resultados.

Apesar de seu caráter subjetivo, a satisfação pode ser medida por meio da realização de testes de usabilidade. Esses testes tem como objetivo coletar informações sobre o uso de um determinado produto por usuários alvo do mesmo. O uso do produto pode ser livre ou guiado pela realização de uma sequência de tarefas típicas e críticas para operar o sistema. As informações coletadas vão servir de insumo para implantar melhorias, correções e/ou modificações no produto.

Alguns exemplos de testes de usabilidade podem ser vistos em (GENTIL, 2008), porém este trabalho não abordará nenhum teste, visto que para trazerem resultados mais perceptíveis, os testes devem ser realizados com usuários finais, enquanto que o *software* desenvolvido neste trabalho está sendo usado apenas internamente até a presente data.

Quando se diz que um produto tem uma boa usabilidade, quer-se dizer que é fácil aprendê-lo e usá-lo, que apresenta uma boa interação com o usuário alvo. Essa facilidade de comunicação entre o usuário e o sistema é o fator responsável por criar uma percepção positiva desse para o usuário, destacando-o perante os outros produtos existentes. Por conta disso, a necessidade de usabilidade tem crescido gradativamente e a preocupação com o usuário tem ganhado prioridade em vários projetos de *software* (GENTIL, 2008).

3.3 Experiência de Usuário

Experiência de usuário, também conhecida por *UX*, é um tema que vem ganhando interesse de diversas empresas e negócios, porém não há um consenso com

relação ao seu escopo e sua natureza. O que se sabe é que *UX* pode ser vista como o conjunto de todos os aspectos da interação do usuário final com a empresa, seus serviços e seus produtos (LAW, 2009). Esse conceito está intimamente ligado com o conceito de desenvolvimento centrado no usuário, o qual determina que o processo de construção do produto deve ser inicializado com base nos usuários e suas necessidades e não pela tecnologia em si (NORMAN, 1999).

A preocupação com o usuário é um assunto que vem ganhando cada vez mais importância dentro dos projetos de produto/*software*. Construir um produto já sabendo que um cliente irá consumi-lo reduz drasticamente o risco do projeto, pois isso representa um investimento com retorno garantido, influenciando positivamente a organização e execução das atividades pela equipe executora.

Dentro de uma empresa, a experiência de usuário pode ser percebida nos processos de elaboração de estratégias para atrair e estabelecer um canal de comunicação com potenciais clientes. Essas escolhas definem não só a forma como abordar um cliente como também fidelizá-lo, ou seja, mantê-lo utilizando o produto ou serviço oferecido.

No caso de uma empresa de base tecnológica, geralmente o produto ou serviço oferecido se baseia ou é propriamente um *software*. Neste trabalho, a proposta era transformar o algoritmo pronto de FWI em um produto que pudesse ser comercializado, pois no formato que estava isso não seria possível. Diante do fato que o FWI era compatível apenas com computadores que tivessem os sistemas operacionais Windows e Linux, era necessário que a solução fosse também para computadores.

Diante dos requisitos, escolheu-se desenvolver uma interface gráfica por conta da facilidade de se implementar um protótipo usável que, ao mesmo tempo, seria capaz de possuir uma identidade visual e poderia transmitir, intuitivamente, o diferencial da solução proposta pela empresa com este *software*. Dessa forma, esta interface serviria como um “cartão de visitas”, demonstrando a área de atuação da empresa e o poder criativo de suas soluções.

Os benefícios da experiência de usuário vão além das melhorias das condições de uso do produto ou serviço e da questão de aceitabilidade e satisfação do usuário, visto que também influencia a organização interna da empresa e de seus colaboradores. A contribuição de todos os envolvidos no projeto é importante e decisiva para a experiência final que será obtida tanto pelos usuários quanto pela equipe desenvolvedora.

Capítulo 4

Inversão de forma de onda (*FWI*)

Por conta da alta lucratividade do mercado de petróleo, áreas relacionadas a geofísica e geologia evoluíram rapidamente para poder atender às crescentes demandas da indústria. Aliado a isso, o avanço tecnológico dos computadores, e, em particular, o desenvolvimento da computação paralela, também fomentou a criação de novas ferramentas e o aprimoramento de técnicas que antes não eram utilizadas. Tais técnicas e ferramentas, que consomem um grande volume de dados, levam muito tempo para gerar resultados quando utilizadas no contexto da computação convencional, onde há uma escassez de recursos computacionais, mas quando utilizadas dentro da abordagem paralela, geram resultados em um ritmo muito mais satisfatório. (MIRANDA, 2011).

A inversão de forma de onda, mais conhecida pela sigla *FWI* (*Full Waveform Inversion*) (TARANTOLA, 1984), é uma técnica de análise de velocidades com alto custo computacional, pois sua formulação é baseada na resolução da equação da onda completa. A inversão de forma de onda (TARANTOLA, 1984a) tem como finalidade estimar a velocidade de propagação das ondas sísmicas nas diversas camadas geológicas que compõem a subsuperfície terrestre, e, a partir disso, construir um modelo de velocidades representante da região analisada. Essa etapa é de suma importância, visto que o modelo de velocidades serve como base para diversas etapas do processamento sísmico e interpretação sísmica. (SANTOS, 2013).

A fim de representar a propagação do campo de ondas em um determinado meio geológico e o registro do campo de ondas em posições arbitrárias, utiliza-se a equação (1), descrita abaixo, que expressa a correspondência entre o modelo de parâmetros m e o que é registrado em superfície. Ele não é o método que estima a velocidade de propagação e sim um operador que representa a propagação em si.

$$G(m) = d, \tag{1}$$

Nela, **m** corresponde ao modelo de velocidades onde ocorre a propagação do campo de ondas; **d** corresponde ao registro da pressão denominado dado sísmico e/ou

sismograma e **G** é o operador de modelagem que descreve a relação entre **m** e **d**. **G** pode ser calculado de diversas maneiras, porém não serão abordadas neste trabalho, mas a descrição de tal processo pode ser encontrada em (VIRIEUX E OPERTO, 2009). Assim, a solução formal da equação (1) é:

$$m = G^{-1}(d), \tag{2}$$

Ou seja, dado o operador de modelagem da propagação do campo de ondas e o dado sísmico registrado é possível obter, através de um processo de inversão, o modelo de velocidades do meio. Em função da complexidade do mecanismo de propagação das ondas sísmicas, o cálculo dos operadores **G** e $\mathbf{G^{-1}}$ pode ser inviável, tornando essencial o uso de aproximações e estratégias de inversão para o seu cálculo (SANTOS, 2013).

No presente, o método sísmico é a fase na qual ocorre a maior parte das descobertas relacionadas às características da subsuperfície e as possibilidades de prospecção de hidrocarbonetos. Por ser um processo que envolve o processamento de dados registrados, é fundamental que a qualidade desses dados seja a melhor possível para garantir que o resultado seja o mais preciso dentro das possibilidades. Dentro dessas circunstâncias, o FWI vem ganhando destaque perante as outras várias técnicas de análise de velocidades, pois ele fornece um modelo de velocidades de alta resolução, facilitando que as etapas subsequentes obtenham resultados mais fiéis, com maior qualidade e menor risco (SANTOS, 2013).

A inversão de forma de onda (TARANTOLA, 1984a) é uma técnica iterativa, em que cada iteração tem como objetivo minimizar o erro entre o dado modelado com base em um modelo de velocidades de entrada **m** e o dado observado **d** obtido em campo. A diferença entre o dado modelado e o dado observado serve de entrada para uma função norma que determina se houve melhoria (convergência) ou piora do modelo de velocidades. A convergência total ocorre quando o valor mínimo de erro previamente definido for atingido (SANTOS, 2013).

Ao iniciar a primeira iteração, tem-se como entrada um modelo de velocidades inicial, um arquivo com uma lista de parâmetros com seus respectivos valores e um ou mais sismogramas referentes a região do modelo de velocidades. Após executada a primeira iteração, um novo modelo de velocidades é gerado, de modo que servirá como entrada para a próxima iteração. Esse ciclo se repete até atingir um critério de parada

pré-estabelecido. Essa convergência significa que foi obtido com sucesso o melhor modelo de velocidades possível com base em todas as entradas fornecidas.

4.1 Integração com a Interface Gráfica

O código FWI, quando executado independente da interface, possui uma estrutura bem definida de diretórios para facilitar a navegação do usuário sobre os arquivos de entrada e saída dos quais precisa e produz. Portanto, garantir que esse ambiente de execução fosse respeitado se tornou um dos requisitos da interface, o que justificou a criação de uma tela de configuração e criação de projeto.

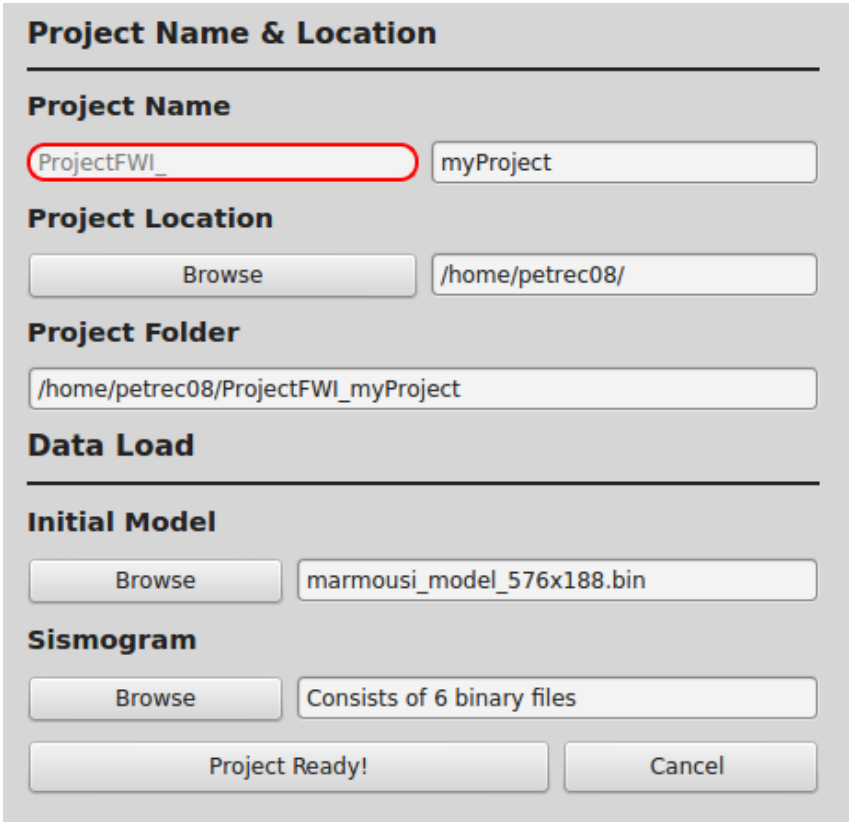


Figura 4-1: Tela de criação e configuração de um projeto

O usuário tem a liberdade de escolher o diretório onde deseja salvar o projeto, bem como dar um nome para a pasta que conterá esse projeto. Além disso, pode-se escolher um modelo de velocidades e um ou mais sismogramas de qualquer lugar do computador local do usuário. Apenas quando todos estes campos estiverem preenchidos é que o projeto e a estrutura de diretórios poderão ser criados.

