Pós-processador para modelo de macroelementos finitos

Bruna Sayuri de Souza Suzuki, Orientador: Rodrigo Provasi Correia

Resumo – O projeto de tubos flexíveis e cabos umbilicais é um processo complicado. Para se fazer todas as validações necessárias, tanto a mecânica global quanto a interna devem ser levadas em conta. Apesar do modelo mostrar grandes avanços, a entrada e a saída de dados ainda são feitas utilizando arquivos de texto e ferramentas comerciais, que tornam o processo demorado e suscetível a falhas. Para tal, são propostas ferramentas de pré e pós-processamento que deixam tal tarefa mais simples. Essa pesquisa trata especificamente da ferramenta de pós-processamento. Objetiva-se construir uma ferramenta utilizando a linguagem C#, integrada ao modelo, que permita a geração de gráficos que facilitem a compreensão do fenômeno. Busca-se uma ferramenta intuitiva e que seja fácil de ser utilizada.

Palavras-chave –Macroelementos finitos; C#; Pós-processador.

1 Introdução

O problema de mecânica interna de tubos flexíveis e cabos umbilicais é um grande desafio na área. Existe um grande esforço da comunidade em encontrar maneiras melhores e mais eficazes de representar esses tubos com maior fidelidade. Sendo assim, modelos que facilitem os cálculos são extremamente desejáveis.

Mesmo com bons modelos, ferramentas que auxiliem no processo de criação e no de verificação de resultados mostram-se de grande valia, já que representam ganhos expressivos na aplicabilidade e produtividade.

Esse projeto é parte de um esforço maior para melhoras às ferramentas e modelos na área de mecânica interna e que é composto de um pré-processador, que facilita a geração e a descrição dos casos; um modelo de mecânica interna chamado de macroelementos finitos por ser composto de macroelementos (que são elementos que levam em conta características geométricas na formulação, descritos com mais detalhes, na qual a formulação fica mais complicada porém minimizam-se o número de elementos necessários para simular o problema como um todo); e um pós-processador, responsável por tratar das saídas e fornecer de maneiras simples e elegante as saídas desejadas.

Atualmente, o pós-processamento é feito através da geração de arquivos de texto em um formato pré-definido e, com auxílio de um programa de planilhas, colocados em forma gráfica que facilite sua interpretação. Para casos simples, isso não se mostra problemático, porém, para casos em que o elemento que utiliza série de Fourier é utilizado (e precisa-se montar todo o campo baseado nas amplitudes de cada termo) ou quando tem-se mais de alguns poucos tendões (no caso real há por volta de 100), executar isso manualmente torna-se inviável.

2 Objetivos

Com base no que foi apresentado, objetiva-se com essa pesquisa a criação de uma ferramenta de pós-processamento para o modelo de macroelementos finitos.

A ferramenta a ser desenvolvida deverá possuir saídas gráficas, permitir a visualização de vários resultados simultaneamente; ser fácil de utilizar e que, em caso de erro, providencie mensagens claras e elucidativas para que o mesmo seja corrigido.

Além disso, ela deverá ser desenvolvida utilizando a linguagem C# (a mesma na qual está o modelo). Sendo assim, a aprendizagem da linguagem C# e o entendimento do funcionamento de um

Versão inicial submetida em 17 de dezembro de 2018. Versão final aceita em 31 de Dez. de 2018. Publicado em 31 de Dez. de 2018. Digital Object Identifier 10.11606/issn.2526-8260.mecatrone.2018.152897

pós-processador, ou seja, como os dados podem ser tratados para serem disponibilizados de uma forma mais simples e de fácil visualização para o usuário, serão objetos a serem trabalhados nesse projeto.

3 Metodologia

As etapas da pesquisa foram executadas seguindo-se uma ordem. Abaixo cada etapa será detalhada.

3.1 Estudo da linguagem C# e sua biblioteca de funções para interface

A pesquisa iniciou-se com a adaptação à linguagem C#. Em (SHARP, 2013) foi possível obter uma panorâmica geral sobre a linguagem. Foi escolhido o Visual Studio 2017 – importante ambiente de desenvolvimento integrado (IDE) da Microsoft para desenvolvimento de *software* – para o desenvolvimento da pesquisa.

Devido ao fato de C# ser uma linguagem orientada a objetos, houve um período de familiarização com esse tipo de programação. Para tal, foram estudados elementos fundamentais da linguagem, tais como classes, encapsulamento, herança, polimorfismo, sobrecarga, conversão de tipos de dados, tratamento de exceções e os *generics*.

As classes são indispensáveis na orientação a objetos. É por meio delas que a declaração de objetos pode ser realizada. Dentro da classe, são definidos os atributos do objeto (que são variáveis que contêm informação). É possível defini-los com o uso de propriedades em vez de simples variáveis públicas, pois com as propriedades pode-se controlar o acesso às informações da classe. Para tal, utiliza-se *get* e *set* para declarar uma propriedade.

Com a finalidade de trabalhar com as classes criadas, podem ser definidos métodos que realizam as operações dos objetos. Um método especial é o construtor, que é responsável pela criação dos objetos. Nos métodos, um jeito de tornar o programa mais modularizado é por meio do encapsulamento, que consiste em esconder os detalhes de implementação de uma classe, isolando as partes do código. É possível declarar níveis de acessibilidade diferentes utilizando os modificadores de acesso para os métodos: *public, protected, internal* e *private.*

Além disso, para que seja possível utilizar uma classe dentro de uma aplicação, é necessário instanciá-la na memória (ou seja, alocar memória suficiente para guardar as informações da classe), sendo necessário utilizar um operador *new*. O *new* retorna uma referência que aponta para o objeto na memória, a qual será utilizada para manipularmos a classe criada. Essa classe criada é denominada de instância ou objeto.

Quando uma classe pode ser construída utilizando-se outras classes, é possível fazê-la herdar códigos. Essa funcionalidade chama-se herança, e é uma boa alternativa de reaproveitar código evitando repetições desnecessárias. Para fazer isso, basta definir uma classe com : (dois pontos), resultando na herança de todos os atributos e métodos da outra classe. O polimorfismo é parecido com a herança, porém nesse caso as palavras-chaves *virtual* e *override* são usadas para sobrescrever/substituir um método e a palavra-chave *new* é usada para ocultar um método. Isso faz com que também seja possível reaproveitar códigos de uma classe base. Outra técnica interessante é a da sobrecarga (*overload*), que consiste em permitir, dentro da mesma classe, mais de um método com o mesmo nome. Contudo, os métodos necessariamente devem possuir argumentos diferentes para que o programa principal faça a distinção entre eles no tempo de execução.

Em relação à conversão de tipos de dados, C# não permite que variáveis que já foram declaradas possam ser utilizadas para armazenar variáveis de outros tipos. Com isso, existem as formas implícita e explícita de conversão de tipos. O modo implícito utiliza estruturas já prontas do C#, convertendo tipos automaticamente. Um exemplo seria o uso do método *Convert.ToDouble( )* que converte um tipo em *double*, ou então utilizando o método *Double.Parse( )*. Já o método explícito, denominado *casting*, é utilizado quando não é possível realizar a conversão sem perda de dados (como, por exemplo, de *double* para *int*), sendo necessário implementar um pequeno bloco de código.

Com a finalidade de evitar conversões de tipo, principalmente as conversões com perda de

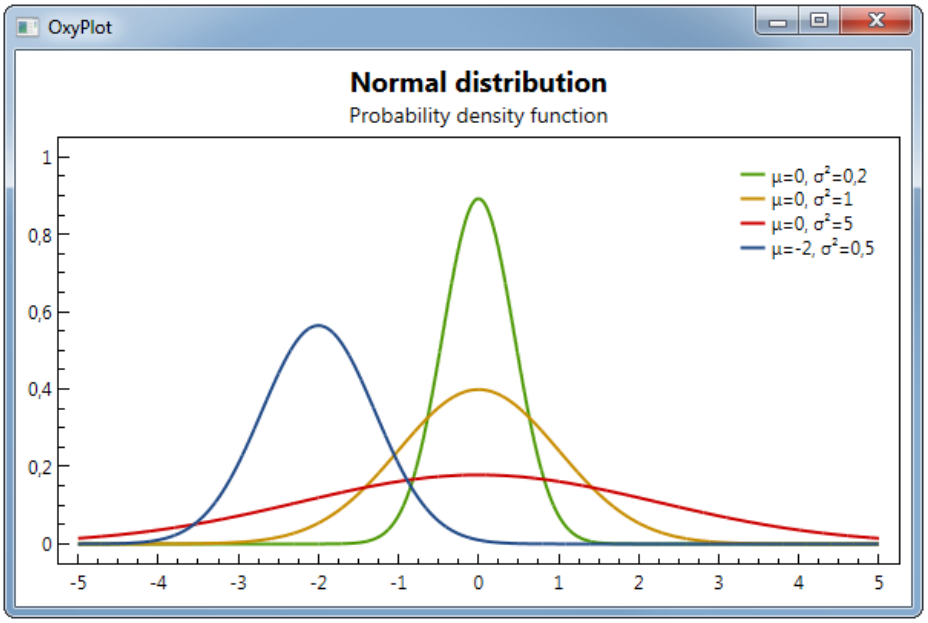
dados (*casting*), foi criada a classe *Generics*. Ela faz parte da biblioteca *System.Collections*. Com ela, é possível criar coleções de tipos variados, facilitando o uso dos dados e a organização geral do código. Os *Generics* são importantes pois oferecem uma tipagem segura, facilitam a criação de classes e métodos, diminuem a quantidade de *boxing* e eliminam o *casting*. Denomina-se *boxing* o ato de converter um tipo de valor (que contém diretamente os dados) em um tipo de referência (objetos), e *unboxing* consiste no processo inverso.

Outra questão importante em um código bem elaborado são as exceções (*exceptions*). Uma exceção é um erro em tempo de execução de um programa que viola uma condição que não foi especificada para acontecer durante a operação normal, parando imediatamente o programa de rodar. Dessa forma, é importante que haja um tratamento de exceções para que esse erro seja comunicado e tratado. Para tal, são utilizadas as declarações *throw*, *try*, *catch* e *finally*.

Em seguida, estudou-se o *Windows Presentation Foundation (WPF)*, subsistema gráfico no qual seria construída toda a interface gráfica da pesquisa. Essa ferramenta conta com uma linguagem de marcação baseada em XML chamada *Extensible Application Markup Language (XAML)*, que tornaria o trabalho de criar interfaces mais simples e rápido. A vantagem da utilização do WPF é que com ele é possível escrever programas misturando C# e XAML, sendo o XAML utilizado para definir o layout da aplicação.

Por fim, outra biblioteca fundamental para a realização dessa iniciação científica é o *OxyPlot*. O *OxyPlot* é uma biblioteca de plotagem de gráficos de fonte aberta sob a licença do MIT. Com ele, tem-se em mãos a ferramenta perfeita para a plotagem de gráficos de maneira a facilitar a visualização dos resultados finais para o usuário, como pode ser visto na Figura 1.

Figura 1 – Exemplo de gráfico plotado com a biblioteca *OxyPlot*



Fonte: Documentação do *OxyPlot*.

Todos esses recursos e funções do C# foram estudados e implementados em códigos de teste antes de serem utilizados todos juntos em um único programa, para que o processo de familiarização ocorresse mais tranquilamente e de modo mais eficiente.

3.2 Definição do *software*

Primeiramente, como projeto da ferramenta, foram feitos alguns esboços de telas que poderiam ser implementadas de fato.

Os requisitos a serem seguidos foram:

- Tela fácil de entender, bem intuitiva;

- Possibilidade de escolher mais de um gráfico para serem plotados na mesma área de plotagem;
- Possibilidade de deletar algum gráfico que havia sido adicionado na área de plotagem anteriormente;
- O arquivo a ser lido é em formato *.txt* e contém um conjunto de dados separados por tabulação, que provém da etapa anterior do processador, no qual houve a geração de valores de pontos e organização desses pontos por eixos (pontos no eixo x, y e z) e seus deslocamentos em relação a cada eixo (Figura 2).