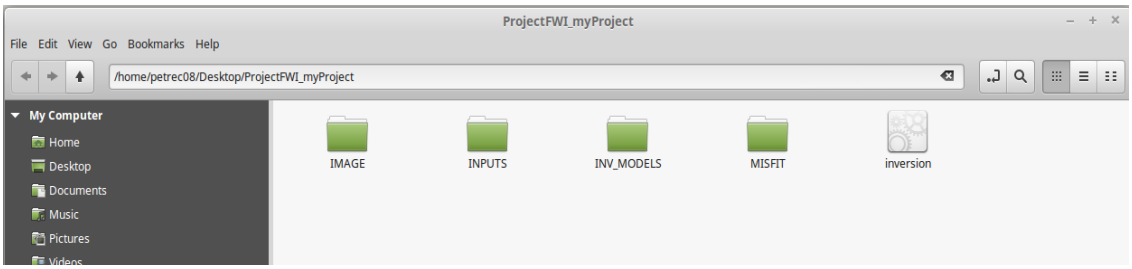
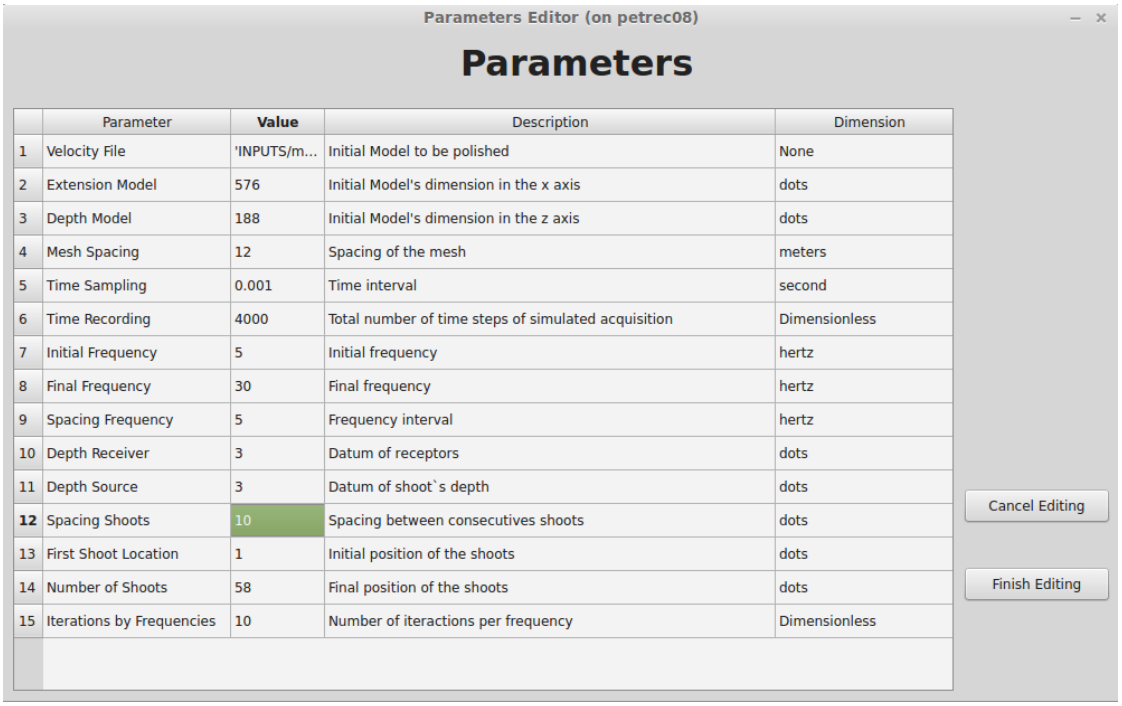


Figura 4-2: Estrutura de diretórios criada pela interface

Sem a interface, os valores dos parâmetros são preenchidos e definidos em um arquivo texto especificamente formatado para que possa ser lido pelo código *FWI*. Por ser uma etapa necessária, criou-se uma tela de edição de parâmetros. Cada parâmetro possui um intervalo de valores no qual pode ser atribuído e também existem parâmetros que influenciam os valores de outros parâmetros. Isso tudo foi projetado para facilitar o preenchimento dos valores e evitar que o usuário possa atribuir valores que não sejam adequados para a FWI, controle esse inexistente quando se edita diretamente no arquivo texto. O tratamento desses valores será mais bem explicado no **Capítulo 5**.



	Parameter	Value	Description	Dimension
1	Velocity File	'INPUTS/m...	Initial Model to be polished	None
2	Extension Model	576	Initial Model's dimension in the x axis	dots
3	Depth Model	188	Initial Model's dimension in the z axis	dots
4	Mesh Spacing	12	Spacing of the mesh	meters
5	Time Sampling	0.001	Time interval	second
6	Time Recording	4000	Total number of time steps of simulated acquisition	Dimensionless
7	Initial Frequency	5	Initial frequency	hertz
8	Final Frequency	30	Final frequency	hertz
9	Spacing Frequency	5	Frequency interval	hertz
10	Depth Receiver	3	Datum of receptors	dots
11	Depth Source	3	Datum of shoot`s depth	dots
12	Spacing Shoots	10	Spacing between consecutives shoots	dots
13	First Shoot Location	1	Initial position of the shoots	dots
14	Number of Shoots	58	Final position of the shoots	dots
15	Iterations by Frequencies	10	Number of iteractions per frequency	Dimensionless

Figura 4-3: Tela de edição de parâmetros

Outra funcionalidade que auxilia bastante o usuário é o console. Ele tem como objetivo mostrar o andamento do algoritmo em tempo real para que se possa ter uma estimativa de tempo total que levará a execução completa do código, bem como

acompanhar o comportamento de cada iteração. Dessa forma, o usuário será capaz de examinar se os valores que utilizou nos parâmetros são satisfatórios e estão levando a convergência do algoritmo. Além disso, viabiliza-se também a observação de eventuais erros de execução que, quando identificados, poderão ser evitados ou contornados.

Por ser um código que utiliza técnicas de computação paralela, é fundamental informar de que forma os recursos computacionais estão sendo alocados e usados durante sua execução. Isso também é feito no console, no início da execução, para que o usuário possa distribuir melhor o uso dos programas do computador e evite o "congelamento" do mesmo.

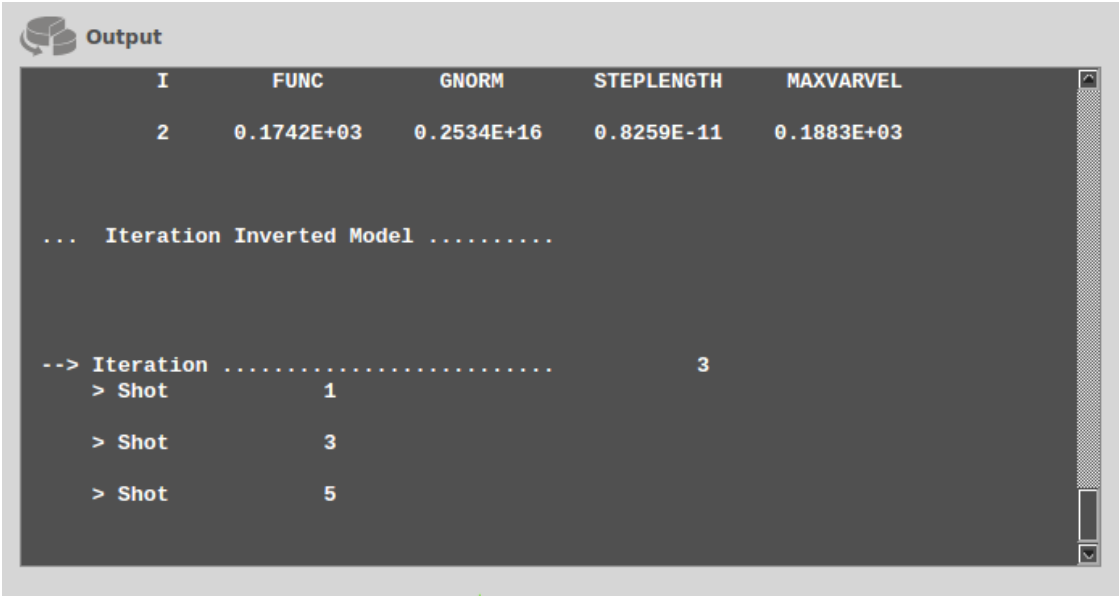


Figura 4-4: Console para mensagens de execução do código

Devido ao grande volume de cálculos e alta complexidade computacional, a *FWI* é um processo que possui um tempo de execução elevado, levando de horas até dias para poder ser executado por completo, sem necessariamente gerar um resultado satisfatório. Por conta dessa demora, é fundamental que o usuário seja capaz de avaliar, em tempo real, a convergência da execução e tenha o poder de interrompê-la caso a previsão aponte para a não convergência. Para isso, foram desenvolvidas 2 formas de visualização de dados que facilitam essa verificação.

A primeira delas é a comparação de imagens. Cada iteração gera um novo modelo de velocidades como um arquivo de saída que, se comparado ao modelo que serviu de entrada para a iteração, apresenta diferenças que podem representar uma

melhora, estagnação ou piora na qualidade dos dados. Esse modelo de velocidades gerado pelo *FWI* é um arquivo de imagem pronto, no qual o trabalho da interface é apenas mostrá-la.

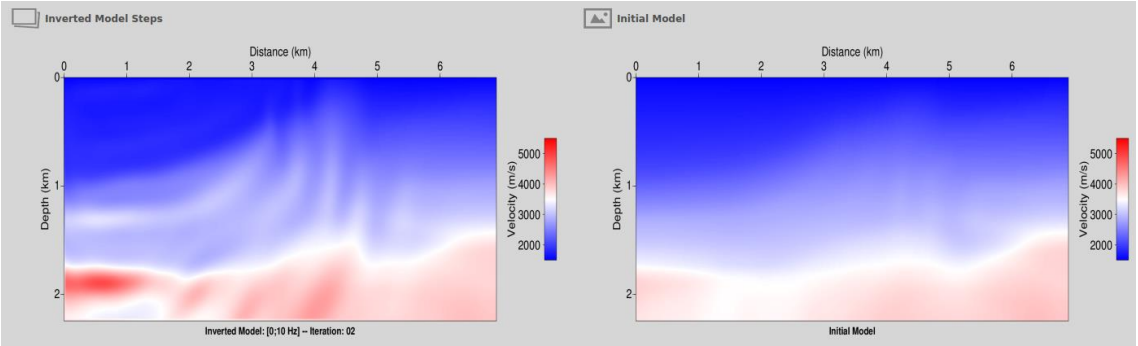


Figura 4-5: Modelo de velocidades parcial (esquerda) e modelo de velocidades inicial (direita)

Por conta de ser uma metodologia que depende de um olhar altamente técnico de um profissional da área de geologia ou geofísica, as imagens são posicionadas lado a lado para facilitar a interpretação e a navegação do usuário, visto que colocando-as em janelas diferentes ou em um arranjo mais complexo levaria a perda de atenção do usuário para os detalhes das imagens em função das necessidades de navegação.

A outra ferramenta para auxiliar a verificação da convergência do algoritmo é a utilização de gráfico de pontos. Foram feitos 2 gráficos, um referente a iteração que está em andamento e outro referente a todas as iterações já executadas. Assim, o usuário poderá verificar, com facilidade, se os valores dos parâmetros utilizados conseguem exercer uma melhoria significativa no modelo de velocidades. Também com o uso do gráfico acumulado pode-se verificar se as próximas iterações trarão ganhos significativos para o modelo final, pois em caso negativo, o processo pode ser interrompido e, assim, evitar maiores gastos computacionais e perda de tempo com um processo que não trará resultado satisfatório.