Figura 2: Exemplo de arquivo texto a ser lido contendo o conjunto de pontos

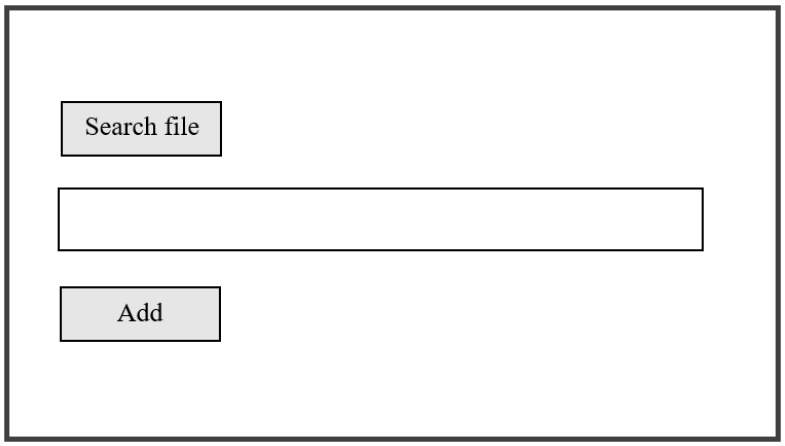
ID	X	Y	Z	U 0 0 Eps 10000	V 0 0 Eps 10000	W 0 0 Eps 10000	
1	101,25	0	0	0	0		
2	101,25	0,0607376727260945		8,46432125645877	-0,000951753003220957	0,000670573163138613	-0,000482787727374412
3	101,25	0,121475345452189		16,9286425129175	-0,00357535286177596	0,00282752267757715	-0,00196539425357179
4	101,25	0,182213018178283		25,3929637693763	-0,00752629439503091	0,00664622459749037	-0,0045090842799918
5	101,25	0,242950690904378		33,8572850258351	-0,0124666559404441	0,0122651179311439	-0,00816335479570786
6	101,25	0,303688363630472		42,3216062822938	-0,0180681363877531	0,0197872758081962	-0,0129667181236873
7	101,25	0,364426036356567		50,7859275387526	-0,0240148271665964	0,0292822016009921	-0,0189475201178969
8	101,25	0,425163709082661		59,2502487952114	-0,0300056979344063	0,0407878225662218	-0,0261247858915477
9	101,25	0,485901381808756		67,7145700516702	-0,0357567956619852	0,0543126530031975	-0,0345090852389429
10	101,25	0,54663905453485		76,1788913081289	-0,0410031410236611	0,069838099095279	-0,0441034106888312
11	101,25	0,607376727260945		84,6432125645877	-0,0455003274889298	0,0873208764426016	-0,0549040602217295
12	101,25	0,668114399987039		93,1075338210465	-0,049025812346965	0,106695512913141	-0,0669015182864999
13	101,25	0,728852072713134		101,571855077505	-0,051379910361456	0,12787690779916	-0,0800813272455511
14	101,25	0,789589745439228		110,036176333964	-0,0523864846123304	0,150762921186283	-0,0944249437053556
15	101,25	0,850327418165323		118,500497590423	-0,0518933499899392	0,175236965417612	-0,109910572169452
16	101,25	0,911065090891417		126,964818846882	-0,0497723890661289	0,201170574597668	-0,126513971397581
17	101,25	0,971802763617512		135,42914010334	-0,0459193999280503	0,22842592575699	-0,144209226406579
18	101,25	1,03254043634361		143,893461359799	-0,0402536805613692	0,25685829033499	-0,162969482508335
19	101,25	1,0932781090697	152,357782616258	-0,0327173727464194	0,286318392089643	-0,182767634985697	
20	101,25	1,1540157817958	160,822103872717	-0,0232745744755667	0,316654653380536	-0,203576971875164	

Fonte: Fornecido pelo orientador.

A partir dos requisitos, a primeira proposta consistiu em uma série de telas que apareceriam seguidamente.

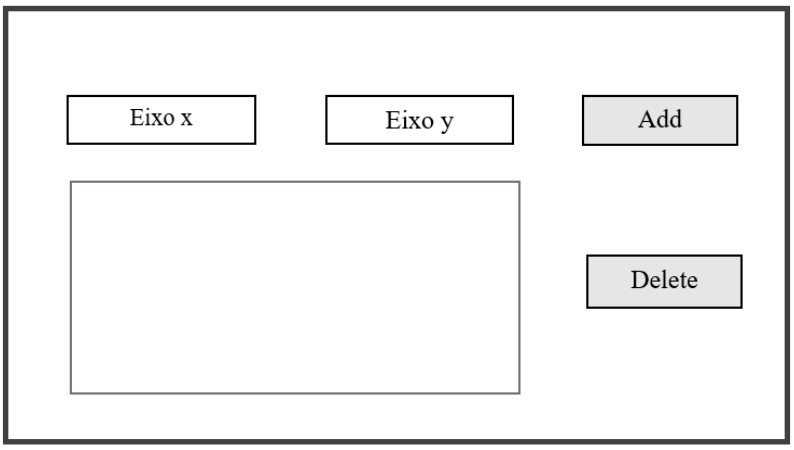
Na primeira tela, haveria um botão “Search”, que abriria a tela de explorador de arquivos do Windows. Teria a finalidade de buscar um arquivo de extensão *.txt* com dados (como o exemplo da Figura 2) a ser lido pelo pós-processador. Encontrado o arquivo, o caminho do arquivo apareceria no *text box* (barra inicialmente vazia no centro da tela) e, sendo esse o caminho correto, o usuário poderia continuar com o programa clicando em “Add”, como mostrado na Figura 3.

Figura 3: Primeira tela, destinada a procura do arquivo a ser lido



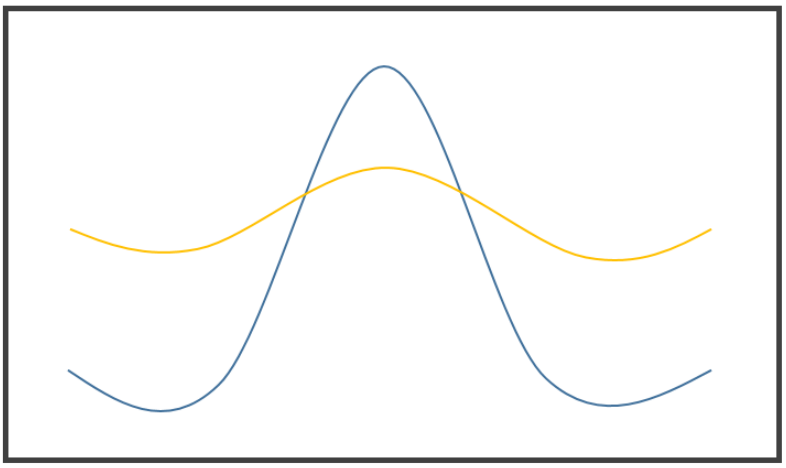
Na próxima tela, haveria dois *combo boxes* destinados à escolha dos dados que comporiam os eixos x e y do gráfico. Escolhidos os dados pelo usuário, eles seriam adicionados na *list view* (área central da Figura 4). Cada item na *list view* consistiria em um gráfico diferente. Seria possível adicionar vários itens e também deletá-los com o botão “Delete”.

Figura 4: Segunda tela, destinada à escolha dos eixos x e y do gráfico



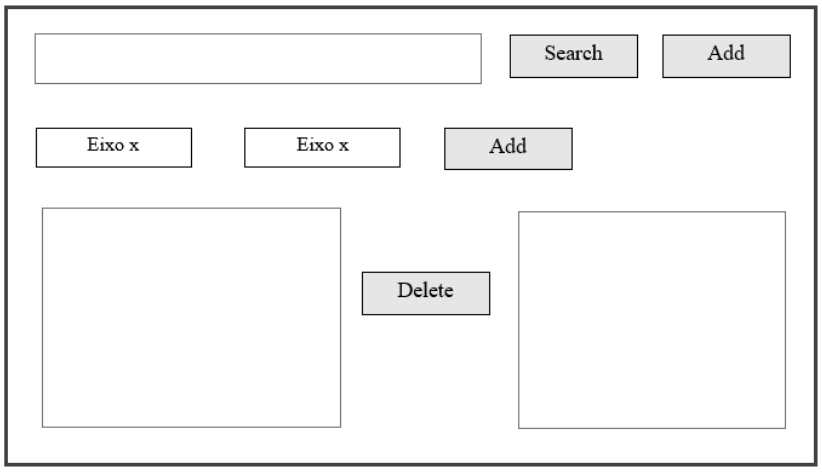
Como resultado do clique nos itens na *list view*, uma nova tela de plotagem seria aberta. Nessa tela, estariam plotados os gráficos escolhidos na tela anterior. Seria possível compará-los e analisá-los conjuntamente (Figura 5).

Figura 5: Área de plotagem resultante contendo vários gráficos



Esse primeiro *design* passou por uma avaliação e precisou sofrer modificações. Uma segunda proposta para as telas do pós-processador consistiu em colocar os mesmos botões e funcionalidades existentes na proposta anterior, porém todos dispostos em uma mesma tela (Figura 6).

Figura 6: Segunda proposta de disposição da tela



Nessa segunda proposta, a área de plotagem estaria presente na área inferior direita da tela. A vantagem dessa disposição é que o usuário teria uma visão completa de todos os botões, sendo mais fácil de visualizar algum erro de escolha e de reparar esse erro.

Foi então escolhida a segunda proposta de apresentação de tela, pois assim ficaria mais simples para o usuário fazer suas escolhas e de voltar alguma ação errada.

3.3 Implementação das funcionalidades

Devido ao fato do WPF ser separado em XAML para a interface gráfica e em C# para a lógica do funcionamento do programa, foi possível focar em uma parte de cada vez, organizando melhor o trabalho.

A Interface gráfica

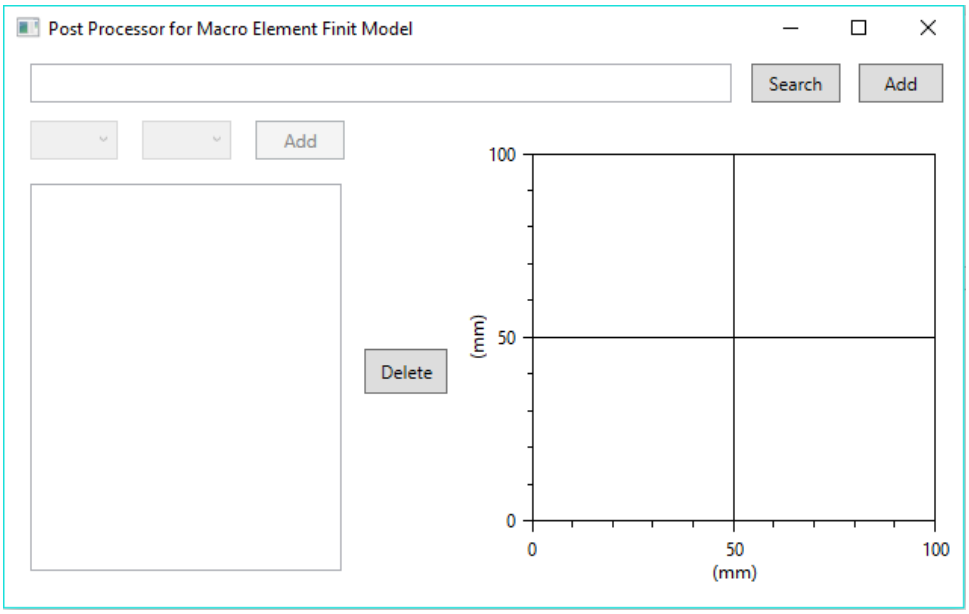
Em primeiro lugar, a tela básica foi desenhada. A partir dela, utilizou-se o painel do tipo *Grid*, elemento mais poderoso do WPF que controla o processamento de conteúdo (tamanho, dimensão, posição e disposição na tela). Uma grande vantagem é que, quando a janela for redimensionada, o conteúdo se manterá na mesma proporção definida.

Com a tela básica feita, foi possível colocar os comandos dispostos como na proposta da Figura 6.

Na parte superior da tela, posicionou-se um *text box* e, do lado direito, dois botões (“Search” e “Add”), que trabalhariam em conjunto. Logo abaixo, foi inserido mais um conjunto de elementos: dois *combo boxes* e um outro botão (“Add”). Por fim, na parte inferior da tela, foram colocadas uma *list view*, um botão (“Delete”) e uma área de plotagem – denominada *plot view* – utilizando-se a biblioteca *OxyPlot*.