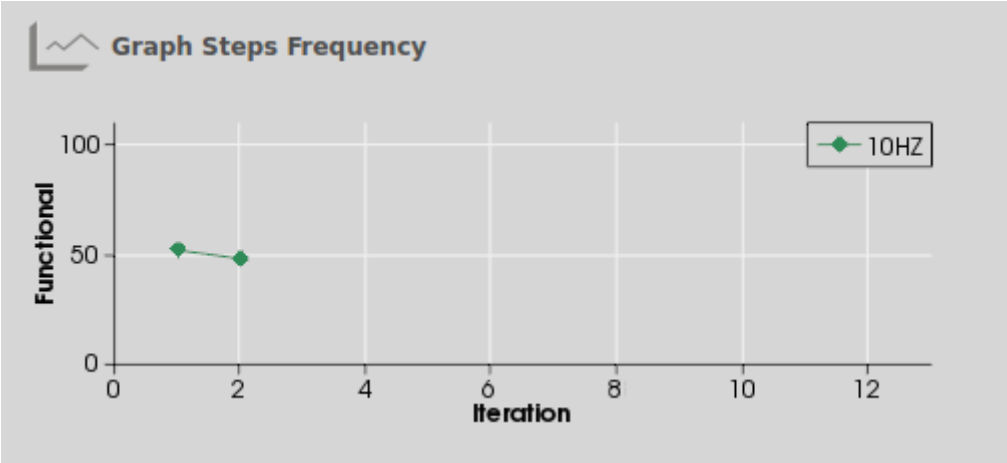


Figura 4-6: Gráfico de pontos de minimização do erro do algoritmo para a iteração que estiver em andamento

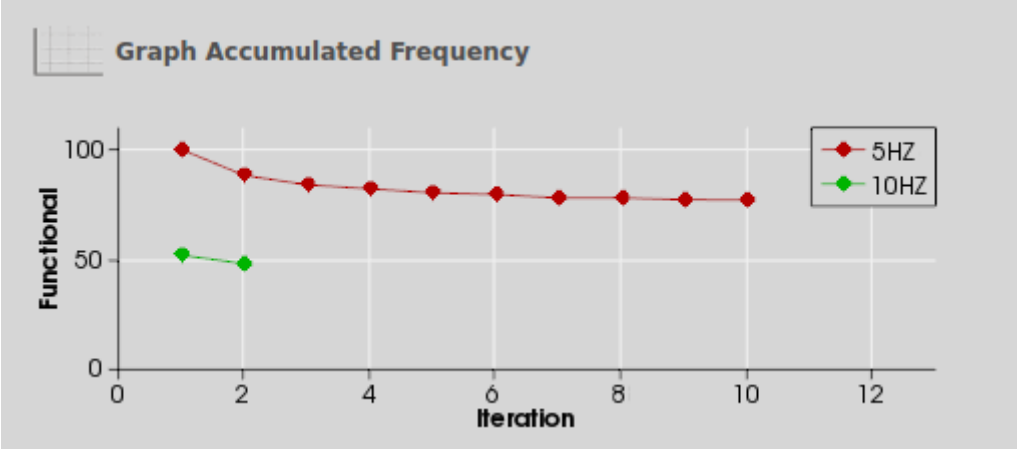


Figura 4-7: Gráfico de pontos acumulado de minimização do erro para todas as iterações

Cada ponto do gráfico representa o resultado da função norma de minimização do erro, expondo o valor do erro entre o dado modelado (modelo de velocidades gerado pela última iteração) e o dado observado (modelo de velocidades inicial). Dessa forma, pode-se observar em quais iterações obteve-se maior minimização do erro e o grau de melhoria que iterações futuras poderão trazer.

Todos os componentes da interface gráfica foram desenvolvidos preparados para poderem ser constantemente atualizados à medida que o código do *FWI* evoluísse. Todas essas funcionalidades foram criadas baseando-se na versão mais atual e estável do código da *FWI* para garantir que, sempre que necessário, existisse uma versão funcional da interface para ser demonstrada para algum cliente, alguma reunião ou

evento de negócios ou investidores. Cada componente também foi desenvolvido de forma independente como se fossem "módulos" para que, quando combinados, pudessem representar o layout principal da interface gráfica. Isso foi feito dessa forma, pois facilita a depuração a nível unitário e em nível de integração, além de facilitar a execução de testes. Ao combinar todas as telas desenvolvidas independentemente, obtém-se a tela principal da interface:

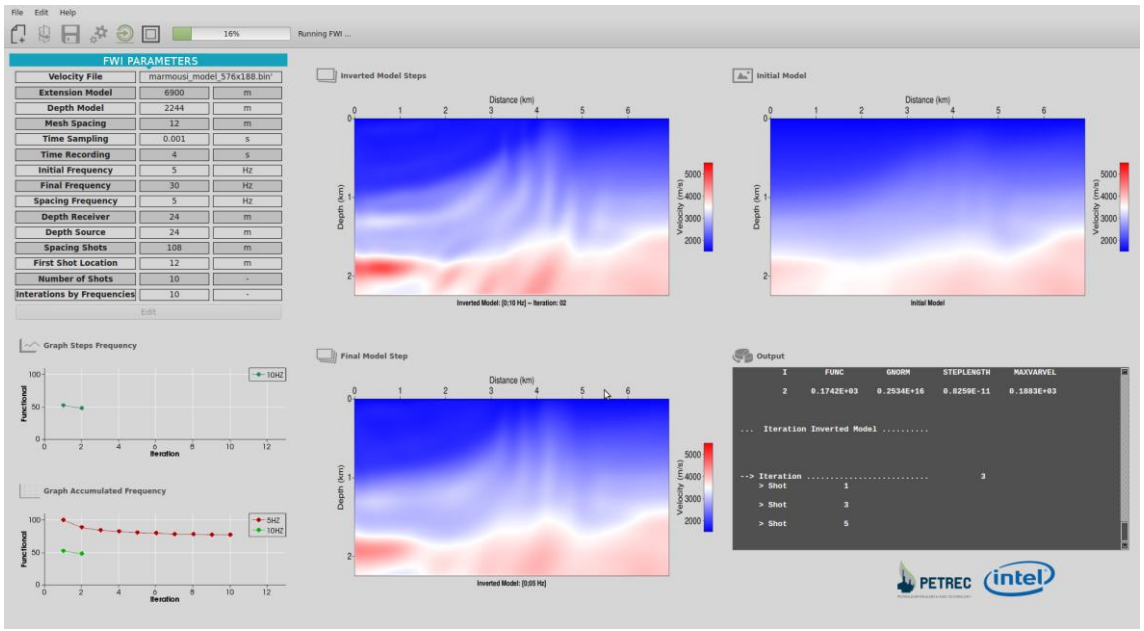


Figura 4-8: Tela principal da interface gráfica

Quando o algoritmo atinge a convergência, a interface emite uma janela de diálogo alertando o usuário de que o processo foi concluído, em que deve ser clicada para que se possa continuar utilizando a interface. Além disso, foi construída uma barra de progresso que indica o andamento do processo para que seja possível estimar o tempo total de execução e, ao mesmo tempo, mostrar que o computador está de fato executando e não está "travado".

4.2 Execução de caso de teste

Para este trabalho, foi realizado um caso de teste em que se executou o *FWI* pela interface com os mesmos valores e arquivos de entrada de um caso de teste anteriormente realizado pelo algoritmo sem a interface e que produziu resultado

satisfatório. Assim, pôde-se avaliar se as janelas e ferramentas de visualização estavam operando da forma esperada.

Com relação aos arquivos de entrada, utilizou-se um modelo de velocidades e seis (06) sismogramas intervalados por um valor de frequência de 5 *Hertz*, variando de 5 *Hertz* até 30 *Hertz*. Deu-se um nome ao projeto e escolhida a sua localização na estrutura de diretórios do computador, criou-se a estrutura de pastas do projeto. Em seguida, preencheram-se os valores das variáveis de entrada para poder dar início a execução do caso de teste.

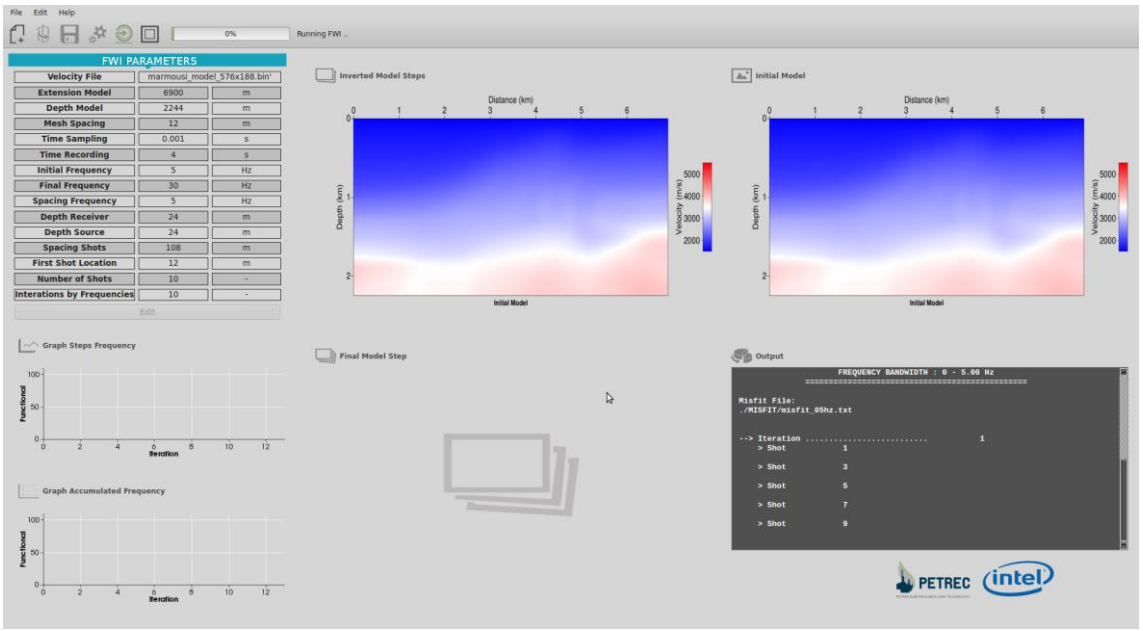


Figura 4-9: Início da execução do caso de teste

A tabela “FWI Parameters” contém os nomes de todas as variáveis de entrada do algoritmo e seus respectivos valores e unidades de medida, que também estão de acordo com os arquivos de entrada utilizados. Nesse estágio inicial, a imagem em “Inverted Model Steps” é a mesma que aparece em “Initial Model” até o momento em que a primeira iteração tiver sido executada por completo. A imagem em “Final Model Step” aparecerá depois que todas as iterações do sismograma na frequência de 5 Hertz terminarem. Nesse caso, serão 10 iterações por frequência, que é o valor da variável “Iterations by Frequencies”. A primeira frequência “Initial Frequency” é de 5 Hertz e a última “Final Frequency” é 30 Hertz, em que são espaçadas de 5 em 5 Hertz, que é o “Spacing Frequency”.

Paralelo a isso, o console vai mostrando o passo-a-passo dentro de cada iteração, imprimindo os tiros executados e o resultado obtido no término da iteração, em que cada iteração executa 10 tiros, que é o valor da variável “Number of Shots”. Os tiros são espaçados entre si de 108 em 108 metros, valor de “Spacing Shots”, em que o primeiro tiro se localiza a 12 metros, valor de “First Shot Location” do ponto de origem da região demarcada para o sismograma.

Após cada iteração, atualiza-se o “Inverted Model Steps”, o “Graph Steps Frequency” e o “Graph Accumulated Frequency”, e a cada troca de frequência atualiza-se o “Final Model Step”.