O resultado final é mostrado abaixo.

Figura 7: Tela implementada após a inserção de cada elemento



B Lógica do programa

Definida a tela, iniciou-se a implementação do código responsável pelo funcionamento do programa. Mas antes de ser feita a implementação diretamente no código do projeto principal, seria necessário lidar com uma questão: como trabalhar com os arquivos *.txt* de forma organizada e fácil? Foi escolhido então o formato *.json* de arquivos.

JSON é um acrônimo de *JavaScript Object Notation*, uma formatação leve de troca de dados. Suas vantagens incluem a facilidade de ser lido e escrito por pessoas e de ser interpretado e gerado por máquinas. O JSON consegue trabalhar com dados fora de ordem e até mesmo com da-

dos faltantes no arquivo (nesse caso, ele mesmo preenche esses dados com valores *default*), sendo vantajoso em relação aos arquivos de texto, que não oferecem essas facilidades. A utilização do JSON com objetos é muito simples, basta especificá-los entre chaves e atribuir um nome (ou rótulo) que descrevem seus significados, podendo ser compostos por múltiplos pares atributo/valor, por *arrays* e também por outros objetos. Além disso, existe uma biblioteca já pronta para C# que trabalha com transformação de arquivos de/para JSON chamada *Newtonsoft.Json*, fato esse que facilitaria muito a utilização dos dados do arquivo texto como objetos no código.

Dessa forma, o primeiro passo foi serializar os dados do arquivo texto transformando-o em um novo arquivo em JSON. Serialização é o processo de tradução de estruturas de dados em um formato que possa ser armazenado e reconstruído posteriormente no mesmo ou em outro ambiente computacional. Na prática, em um *software* completo, essa etapa seria realizada pelo processador, o qual enviaria para o pós-processador apenas o arquivo final em JSON.

Inicialmente, como é possível observar na Figura 2, havia o arquivo texto com as posições de um ponto em relação aos eixos x, y e z e suas variações em relação a esses eixos. Para transformar em JSON, o arquivo foi separado por IDs: cada ID corresponderia a um objeto diferente, em que cada valor de cada ID seria correspondente a um atributo, assim como na figura abaixo.

Figura 8: Arquivo resultante no formato JSON

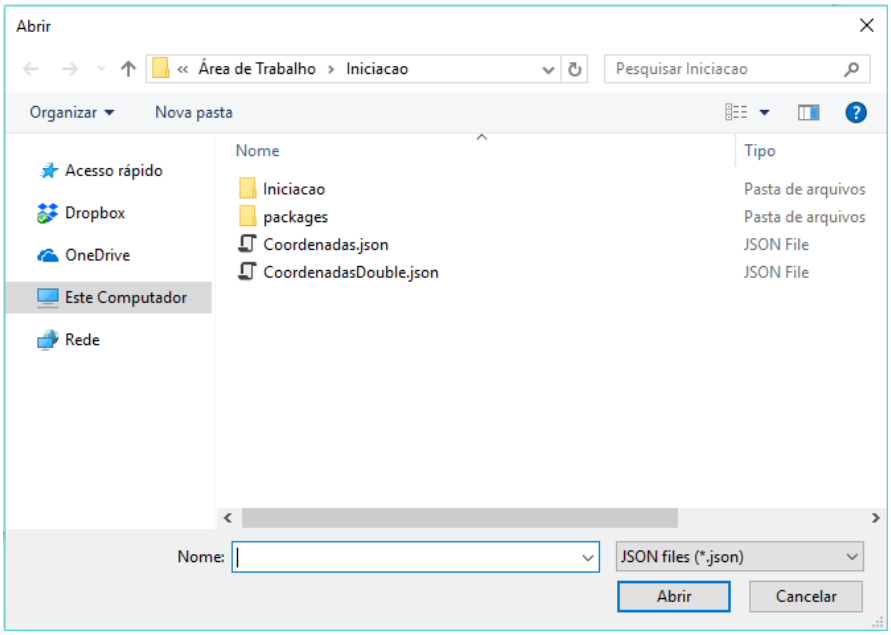

```
[{"ID":1.0,"X":101.25,"Y":0.0,"Z":0.0,"delX":0.0,"delY":0.0,"delZ":0.0},
{"ID":2.0,"X":101.25,"Y":0.0607376727260945,"Z":8.46432125645877,"delX":-0.000951753003220957,"delY":0.0006705731631386l3,"delZ":-0.000482787727374412},
{"ID":3.0,"X":101.25,"Y":0.121475345452189,"Z":16.9286425129175,"delX":-0.00357535286177596,"delY":0.00282752267757715,"delZ":-0.00196539425357179},
{"ID":4.0,"X":101.25,"Y":0.182213018178283,"Z":25.3929637693763,"delX":-0.00752629439503091,"delY":0.00664622459749037,"delZ":-0.0045090842799918},
{"ID":5.0,"X":101.25,"Y":0.242950690904378,"Z":33.8572850258351,"delX":-0.0124666559404441,"delY":0.0122651179311439,"delZ":-0.00816335479570786},
{"ID":6.0,"X":101.25,"Y":0.303688363630472,"Z":42.3216062822938,"delX":-0.0180681363877531,"delY":0.0197872758081962,"delZ":-0.0129667181236873},
{"ID":7.0,"X":101.25,"Y":0.364426036356567,"Z":50.7859275387526,"delX":-0.0240148271665964,"delY":0.0292822016009921,"delZ":-0.0189475201178969},
{"ID":8.0,"X":101.25,"Y":0.425163709082661,"Z":59.2502487952114,"delX":-0.0300056979344063,"delY":0.0407878225662218,"delZ":-0.0261247858915477},
{"ID":9.0,"X":101.25,"Y":0.485901381808756,"Z":67.7145700516702,"delX":-0.0357567956619852,"delY":0.0543126530031975,"delZ":-0.0345090852389429},
{"ID":10.0,"X":101.25,"Y":0.54663905453485,"Z":76.1788913081289,"delX":-0.0410031410236611,"delY":0.069838099095279,"delZ":-0.0441034106888312},
{"ID":11.0,"X":101.25,"Y":0.607376727260945,"Z":84.6432125645877,"delX":-0.0455003274889298,"delY":0.0873208764426016,"delZ":-0.0549040602217295},
{"ID":12.0,"X":101.25,"Y":0.668114399987039,"Z":93.1075338210465,"delX":-0.049025812346965,"delY":0.106695512913141,"delZ":-0.0669015182864999},
{"ID":13.0,"X":101.25,"Y":0.728852072713134,"Z":101.571855077505,"delX":-0.051379910361456,"delY":0.12787690779916,"delZ":-0.0800813272455511},
{"ID":14.0,"X":101.25,"Y":0.789589745439228,"Z":110.036176333964,"delX":-0.0523864846123304,"delY":0.150762921186283,"delZ":-0.0944249437053556},
{"ID":15.0,"X":101.25,"Y":0.850327418165323,"Z":118.500497590423,"delX":-0.0518933499899392,"delY":0.175236965417612,"delZ":-0.109910572169452},
{"ID":16.0,"X":101.25,"Y":0.911065090891417,"Z":126.964818846882,"delX":-0.0497723890661289,"delY":0.201170574597668,"delZ":-0.126513971397581},
{"ID":17.0,"X":101.25,"Y":0.971802763617512,"Z":135.42914010334,"delX":-0.0459193999280503,"delY":0.22842592575699,"delZ":-0.144209226406579},
{"ID":18.0,"X":101.25,"Y":1.03254043634361,"Z":143.893461359799,"delX":-0.0402536805613692,"delY":0.25685829033499,"delZ":-0.162969482508335},
{"ID":19.0,"X":101.25,"Y":1.0932781090697,"Z":152.357782616258,"delX":-0.0327173727464194,"delY":0.286318392089643,"delZ":-0.182767634985697},
{"ID":20.0,"X":101.25,"Y":1.1540157817958,"Z":160.822103872717,"delX":-0.0232745744755667,"delY":0.316654653380536,"delZ":-0.203576971875164},
```


Para realizar a serialização do arquivo texto, o arquivo foi lido com o uso da classe *StreamReader*, separando-o em linhas (por "IDs") e armazenando cada valor em um objeto (dos tipos ID, X, Y, Z, delX, delY e delZ). Ao final de uma linha, os valores foram adicionados em uma lista genérica de nós, e essa lista foi serializada com a utilização da biblioteca *Newtonsoft.Json*. Finalmente, o arquivo resultante mostrado na Figura 8 pôde ser utilizado pelo pós-processador.

Resolvido o problema com os arquivos de entrada do pós-processador, o foco agora seria a implementação dos códigos para os botões. Essa implementação necessitaria um conhecimento prévio sobre eventos. Basicamente, a função dos eventos é a de indicar uma ação que aconteceu com determinado objeto, permitindo que isso seja utilizado como um aviso para outras partes do código serem habilitadas ou não. Os eventos são essenciais quando se trabalha com interface gráfica.

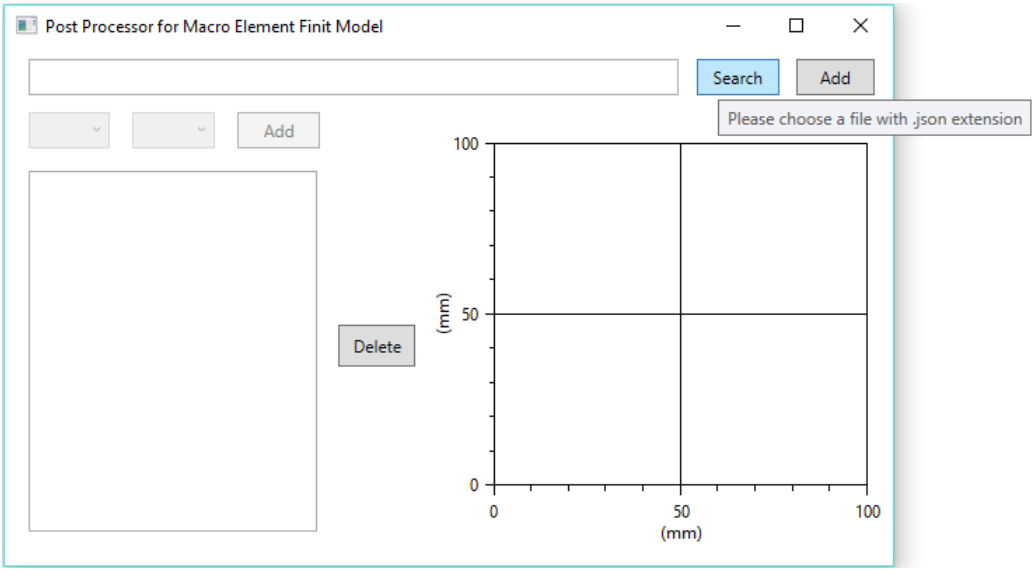
Entendido como os eventos funcionavam, foi possível iniciar a escrita do código de cada elemento da interface. Começou-se então pelo botão "Search", cuja finalidade seria a de abrir o explorador de arquivos do Windows para o usuário procurar um arquivo no formato *.json*. Isso foi implementado com a utilização da classe *Microsoft.Win32.OpenFileDialog*. Estabeleceu-se que o explorador de arquivos trabalharia com um filtro em suas buscas, sendo possível visualizá-lo no canto inferior direito da Figura 9 abaixo. Caso o usuário escolha um arquivo no formato correto, o caminho do arquivo seria mostrado no *text box*. Uma vez satisfeito, o usuário pode clicar em "Add" do lado direito do botão "Search", fazendo o arquivo selecionado ser adicionado no programa. Com o clique no botão, o código já lê e deserializa o arquivo em JSON com a finalidade de ser separado em objetos e estar pronto para ser utilizado.

Figura 9: Tela do explorador de arquivos do Windows, que é aberta ao se clicar o botão “Search”



Uma observação é que foi colocada uma mensagem lembrete para o usuário escolher o tipo de arquivo correto. Essa mensagem aparece quando o usuário passa o mouse por cima do botão “Search”.