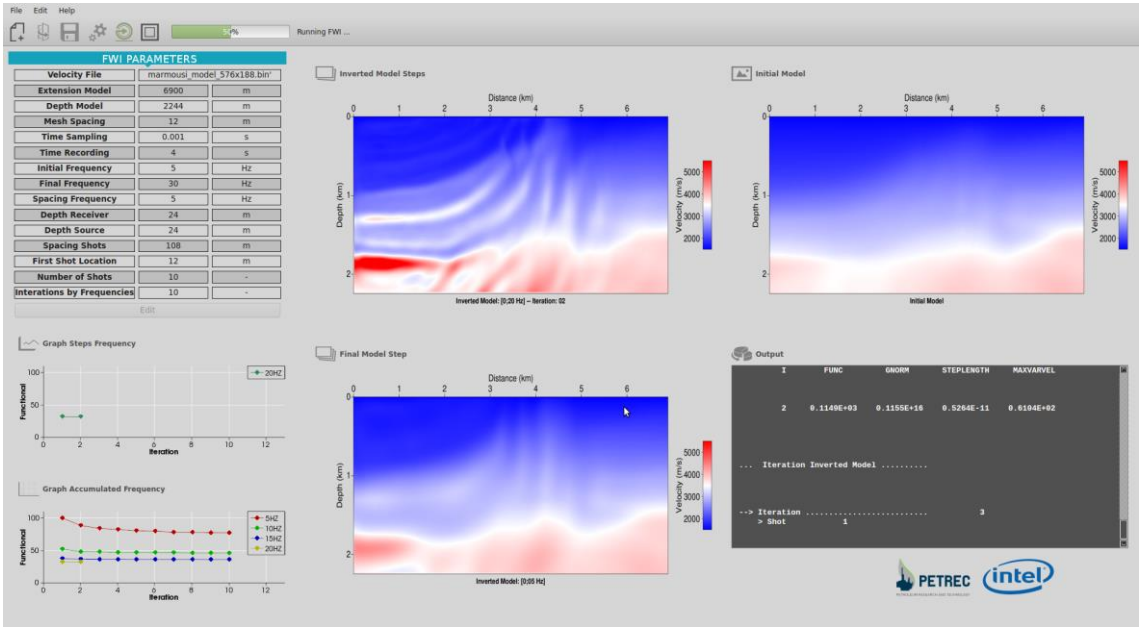


Figura 4-10: Metade da execução do caso de teste

Nessa etapa desse processo, já pode-se perceber uma melhoria significativa na definição do modelo de velocidades. Além disso, pode-se verificar em quais frequências houve o maior percentual de melhoria do modelo, bem como estimar o tempo total de execução do algoritmo.

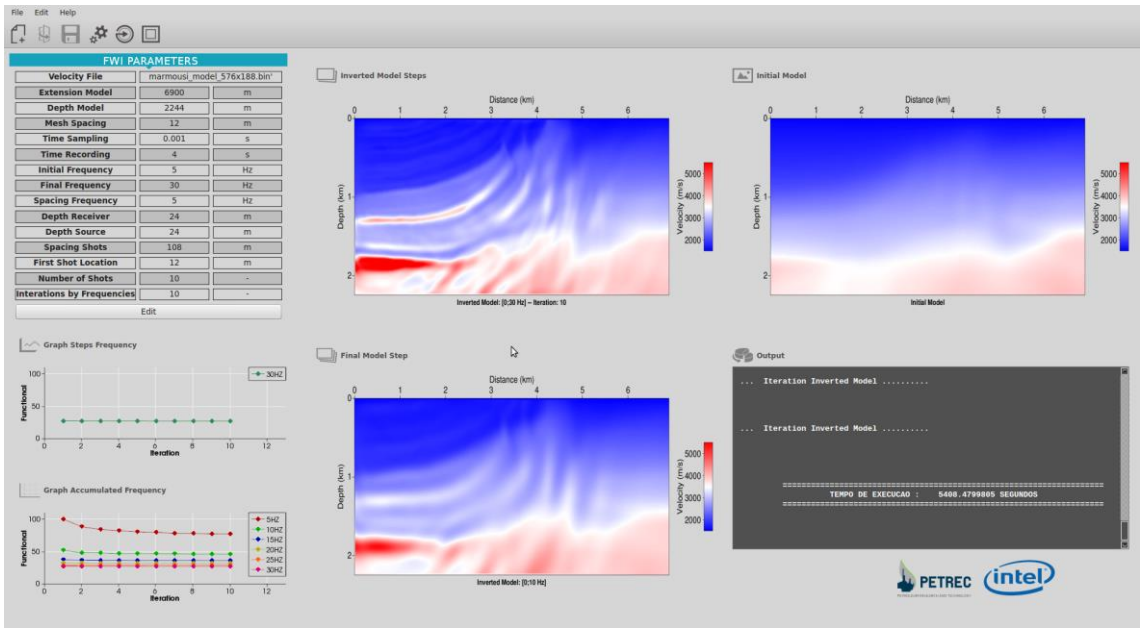


Figura 4-11: Término da execução do caso de teste

Após a execução de todas as iterações em todas as frequências, disponibiliza-se o tempo total de execução, que nesse caso foi de 5409 segundos, aproximadamente 1 hora e meia, e todas as ferramentas de visualização continuam preenchidas para análise do usuário. Caso queira uma nova execução, o usuário pode atualizar ou não os valores dos parâmetros de entrada e clicar no botão de executar.

Esse caso foi possível graças a posse do modelo de velocidades inicial e de sismogramas dentro dessa faixa de frequência e para essa região. O resultado final obtido foi considerado satisfatório e desencadeou o planejamento de novas funcionalidades e possibilidades para o *software* e para a empresa, o que será mais bem detalhado no **Capítulo 6** no tópico **Trabalhos Futuro**.

Capítulo 5

Desenvolvimento do *Software*

Desenvolvimento de *Software* pode ser definido como um processo de concepção de um *software*, em que o objetivo é resolver um problema, seja ele de caráter científico, mercadológico ou mesmo pessoal. Nesse processo, um conjunto de atividades é definido a fim de identificar todos os recursos necessários para que o *software* possa ser concretizado. Essas atividades são organizadas e distribuídas sob a forma de etapas, que são ordenadas de acordo com o projeto elaborado. Dentre essas etapas, existe uma que é a mais conhecida na computação, que é a **codificação**.

A **codificação** é umas das etapas mais delicadas do processo de desenvolvimento de *software*, visto que ela é a responsável por traduzir todo o projeto do *software* para a linguagem de máquina, que é a única que o computador é capaz de interpretar. Essa tradução é feita por meio do uso de uma linguagem de programação, que permite um programador definir instruções a serem executadas pelo computador. Cada linguagem de programação possui o seu próprio conjunto de regras sintáticas e semânticas, definindo, assim, sua estrutura de dados e seu escopo de atuação.

Neste trabalho, a linguagem de programação escolhida para ser usada foi o C++. Devido ao seu grande desempenho e a possibilidade de gerenciar o uso da memória manualmente, ela é a escolha preferencial para aplicações que demandam muitos recursos computacionais como, por exemplo, aplicações científicas, jogos, entre outras.

Por conta de sua popularidade, existem diversas bibliotecas em C++ que implementam algoritmos e funcionalidades voltados para as mais diversas finalidades de um *software*. Para o caso de desenvolvimento de interfaces gráficas, existem algumas opções dentre as quais a escolhida para este projeto foi o Qt com a *IDE* Qt Creator.

5.1 Qt

Qt é um *framework* de desenvolvimento de interfaces gráficas utilizando a linguagem de programação c++ como base. Sua principal característica é a

possibilidade de criar aplicações multiplataforma a partir de um mesmo código fonte, além de possuir diversas outras funcionalidades não relacionadas à *GUI* como, por exemplo, banco de dados, redes e reprodução de gráficos 3D, que são entregues sob a forma de módulos, podendo ser usados de acordo com a necessidade de cada projeto (BLANCHETTE, 2008, p. 8).

5.1.1 Estrutura de projeto

Todo projeto de interface gráfica em C++ no Qt apresenta um estrutura básica de diretórios e arquivos como pode ser vista abaixo:

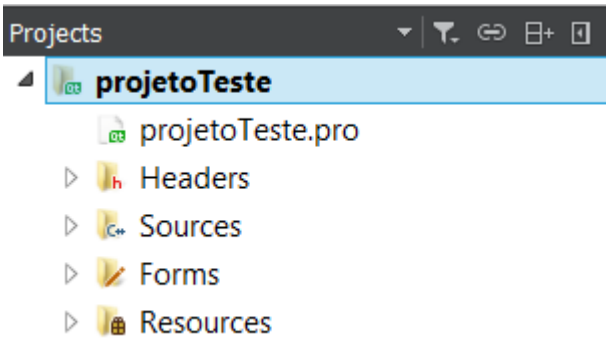


Figura 5-1: Estrutura base de diretórios de um projeto Qt

O arquivo projeto (.pro) é um arquivo com as configurações e diretivas básicas de compilação de todo o código fonte usado no projeto. A pasta “Header” contém todos os arquivos *header* (.h), a pasta “Source” contém todos os arquivos *source* (.cpp), a pasta “Forms” contém todos os arquivos *Form* (.ui), e a pasta “Resources” contém arquivos *resource* (.qrc).

Os arquivos *header* armazenam as declarações das funções/métodos e das variáveis. Os arquivos *source* armazenam as definições das funções/métodos e variáveis declaradas no arquivo *header* relacionado. Os arquivos *Form* armazenam dados e características da *widget* e de seus elementos gráficos no formato *XML*. Os arquivos *Resource* armazenam os caminhos relativos para outros tipos de arquivos que fazem parte da aplicação e são utilizados por ela como imagens, arquivos de tradução, entre outros. Respeitando essa estrutura, o mapeamento de todos os arquivos a serem usados no projeto se torna simples e intuitivo, como pode ser visto abaixo.

projetoTeste.pro

```
QT       += core gui

greaterThan(QT_MAJOR_VERSION, 4): QT += widgets

TARGET = projetoTeste
TEMPLATE = app

# The following define makes your compiler emit warnings if you use
# any feature of Qt which as been marked as deprecated (the exact warnings
# depend on your compiler). Please consult the documentation of the
# deprecated API in order to know how to port your code away from it.
DEFINES += QT_DEPRECATED_WARNINGS

SOURCES += \
        main.cpp \
        mainwindow.cpp

HEADERS += \
        mainwindow.h

FORMS += \
        mainwindow.ui

RESOURCES += \
    images.qrc

```

Figura 5-2: Arquivo projeto (.pro) básico de uma aplicação Qt

5.1.2 Elementos Gráficos

O Qt, assim como outros programas de desenvolvimento de interface gráfica como, por exemplo, o Java *Swing*, apresentam um modelo de construção de janelas que se baseia em um formato "aplicação com janela principal".

main.cpp

```cpp
#include "mainwindow.h"
#include <QApplication>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    w.show();

    return a.exec();
}

```

Figura 5-3: Código base para execução de uma aplicação Qt em C++

Os componentes básicos do Qt *QApplication* e *MainWindow* representam, respectivamente, a aplicação e a janela principal. Nesse modelo, todos os elementos gráficos e janelas adicionais são criados de forma independente, mas, em tempo de execução, tornam-se acessíveis apenas a partir da existência da janela principal, ou seja, *MainWindow* deve estar visível para que o usuário possa interagir com a interface (BLANCHETTE, 2008, p. 18).

O Qt Creator possui uma ferramenta de criação de interfaces de forma interativa e rápida denominada *Designer*. Nela é possível escolher e posicionar elementos gráficos dentro de uma janela e visualizar o resultado final antes da execução do programa, permitindo uma rápida modelagem da arquitetura da informação a ser usada (BLANCHETTE, 2008, p. 40).