Figura 10: Mensagem lembrete para o usuário



Concluída essa etapa, o primeiro *combo box*, anteriormente não disponível para cliques, agora passaria a estar disponível. Nesse combo, o usuário poderia escolher qual será a abscissa do gráfico que será plotado dentre uma das opções: eixos X, Y ou Z do arquivo em JSON. Uma vez selecionada a opção, o segundo combo ficaria disponível. Nesse, seria possível escolher entre delX, delY e delZ. Em seguida, o usuário clicaria em “Add”, fazendo com que as opções escolhidas fossem acrescentadas na *list view*. Concomitantemente, é lançado um evento com o clique do botão que faz com que seja formado um gráfico dos pontos escolhidos com a ajuda da biblioteca *OxyPlot*. A cada nova escolha de objetos a serem plotados, eles apareceriam listados na *list view*.

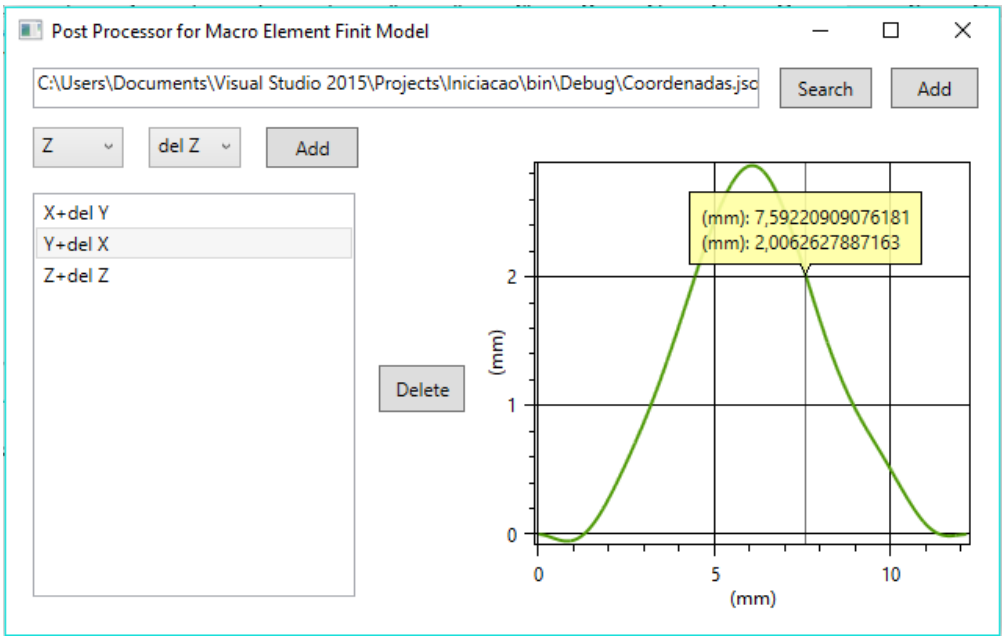
Após isso, com o clique do usuário em um dos itens que foram adicionados na *list view*, o gráfico correspondente apareceria na área de plotagem definida anteriormente. É possível selecionar e adicionar diversos conjuntos de valores de uma só vez (cada um aparece com uma cor diferente) e

verificar as coordenadas de um determinado ponto. Para deletar algum gráfico indesejado, foi implementado o botão "Delete": com ele, é necessário apenas que o usuário selecione o item indesejado da *list view* e clicar o botão em questão. É possível selecionar mais de uma opção e deletar todos de uma vez.

Outro conceito fundamental que foi estudado nessa pesquisa é o conceito de vinculação de dados. Vinculação de dados é o processo que estabelece uma conexão entre a Interface de Usuário e a lógica do programa, ou seja, se um valor ou uma opção for definida pelo usuário, automaticamente os elementos associados aos dados refletirão automaticamente quaisquer alterações que venham a ocorrer no valor desses dados. E para realizar essas vinculações, foi-se utilizado o *data binding*. Utilizando o *binding*, foi possível conectar uma propriedade da interface com uma propriedade do código, permitindo assim a mudança nos valores no código. Além disso, o *binding* pode ser bidirecional: uma mudança na propriedade do código é propagada para a propriedade da interface.

O *data binding* se mostrou extremamente útil para relacionar as opções escolhidas pelo usuário com o código para a plotagem de gráficos. Para tal, foi feito um *Binding PlotModel* no código em XAML para que a área de plotagem na interface recebesse o conjunto de pontos correto, sendo necessário relacionar essas informações com a utilização do comando *this.PlotModel.(propriedade).*

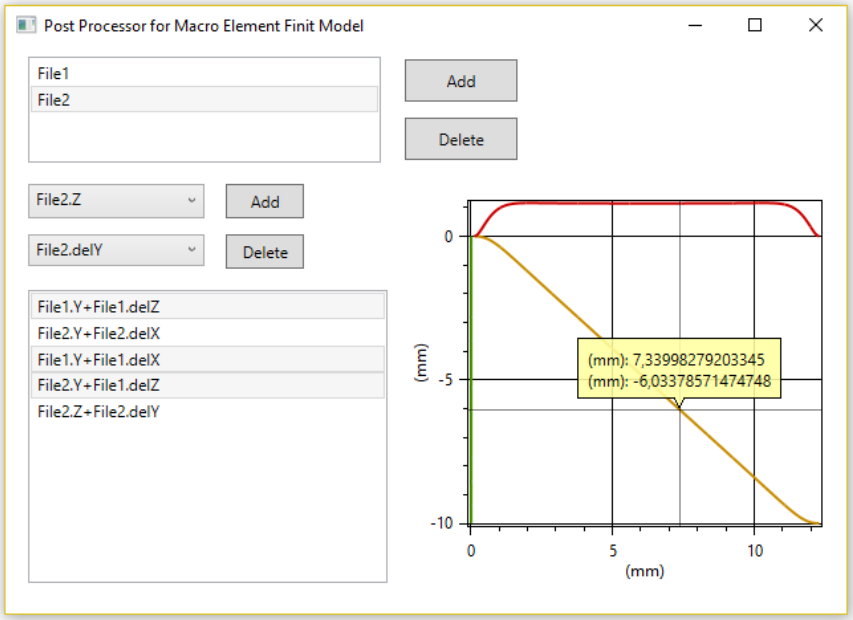
Figura 11: Resultado parcial da interface



Obteve-se então um resultado parcial da interface gráfica mostrada na Figura 11, já funcional, porém com algumas melhorias que poderiam ser feitas. Essa melhorias incluíam a possibilidade de escolha de mais de um arquivo em *.json*, e que fosse possível mesclar o conjunto de dados de um arquivo com um conjunto de dados de outro arquivo em um mesmo gráfico. Além disso, os botões "Search" e "Add" se mostraram equivalentes, podendo ser mudadas para "Add" e "Delete", que teriam funções bem delimitadas. Por último, com a finalidade de tornar a interface mais elegante e organizada, o caminho do arquivo no computador (no *text box*) mostrou-se ser algo desnecessário, portanto, uma solução seria a de listar os arquivos apenas pelo nome deles em uma *list view*.