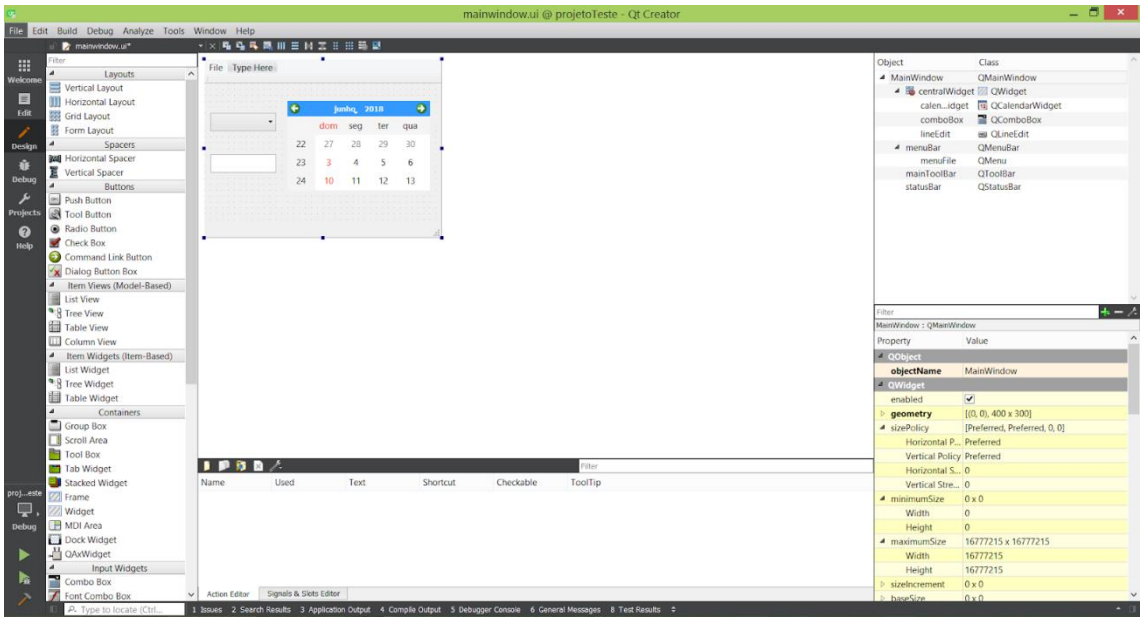


Figura 5-4: Ferramenta *Designer* disponível no Qt *Creator*

Nesta janela, ao lado esquerdo, existe um *menu* contendo vários elementos gráficos já implementados que podem ser reutilizados na construção da *GUI*. Eles podem ser posicionados dentro da janela que está na região central e suas propriedades podem ser visualizadas e modificadas no *menu* ao lado inferior direito. Além disso, esses elementos podem ser posicionados na janela de maneira livre ou respeitando um *layout*.

No Qt, o *layout* representa um mecanismo de arranjo automático de elementos gráficos dentro de uma janela, cujo objetivo é garantir o melhor aproveitamento do espaço livre disponível na janela. Existem diversos tipos de *layout* e também é permitido fazer quaisquer combinações de *layouts*, de forma que seja possível ter na janela toda a informação necessária distribuída de maneira amigável para garantir uma boa usabilidade do programa (BLANCHETTE, 2008, p. 140). A construção do *layout* de uma janela pode ser feita diretamente pelo *Designer* como também por código.

Code View:

```
1 #include <QApplication>
 2 #include <QHBoxLayout>
 3 #include <QSlider>
 4 #include <QSpinBox>

 5 int main(int argc, char *argv[])
 6 {
 7     QApplication app(argc, argv);

 8     QWidget *window = new QWidget;
 9     window->setWindowTitle("Enter Your Age");

10     QSpinBox *spinBox = new QSpinBox;
11     QSlider *slider = new QSlider(Qt::Horizontal);
12     spinBox->setRange(0, 130);
13     slider->setRange(0, 130);

14     QObject::connect(spinBox, SIGNAL(valueChanged(int)),
15                      slider, SLOT(setValue(int)));
16     QObject::connect(slider, SIGNAL(valueChanged(int)),
17                      spinBox, SLOT(setValue(int)));
18     spinBox->setValue(35);

19     QHBoxLayout *layout = new QHBoxLayout;
20     layout->addWidget(spinBox);
21     layout->addWidget(slider);
22     window->setLayout(layout);

23     window->show();

24     return app.exec();
25 }
```

Resultado visual:

Figura 5-5: Exemplo de *Layout* codificado

Fonte: Adaptado de (BLANCHETTE, 2008, p. 22)

A ferramenta *Designer* trabalha diretamente sobre os arquivos *Form*. Assim, todas as modificações feitas na janela e em seus elementos geram um código *XML* automaticamente que guarda todas as informações referentes a posicionamento e atributos de todos esses *widgets*. Além de trabalhar com o *design*, o *Designer* também permite definir comportamento e a comunicação entre os elementos por meio de uma estrutura única do Qt denominada *Signals and Slots*.

5.1.3 *Signals and Slots*

Signals and Slots é uma funcionalidade específica do Qt que tem como objetivo facilitar a comunicação entre diferentes objetos. Neste sistema, um *signal* representa uma mensagem que pode ser enviada por um objeto e *slot* representa um comportamento a ser executado em resposta a um dado *signal*. Um objeto pode ter vários *signals* e vários *slots* e pode estabelecer qualquer combinação de conexões com outros objetos (BLANCHETTE, 2008, p. 38).

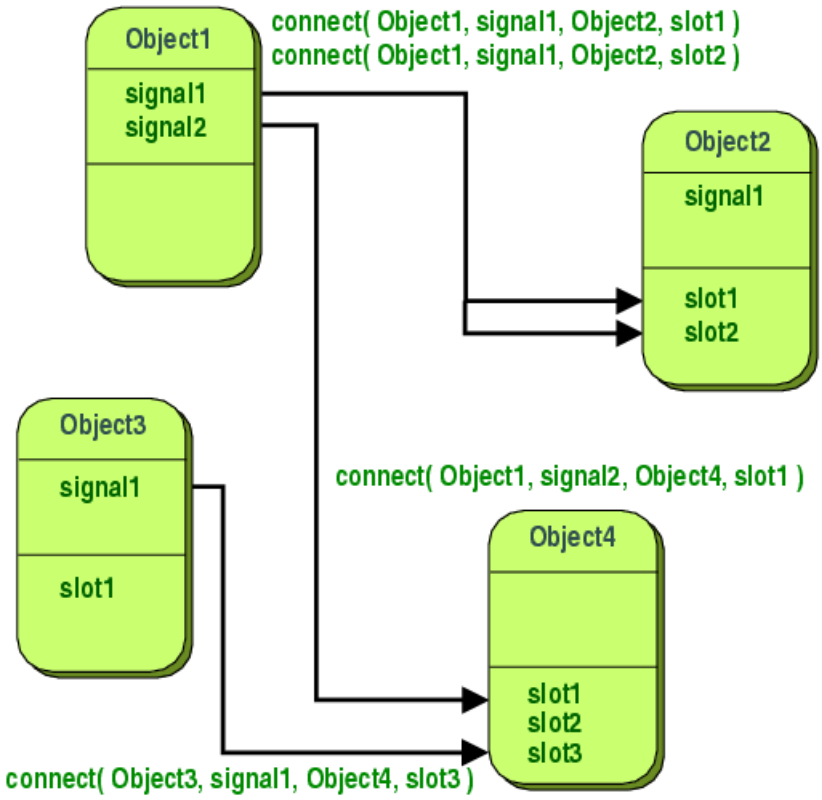


Figura 5-6: Sistema *Signals & Slots*

O *Slot* é, basicamente, uma função C++. Isso permite que o *Slot* possa ser uma função virtual, pública, privada, sobrecarregada, entre outras propriedades padrão de uma função C++, de modo que, a única característica que de fato a torna um *Slot* é a possibilidade de poder ser conectada a um *Signal* (BLANCHETTE, 2008, p. 38). O uso de *Signals and Slots* é permitido apenas para objetos que herdarem de *QObject* ou uma de suas subclasses (BLANCHETTE, 2008, p. 39).

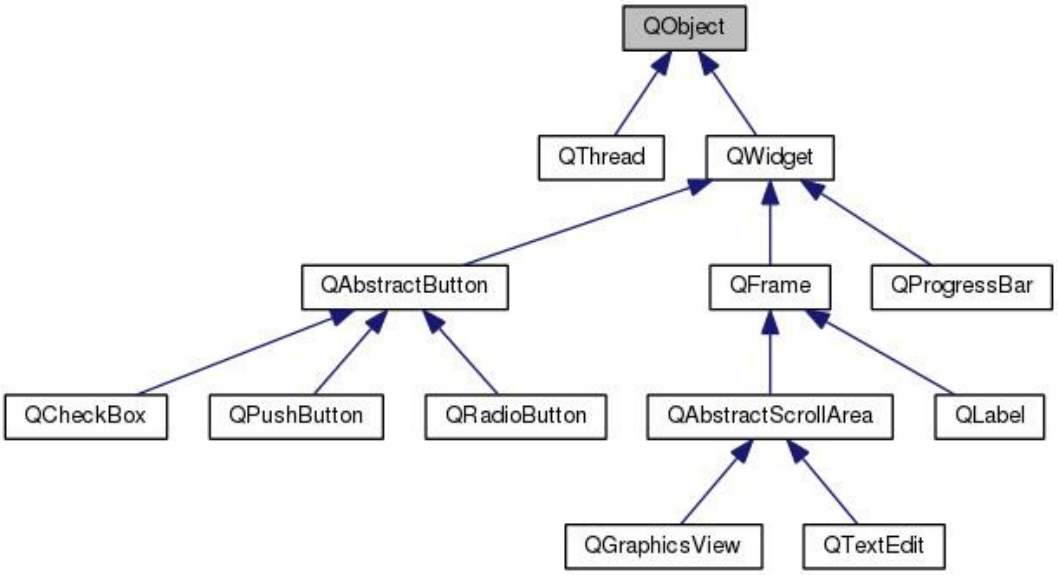


Figura 5-7: Exemplo de subclasses do *QObject*

A existência do sistema de *Signals and Slots* só foi viabilizada devido ao sistema *Meta-Object* do Qt. O mecanismo *Meta-Object* permite criar componentes de *software* independentes capazes de conversar entre si, mesmo sem se conhecerem. Com base nisso, o Qt possui uma ferramenta chamada *moc* que implementa, processa e disponibiliza todas essas "informações meta" por meio de funções C++, permitindo seu funcionamento com qualquer compilador C++ (BLANCHETTE, 2008, p. 39).

O uso de *Slots* pode ser feito a partir do reaproveitamento de um *Slot* existente da subclasse de *QObject* que estiver sendo utilizada ou por meio da própria codificação de um *Slot* customizado. Abaixo segue um exemplo simples de conexão.

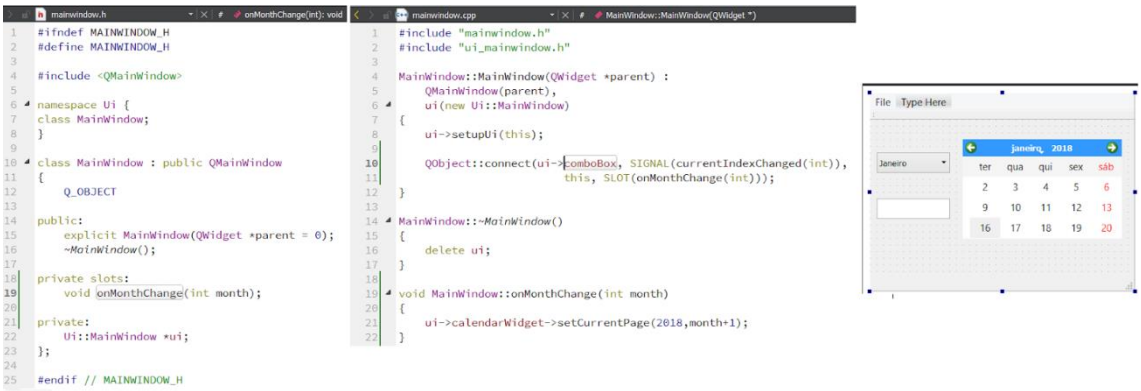


Figura 5-8: Exemplo de conexão entre um *signal* e um *slot*

Nesse exemplo, é feita a conexão de um dos *signals* existentes do *QComboBox* da janela, mais precisamente o *currentIndexChange*, que envia um argumento do tipo *int*, com um *slot* customizado da classe *MainWindow*, o *onMonthChange*, que recebe um argumento do tipo *int*. Essa conexão criará o seguinte comportamento na interface: toda vez que o usuário trocar o valor atual da *QComboBox*, o calendário terá o seu mês atual atualizado de acordo com o mês atual selecionado na *QComboBox*.

Outros *frameworks* também implementam a comunicação entre objetos, mas por meio de *callbacks*. Entretanto, o uso de *callbacks* pode apresentar os seguintes problemas:

- ***Callbacks* não são *type-safe***: não há garantia que o processo executor do *callback* entregará os argumentos de forma correta;
- **Dependência**: O processo executor de um *callback* precisa conhecer o *callback a priori* para poder executá-lo.

O sistema de *Signals and Slots* se mostra como uma alternativa robusta ao uso de *callbacks*, pois, pela sua estrutura, consegue eliminar os problemas existentes em *callbacks*. Para conectar um *signal* a um *slot*, é necessário que a assinatura de ambos sejam correspondentes. Uma vez tendo as assinaturas compatíveis, o compilador será capaz de detectar se há ou não incompatibilidade de tipo. Além disso, o objeto que emite um *signal* não precisa saber qual objeto ou *slot* receberá tal *signal* (THE QT COMPANY LTD, 2016).

Além de poder criar diretamente no código, é possível construir conexões dentro do próprio *Designer* utilizando o editor de *Signals and Slots*, que pode ser identificado na figura 5-4. Esse editor identifica todos os objetos presentes que possuem *Signals* e/ou *Slots* e organiza a estrutura da conexão de acordo com os argumentos necessários, bem como disponibiliza os *Slots* com assinaturas compatíveis com o *Signal* escolhido. A vantagem de usar esse editor é que todas as conexões mapeadas ficam organizadas em um mesmo lugar e não poluem o código fonte com chamadas de criação de conexões, pois elas ficam armazenadas no arquivo *Form* correspondente a janela em edição.

5.2 XML e Forms

O *XML* (eXtensible Markup Language) é uma linguagem de marcação criada para ser usada de forma universal pela internet, de forma que seu formato fosse de fácil

interpretação tanto para máquinas quanto para computadores. A partir de um conjunto de regras para codificação, o *XML* fornece uma sintaxe baseada em *tags* para estruturar seus dados e poder aplicar as marcações em seus documentos (SKONNARD, 2001, p. 1).

As *tags* a serem usadas em um arquivo *XML* podem variar de acordo com o projeto a ser desenvolvido (SKONNARD, 2001, p. 2). Os arquivos *form* do Qt, escritos em *XML*, utilizam um conjunto de *tags* criados com relação aos nomes dos componentes existentes.

mainwindow.ui

```
<?xml version="1.0" encoding="UTF-8"?>
<ui version="4.0">
 <class>MainWindow</class>
 <widget class="QMainWindow" name="MainWindow">
  <property name="geometry">
   <rect>
    <x>0</x>
    <y>0</y>
    <width>400</width>
    <height>300</height>
   </rect>
  </property>
  <property name="windowTitle">
   <string>MainWindow</string>
  </property>
  <widget class="QWidget" name="centralWidget"/>
  <widget class="QMenuBar" name="menuBar">
   <property name="geometry">
    <rect>
     <x>0</x>
     <y>0</y>
     <width>400</width>
     <height>26</height>
    </rect>
   </property>
  </widget>
  <widget class="QToolBar" name="mainToolBar">
   <attribute name="toolBarArea">
    <enum>TopToolBarArea</enum>
   </attribute>
   <attribute name="toolBarBreak">
    <bool>false</bool>
   </attribute>
  </widget>
  <widget class="QStatusBar" name="statusBar"/>
 </widget>
 <layoutdefault spacing="6" margin="11"/>
 <resources/>
 <connections/>
</ui>
```

Figura 5-9: Exemplo de arquivo *Form* do Qt

No exemplo acima, pode-se observar algumas *tags* padrão do Qt como, por exemplo, a <*widget*>, <*attribute*> e <*property*>. A *tag* <*ui*> é a de maior nível hierárquico, sendo a responsável por encapsular a janela em edição e todos os elementos gráficos adicionados a essa. Todas as *tags* correspondentes a um objeto gráfico

(<*widget*>) apresentam atributos como “class” e “name” para poder identificar e diferenciar objetos que tenham atributos em comum, permitindo o uso simultâneo de vários objetos de um mesmo tipo em uma mesma janela.

Outras *tags* como, por exemplo, <*resources*> e <*connections*> estão sempre presentes, mas nem sempre são utilizadas. Neste caso, elas estão nas formas <*resources/*> e <*connections/*>, pois não definiu-se nenhuma conexão através do editor de *Signals and Slots* e não há nenhum arquivo *resource* sendo usado no projeto do exemplo.

A partir da leitura de um arquivo *form*, o *Designer* consegue reconstruir automaticamente toda a janela e seus componentes. A edição de arquivos *form* pelo Qt só é permitida por meio do *Designer*, visto que qualquer edição direta no *XML* pode comprometer o funcionamento integrado entre os arquivos *header* e *source* correspondentes do arquivo *form*.

A vantagem de se utilizar arquivos *XML* para a construção de interfaces é que, além da possibilidade de visualizar o resultado gráfico antes da execução do *software*, ele “enxuga” o código principal da aplicação. No caso do Qt, por exemplo, definir intervalos de valores, tamanho de janelas, valores iniciais, *layout* e conexões e guardar todas esses dados no *XML* deixa o código em C++ mais voltado para funcionalidades mais matemáticas e computacionais, ajudando na organização e manutenibilidade do *software*.

5.3 The Visualization Toolkit (VTK)

O *VTK* (*Visualization Toolkit*) é uma biblioteca C++ *open source* voltada para a computação gráfica. Além de implementar diversos algoritmos para visualização 2D e 3D e processamento de imagem, ela é multiplataforma e pode ser integrada a ferramentas de desenvolvimento de interface gráfica como, por exemplo, o Qt (SCHROEDER, 2006, Preface).

Para esta interface, um dos objetivos era construir 2 gráficos referentes a execução do *FWI*. Um iria exibir as informações de convergência do algoritmo para apenas uma única frequência, enquanto o outro iria exibir as informações de convergência para todas as frequências definidas pelo usuário. Em ambos os casos, os dados seriam

apresentados por meio de um gráfico de linhas, que permite a fácil identificação do comportamento convergente ou divergente da aplicação.

O *VTK* foi utilizado para construir ambos os gráficos. Essa ferramenta foi escolhida, pois podia ser integrada ao Qt por meio do *widget QVTKWidget* e, além disso, ela oferece a possibilidade de se trabalhar com visualização 3D, que não foi explorado neste trabalho, mas seria uma funcionalidade a ser adicionada nesta interface posteriormente.

Para desenhar um gráfico de linhas em *VTK*, alguns objetos *vtk* foram necessários, como *vtkContextView*, *vtkChartXY* e *vtkPlotLine*, que podem ser descritos da seguinte forma:

- ***vtkContextView***: Objeto responsável por definir um contexto de desenho para a *QVTKWidget* onde será desenhada a curva de pontos;
- ***vtkChartXY***: Objeto responsável por desenhar um gráfico de dois (02) eixos utilizando o sistema de coordenadas cartesiano;
- ***vtkPlotLine***: Objeto responsável por desenhar uma linha, armazenando todos os pontos pelos quais a linha deverá passar;

Esses elementos também possuem outras propriedades gráficas para facilitar a exibição de informações extras como, por exemplo, legenda do gráfico, cor da linha de pontos, formato dos pontos, tamanho do intervalo de valor de um eixo, entre outros. Segue abaixo dois exemplos de construção de um gráfico em *VTK*.

```
285         //================================================================================
286         //CONFIGURANDO O CHART DAS RODADAS DAS FREQUENCIAS
287         //================================================================================
288         //instanciando um contextview para cada qvtkwidget
289         this->ContextViewChart = vtkSmartPointer<vtkContextView>::New();
290
291         // Set up a 2D scene, add an XY chart to it
292         this->ContextViewChart->GetRenderer()->SetBackground(this->rgb, this->rgb, this->rgb);
293         //this->ContextView->GetRenderWindow()->SetSize(400, 300);
294
295         //Instanciando um grafico o QVTKWidget dos Steps de Frequencia
296         this->ChartSteps = vtkSmartPointer<vtkChartXY>::New();
297         this->ContextViewChart->GetScene()->AddItem(this->ChartSteps);
298         this->ChartSteps->SetShowLegend(true);
299
300
301         //Configurando o plot que vai receber a tabela de dados
302         this->PlotStep = vtkSmartPointer<vtkPlotLine>::New();
303         this->ChartSteps->AddPlot(this->PlotStep);
304         //cor da linha do grafico
305         this->PlotStep->SetColor(46,139,87,255);
306         this->PlotStep->SetWidth(1.0);
307         vtkPlotPoints::SafeDownCast(this->PlotStep)->SetMarkerStyle(vtkPlotPoints::DIAMOND);
308
309         //Renderizando a cena...
310
311         this->setDefaultChartStepsAxisConfigs();
```

Figura 5-10: Construção do gráfico de convergência para uma única frequência

```
337         //================================================================================
338         //CONFIGURANDO O CHART DAS RODADAS DAS FREQUENCIAS ACUMULADAS
339         //================================================================================
340         //instanciando um contextview para cada qvtkwidget
341         this->ContextViewChartAcumulate = vtkSmartPointer<vtkContextView>::New();
342
343         // Set up a 2D scene, add an XY chart to it
344         this->ContextViewChartAcumulate->GetRenderer()->SetBackground(this->rgb, this->rgb, this->rgb);
345         //this->ContextViewChartAcumulate->GetRenderWindow()->SetSize(400, 300);
346
347         //Instanciando um grafico o para os valores acumulados de frequencia
348         this->ChartAcumulate = vtkSmartPointer<vtkChartXY>::New();
349         this->ContextViewChartAcumulate->GetScene()->AddItem(this->ChartAcumulate);
350         this->ChartAcumulate->SetShowLegend(true);
351
352         //Configurando o plot que vai receber a tabela de dados
353         this->PlotAcumulate = vtkSmartPointer<vtkPlotLine>::New();
354         this->ChartAcumulate->AddPlot(this->PlotAcumulate);
355         this->PlotAcumulate->SetColor(1.0, 0.0, 0.0);
356         this->PlotAcumulate->SetWidth(1.0);
357         vtkPlotPoints::SafeDownCast(this->PlotAcumulate)->SetMarkerStyle(vtkPlotPoints::DIAMOND);
358
359         //Renderizando a cena...
360
361         this->setDefaultChartAccumulateAxisConfigs();
```

Figura 5-11: Construção do gráfico de convergência acumulado das frequências

Os dados dos gráficos são obtidos a partir da leitura de um arquivo texto que é escrito pelo algoritmo *FWI* para cada frequência executada. O arquivo texto contém o valor de minimização de erro resultante referente a última iteração executada de sua respectiva frequência. Quando ocorre a mudança de frequência, o gráfico acumulado armazena os dados já obtidos e o gráfico de frequência individual é apagado para exibir os valores da próxima frequência.

A leitura do arquivo texto de uma dada frequência é realizada utilizando o *QFileInfo* e o *QFile*. *QFileInfo* é responsável por armazenar informações de um arquivo como o nome e o caminho dele dentro de um sistema de arquivos enquanto *QFile* é responsável por fazer a leitura e escrita de arquivos, que podem ser do tipo texto, binário ou *resources*. Assim, *QFileInfo* consegue, a partir de um dado caminho, identificar se esse caminho de fato existe e se ele corresponde a um arquivo ou diretório para, em seguida, *QFile* poder fazer a leitura desse arquivo.

```
507         if(fileInfo.exists())
508         {
509             QFile f(str);
510             cin.clear();
511 
512             if (f.open(QIODevice::ReadOnly | QIODevice::Text))
513             {
514                 QTextStream in(&f);
515                 in.setRealNumberPrecision(2);
516                 QString line;
517 
518                 line = in.readLine(256);
519                 stringstream stream;
520                 stream.clear();
521                 stream << (line.toAscii().constData());
522 
523                 //Pegando os valores do arquivo
524                 stream >> this->CurrentValue;
525 
526                 //Salvando os valores lidos do arquivo texto
527                 vtkSmartPointer<vtkVariantArray> va = vtkSmartPointer<vtkVariantArray>::New();
528                 va->InsertValue(0, this->CurrentValue);
529             }
530 
531             f.close();
532         }
533 
534         //Atualizando o Grafico de Frequencias
535         this->slotArrangeStepChart(this->idGraph);
536 }
```

Figura 5-12: Exemplo de código para leitura do arquivo texto

O trecho de código acima mostra como é feita a leitura do arquivo texto e a passagem do valor lido para o gráfico de convergência para uma única frequência. Não é necessário fazer um processamento do valor obtido, visto que tal processamento já é feito pelo algoritmo *FWI*. Após a leitura, o valor é adicionado ao vetor de valores utilizado para desenhar o gráfico de linha e atualiza-se o gráfico com o novo ponto adicionado.

5.4 Tratamento de dados de entrada

Uma das grandes vantagens de se utilizar uma interface gráfica é poder impedir o máximo de erros possíveis decorrentes da interação do usuário com o *software*. Para construir esses impedimentos, é fundamental observar e prever fraquezas do *software* a fim de elaborar regras de comportamento que guiem o uso do sistema. Para isso, antes de ir diretamente para a **codificação**, é imprescindível definir o passo-a-passo que deve ser feito pelo usuário, tanto sob a forma de requisitos quanto fluxogramas, para que se possa criar a maneira mais intuitiva de uso do programa.

Existem diferentes formas de abordar o usuário para indicar que uma determinada ação não pode ser feita e a escolha delas varia de acordo com a situação em que o usuário estiver navegando pelo sistema. Em certos casos, o travamento do *click* de um botão pode ser suficiente para indicar ao usuário que algum dado de entrada não é válido, prevenindo-o de um erro no *software*. Pode existir também casos em que, mesmo travando algum componente da interface, seja necessário explicar para o usuário o porquê do travamento e o que precisa ser feito para desbloquear e dar continuidade ao fluxo de navegação.

Neste trabalho, algumas janelas, além de terem suas funcionalidades básicas desenvolvidas, precisavam tratar a informação de entrada fornecida pelo usuário para garantir a correta execução do algoritmo *FWI*. Uma delas é a de criação e configuração de um projeto e a outra é a de edição de parâmetros.

A janela de criação e configuração de um projeto (Figura 4-1) é constituída pelos seguintes *widgets*: *QWidget, QGridLayout, QPushButton, QLabel, QLineEdit, Line* e *QProgressBar*. Desses elementos, apenas *QPushButton, QLineEdit* e *QProgressBar* foram utilizados para tratar valores de entrada.

As linhas editáveis *QLineEdit* dessa janela possuem comportamentos diferentes. Em “Project Name”, a linha envolta em vermelho não é editável, servindo apenas para mostrar o prefixo do nome da pasta de projeto a ser criada, e a linha ao lado permite que o usuário coloque o nome que desejar. As linhas restantes na janela também não são editáveis, pois elas servem apenas para mostrar o resultado das seleções feitas utilizando seus respectivos botões.

```
398 //
399 //Slot para detectar quando o usuario atualiza o nome do projeto ao pressinar alguma tecla do teclado
400 //
401 void NewFWIProjectWidget::on_projectNameLineEdit_textEdited(const QString &arg1)
402 {
403     this->setProjectFolderText(arg1);
404 }
```

Figura 5-13: Comportamento da linha editável para nomear pasta do projeto

Os botões *QPushButton* dessa janela possuem comportamentos diferentes. Os que possuem o nome “Browse” servem para navegar no sistema de arquivos do sistema operacional para selecionar uma pasta ou arquivos. O botão em “Project Location” permite apenas selecionar uma única pasta, o botão em “Initial Model” permite apenas escolher um único arquivo binário (modelo de velocidades), e o botão em “Sismogram” permite apenas selecionar arquivos binários, podendo ser um ou mais arquivos (sismograma(s)). O botão “Project Ready!” cria a estrutura de diretórios com os arquivos selecionados apenas se houver pelo menos 1 modelo de velocidades e 1 sismograma selecionados e se houver uma pasta selecionada. O botão “cancel” serve para fechar a janela.

```
110 void NewFWIProjectWidget::on_browseLocationPushButton_clicked()
111 {
112     QString filename = QFileDialog::getExistingDirectory(this, tr("Choose Directory to Create Project"),
113                                                          QDir::homePath(),
114                                                          QFileDialog::ShowDirsOnly | QFileDialog::DontResolveSymlinks);
115
116     //Quando o usuario nao cancelar ou fechar a caixa de dialogo, nos podemos atualizar os campos "Project Location" e "Project Folder"
117     if(filename != "")
118     {
119         ui->projectLocationLineEdit->setText(filename+QDir::separator());
120         this->setProjectFolderText(ui->projectNameLineEdit->text());
121     }
122
123     //qDebug() << "NewFWIProjectWidget: Project Location chosen is:  " << filename;
124 }
```

Figura 5-14: Exemplo do comportamento do botão “browse” para selecionar a pasta do projeto

A barra de progresso *QProgressBar* foi utilizada com o objetivo de indicar ao usuário o tempo decorrido e o tempo restante do processo de criação da estrutura de diretórios e do processo de cópia do modelo de velocidades e do(s) sismograma(s) para dentro da estrutura criada.

```
251             //Criando o FolderSystem com os arquivos a serem copiados
252             this->folderSystem = new FolderSystem(ui->projectLocationLineEdit->text(), ui->projectFolderLineEdit->text(), this,
253                                                   ui->progressBar->maximum()-ui->progressBar->minimum());
254
255             //*********************************************************************************************************************//
256             //                                          Connects que precisam que um FolderSystem esteja previamente definido
257             //*********************************************************************************************************************//
258             //Caso o usuario escolha "Yes" para criar um FolderSystem
259             connect(this->folderSystem, SIGNAL(signalCreatingFolderSystem()), this, SLOT(slotCreatingFolderSystem()));
260
261             //Para cada arquivo que for copiado, atualizaremos a barra de progresso
262             connect(this->folderSystem, SIGNAL(fileCopied(double)), this, SLOT(slotUpdateProgressBar(double)));
263
264             //Quando todos os arquivos forem copiados, o usuario sera alertado que o projeto foi criado com sucesso
265             //e essa janela sera fechada
266             connect(this->folderSystem, SIGNAL(finishedCopying()), this, SLOT(slotFinishedCopying()));
267
268             //Depois que todos os connects acima foram estabelecidos, nos podemos criar a pasta do projeto
269             this->folderSystem->createFolderSystem(this->filesToBeCopied);
```

Figura 5-15: Utilização da barra de progresso durante a criação da estrutura de diretórios com cópia de arquivos

A janela de edição de parâmetros (Figura 4-3) é constituída pelos seguintes *widgets*: *QWidget*, *QPushButton*, *QLabel* e *QTableWidget*. Desses elementos, apenas *QPushButton* e *QTableWidget* foram utilizados para tratar valores de entrada.

Os botões *QPushButton* dessa janela possuem comportamentos diferentes. O botão “cancel” serve apenas para fechar a janela, enquanto o botão “Finish Editing” além de fechar a janela, é responsável por pegar todos os valores inseridos na tabela e escrever o arquivos de parâmetros que servirá como dado de entrada para o algoritmo *FWI*.

```
514 //
515 //Metodo para detectar quando o usuario pressionou o botao "finish editing"
516 //
517 void ParametersEditorWidget::on_finishEditingPushButton_clicked()
518 {
519     //Setando os valores dos parametros na janela com a tabela de parametros
520     this->setParametersTableValues();
521
522     //Escrevendo o arquivo de parametros
523     this->writeParametersFile();
524
525     //Escondendo essa janela
526     this->hide();
527
528     //Signal para notificar que essa janela foi fechada
529     emit this->signalWindowClosed();
530
531     //Signal para notificar que o botao de "finish editing" foi presionado
532     emit this->signalFinishEditingButtonClicked();
533 }
```

Figura 5-16: Comportamento do botão “Finish Editing”

A tabela *QTableWidget* tem como objetivo mostrar e definir valores para todos os parâmetros necessários para poder executar o algoritmo *FWI*. Em função do caso de

execução demonstrado no capítulo anterior, ela sempre apresenta os mesmos valores iniciais dos parâmetros ao ser inicializada. Das 4 colunas que possui, apenas a coluna “Value” é editável, pois as outras servem apenas para dar informações para o usuário. Nessa coluna editável, existem alguns parâmetros que só podem ser números e outros que só podem ser texto, sendo necessário verificar se o que o usuário coloca de um parâmetro é compatível com o seu tipo esperado. Essa verificação ocorre no *Slot* customizado *on_parametersTable_cellChanged*, como pode ser visto abaixo.

```
476  //
477  //Metodo para detectar quando o usuario atualizou alguma celula da tabela
478  //
479  void ParametersEditorWidget::on_parametersTable_cellChanged(int row, int column)
480  {
481      QTableWidget *table = ui->parametersTable;
482      QString paramName = table->model()->index(row,0).data().toString();
483      RowData::ParameterType rowType = getParameterType(paramName);
484
485      if(rowType==RowData::STRING)
486      {
487          //qDebug() << paramName << "is a string";
488          this->checkStringValidity(row);
489      }
490      else if(rowType==RowData::INTEGER)
491      {
492          //qDebug() << paramName << "is an integer";
493          this->checkIntegerValidity(row);
494      }
495      else if(rowType==RowData::FLOAT)
496      {
497          //qDebug() << paramName << "is a float";
498          this->checkFloatValidity(row);
499      }
500  }
```

Figura 5-17: Comportamento da tabela de edição de parâmetros

Todas essas chamadas de verificação do valor de entrada fornecido pelo usuário apresentam a mesma lógica: se o valor fornecido for válido dado o parâmetro editado, o novo valor é salvo na tabela, caso contrário o valor antigo permanece na tabela.

```
260 //
261 //Metodo para verifcar se o input do usuario eh um int
262 //
263 void ParametersEditorWidget::checkIntegerValidity(int row)
264 {
265     QTableWidget *table = this->ui->parametersTable;
266
267     QString currentParameter = table->model()->index(row,0).data().toString();
268     int newValue = table->model()->index(row,1).data().toInt(); //this returns 0 if the input was a float or text or negative value
269
270     if(newValue > 0)
271     {
272         this->setParameterValue(currentParameter, QString::number(newValue));
273     }
274     else
275     {
276         table->setItem(row,1, new QTableWidgetItem(getParameterValue(currentParameter)));
277     }
278
279     //qDebug() << currentParameter << " value is: " << getParameterValue(currentParameter);
280 }
```

Figura 5-18: Lógica de verificação de valor de entrada para o caso de um número inteiro

5.5 Documentação e Versionamento

A **Documentação** é o processo de escrita de todas as características do *software*, explicando desde sua configuração do ambiente de desenvolvimento até a explicação de como utilizar o sistema. Essa é uma atividade importante, visto que durante o processo de desenvolvimento, com a adição ou remoção de funcionalidades e atualização dos requisitos, bibliotecas e/ou ferramentas podem deixar de ser integrantes da solução como um todo.

Por este trabalho ser o primeiro projeto de desenvolvimento de *software* da empresa, não existia um processo formal previamente elaborado para documentação de *softwares* em geral. Diante disso, ficou acordado que a documentação seria toda construída diretamente no código fonte enquanto se buscava uma solução que gerasse documentos descritivos de código de forma automática.

Caso novos desenvolvedores viessem a contribuir para o projeto, haveria necessidade de um treinamento para entendimento do funcionamento do sistema em geral, mas o código estaria legível e bem descrito. Dessa forma, a ideia era que o colaborador pudesse trabalhar em cima do código facilmente, o mais rápido e independente possível do(s) outro(s) desenvolvedor(es).

Outra atividade também muito importante durante o processo de desenvolvimento de *software* é o uso de um **Sistema de Controle de Versões** (VCS). Criado com o intuito de gerenciar diferentes versões de um documento qualquer, o VCS é altamente recomendado durante o desenvolvimento de *software*, pois proporciona uma maior consistência dos arquivos, servindo como uma cópia de segurança dos mesmos.

Neste trabalho também foi necessário definir um Sistema de Controle de Versão a ser usado antes do início da implementação do código. Dentre as diversas opções existentes, o **Git** foi escolhido por conta da minha experiência prévia com essa ferramenta, não sendo necessária uma alocação de tempo para treinamento/aprendizado, além de ser multiplataforma e *open source*, não adicionando custos financeiros ao projeto.

Em relação ao serviço de hospedagem do projeto, escolheu-se o **Bitbucket**, pois viabiliza o uso de repositórios baseados em **Git** e possui uma opção de serviço grátis com repositórios privados com capacidade de armazenamento de até 2 GB de arquivos. Assim, o acesso ao repositório é restrito apenas a quem estiver desenvolvendo a aplicação ou quem tiver permissão de visualizar o código fonte da mesma.

No sistema **Git**, a estrutura de diretórios do projeto em desenvolvimento é denominada de **repositório**, no qual guarda e organiza todas as alterações feitas em seus documentos por meio de **revisões**, que funcionam analogamente aos estados de uma máquina de estados. Cada revisão possui um conjunto de alterações feitas em um ou mais documentos em relação a última revisão registrada e pode ser criada utilizando-se como base a data de registro (*commit*) das novas alterações, criando-se, assim, um histórico de desenvolvimento do *software*.

A criação de vários *commits* descrevendo o passo-a-passo do que foi implementado é crucial para situações em que se busca um trecho código que possa ter inserido algum *bug* no *software* ou mesmo quando se procura a metodologia de implementação por trás de uma determinada funcionalidade existente. Ademais, esse sistema permite também que múltiplos programadores possam trabalhar simultaneamente nos mesmos arquivos de forma independente.

Para este trabalho, por conta de ter sido implementado, até esta versão da interface, por apenas uma pessoa, a estrutura utilizada de *branchs* do Git foi bem simples, contando apenas com o *master*, que sempre continha a última versão mais estável da interface gráfica. *Branch* pode ser definido como uma cópia de um conjunto de arquivos, que pode ter sua própria lista de *commits*. Um projeto pode possuir vários *branches* a depender de sua complexidade, e esses *branches* existem de forma paralela e independente. Segue abaixo uma figura ilustrando os arquivos presentes no repositório usado para este trabalho.

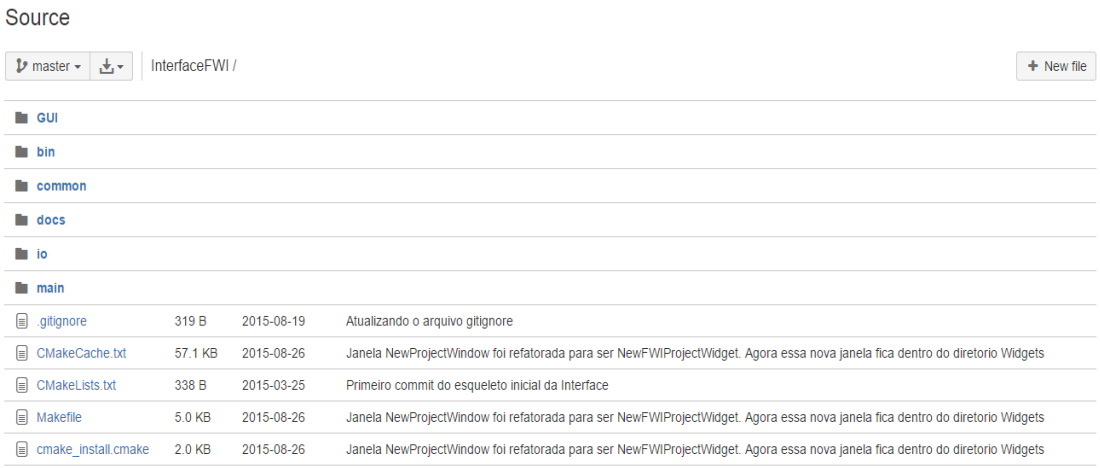


Figura 5-19: Estrutura de diretórios do projeto armazenado no *Bitbucket*

Capítulo 6
Considerações Finais

6.1 Conclusão

Diante da proposta de se desenvolver um *software* do zero de maneira rápida e que atendesse a todas as necessidades da empresa, a escolha de se utilizar uma interface gráfica se mostrou uma excelente estratégia. A utilização da linguagem c++ em conjunto com a ferramenta de desenvolvimento Qt (versão 5.4) permite criar aplicações capazes de trabalhar em diferentes sistemas operacionais, além de possuir bibliotecas que entregam soluções para diversas áreas como, por exemplo, visualização gráfica, redes, banco de dados, entre outras.

A gama de opções e a disposição de informações fornecidas pelo Qt tornam o processo de modelagem de *GUIs* prático, escalável e facilmente manutenível. Adicionar novos componentes, integrar ou reaproveitar funcionalidades entre diferentes objetos proporcionam ao desenvolvedor liberdade e agilidade no processo de codificação, entregando uma robusta estrutura de dados. Essa facilidade de modelagem também simplifica a etapa de transposição da conceitualização (*design*) do produto e de suas funcionalidades em uma aplicação concreta e usável.

A escolha pela linguagem c++, amplamente utilizada no meio acadêmico e em projetos comerciais, teve como principais motivos o seu grande desempenho e compatibilidade com os *softwares* proprietários da empresa. Com ela é possível elaborar aplicações que se baseiam em um grande volume de cálculos e simulações, ou seja, demandam muitos recursos computacionais, entregando resultados de maneira rápida, eficiente e satisfatória.

Este projeto foi um passo muito importante para a empresa, pois, por meio dele, pôde-se mapear processos de desenvolvimento que existiam e deveriam ser criados e/ou melhorados para poder atender as necessidades internas e externas da mesma. Deste modo, os objetivos traçados para este projeto foram cumpridos com êxito.

6.2 Trabalhos Futuros

Todo o código e artefatos produzidos paralelamente no decorrer do desenvolvimento deste trabalho permitiram a execução do caso de teste abordado no final do capítulo 4 (seção 4.2). Porém, o algoritmo de *FWI*, implementado na linguagem Fortran, está sendo executado a partir de um comando *shell*, impossibilitando a compatibilidade dessa aplicação em computadores que usam *Windows*.

Uma possível solução para contornar esse problema seria criar uma comunicação direta entre a interface gráfica e o código fortran, que poderia ser feito transformando o algoritmo *FWI* em uma biblioteca estática e incorporando ao código da interface. Uma segunda solução seria portar todo o código em Fortran para linguagem c++, o que está sujeito a disponibilidade de tempo, recursos financeiros e humanos.

Outra funcionalidade para a interface gráfica seria capacitá-la para executar outros algoritmos de sísmica além do *FWI*. Com isso, a interface se tornaria modular, apresentando diferentes ferramentas de visualização e análise de dados de acordo com o algoritmo escolhido. Em função desta natureza modular e da dependência dos algoritmos a serem agregados, para que essa tarefa possa ser iniciada, é necessário, primeiro, definir qual seria a solução para a comunicação entre a interface e o algoritmo *FWI*.

Referências bibliográficas

RIZVANDI, N. B., BOLOORI, A. J., KAMYABPOUR, N., et al., 2011, "MapReduce Implementation of Prestack Kirchhoff Time Migration (PKTM) on Seismic Data"., pp. 86-91.

PFLEEGER, S. L., 2004, ***Engenharia de software: teoria e prática***. 2 ed. São Paulo, Prentice Hall.

PREECE, J., ROGERS, Y., SHARP, H., 2005, ***Design de Interação: além da interação homem-computador***. Porto Alegre, Bookman.

CHILLAREGE, Ram *et* al. Orthogonal defect classification: a concept for in-process measurements. *IEEE Transactions on Software* Engineering, v. 18, n. 11, p. 943-956. Nov. 1992.

GENTIL, B., 2008, *Estudo de usabilidade de ambientes virtuais tridimensionais através do Second* life. Dissertação de M.Sc., Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, Rio de Janeiro, Brasil.

PREECE, J. *et* al., 1994, ***Human-Computer*** Interaction. Addison-Wesley.

LINDSAY, P., NORMAN, D. A., 1977, ***Human Information Processing: Na Introduction to*** Psychology. 2 ed. Nova Iorque, Academic Press.

NORMAN, D. A., 1999, ***The Invisible Computer***. Cambridge, Massachusetts: MIT.

LAW, E. L. C. *et al*., 2009, ***Understanding, scoping and defining user experience: a survey approach***. In Proceedings of the SIGCHI conference on human factors in computing systems (pp. 719-728). ACM.

TARANTOLA, A., 1984, Inversion of seismic reflection data in the acoustic approximation. Geophysics, **49**(8):1259-1266.

VIRIEUX, J., OPERTO, S., 2009, An overview of full-waveform inversion in exploration geophysics. Geophysics, **74**(6):WCC1-WCC26.

MIRANDA, L. G., 2011, *Implementação de algoritmo paralelo para inversão de dados* geofísicos. Trabalho de Graduação, Universidade Federal da Bahia, Salvador, Bahia, Brasil.

SANTOS, A. W. G., 2013, *Inversão de forma de onda aplicada à análise de velocidades sísmicas utilizando uma abordagem multiescala*. Dissertação de M.Sc., Universidade Federal da Bahia, Salvador, Bahia, Brasil.

BLANCHETTE, J., SUMMERFIELD, M., 2008, ***C++ GUI Programming with Qt 4.*** 2 ed. Massachusetts, Prentice Hall.

SKONNARD, A., GUDGIN, M., 2001, ***Essential XML Quick Reference: a programmers's reference to XML, XPath, XSLT, XML Schema, SOAP, and More.*** 1 ed. Boston, Pearson Education.

THE QT COMPANY LTD., 2016. Disponível em <http://doc.qt.io/archives/qt-4.8/signalsandslots.html>. Acessado em: 30 Junho 2018.

SCHROEDER, W., MARTIN, K., LORENSEN, B., 2006, ***The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics.*** 4 ed. Kitware.

KITWAREPUBLIC, 2014. Disponível em <https://www.vtk.org/Wiki/VTK/Examples/Cxx/Plotting/LinePlot>. Acessado em: 01 Julho 2018.