Essas mudanças foram acatadas, cujo resultado pode ser visualizado na Figura 12.

Figura 12 – Resultado final da interface



3.4 Testes

A etapa de testes consistiu em verificar possíveis erros que poderiam ocorrer durante a execução do programa e realização de tratamentos desses erros para garantir o funcionamento do código.

Uma ferramenta amplamente utilizada para a verificação de erros ao longo da concepção da lógica foi o *debugger* do próprio Visual Studio. Com o *debugger*, foi possível localizar de modo prático e rápido o local no código em que estava com a lógica incorreta, pois ele permite visualizar o passo a passo da lógica sendo executada. Essa ferramenta é muito útil para localizar erros durante o desenvolvimento do código.

Porém, existem erros que podem ocorrer no meio do funcionamento do programa já finalizado, e que podem não estar errados em termos de lógica, mas sim podem lançar exceções quando algum comando é executado de forma incorreta. Algumas exceções que foram tratadas nessa pesquisa foram:

- Ao abrir o explorador de arquivos com o clique no botão “Search”, o usuário deve escolher obrigatoriamente um arquivo *.json*. Caso contrário, o programa mostrará uma mensagem de erro relembrando o usuário dessa condição (Figura 13).

- Devido ao modo em que o código foi escrito, ele só aceita arquivos *.json* cujos valores estejam em *double*. Caso contrário, o programa não funcionará. Com isso, quando o explorador de arquivos for aberto, o usuário deverá escolher um arquivo JSON contendo valores em *double*, pois se essa condição não for satisfeita, ao se clicar em “Add” o programa mostrará uma mensagem de erro (Figura 14).

- Se o usuário tentar adicionar um item na *list view* e esse item já tiver sido adicionado anteriormente, o programa também mostrará uma mensagem de erro (Figura 15).

Figura 13: Mensagem de erro ao se selecionar um arquivo com extensão incorreta

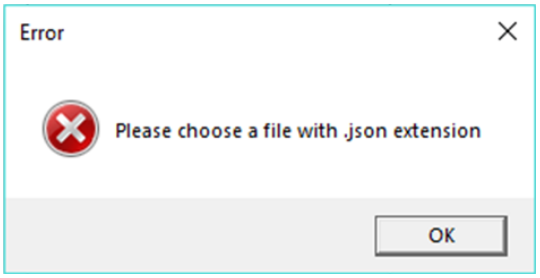


Figura 14: Mensagem de erro ao se selecionar um arquivo em *.json*, porém os valores contidos nesse arquivo estão com um tipo incorreto de valores.

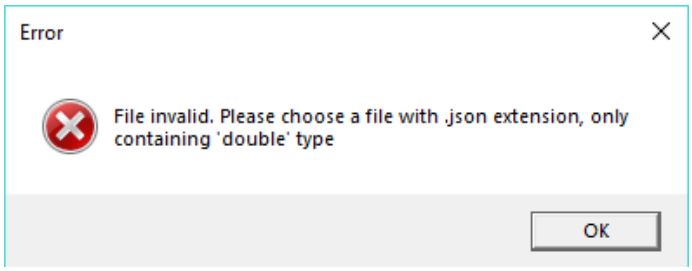
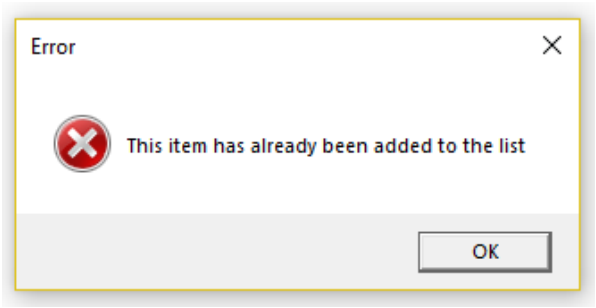
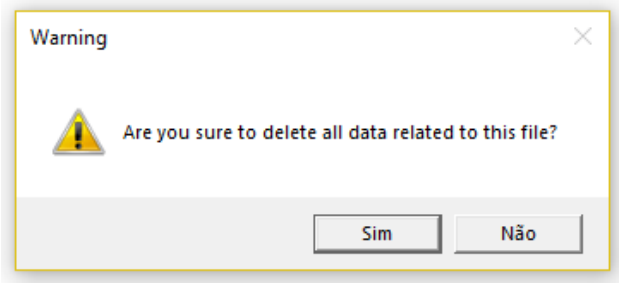


Figura 15: Mensagem de erro na tentativa de adicionar um item que já havia sido adicionado anteriormente na *list view*



Uma mensagem de aviso foi colocada para alertar o usuário no momento em que o primeiro botão de "Delete" for clicado. Esse botão é crítico, pois se o usuário aceitar deletar o arquivo selecionado, todos os demais elementos referentes a esse arquivo serão apagados, inclusive os itens na *list view* e os gráficos (Figura 16).

Figura 16: Mensagem de alerta ao se clicar no primeiro botão de "Delete"



4 Conclusões

A presente pesquisa permitiu um estudo mais detalhado sobre a linguagem C# de programação e a orientação a objetos. Entender sobre várias das características desse tipo de programação, tais como classes, encapsulamento, herança, polimorfismo, sobrecarga, conversão de tipos de dados, tratamento de exceções e os *generics* foram fundamentais para dar prosseguimento a pesquisa. Em relação à interface gráfica em si, aprofundar os conhecimentos sobre o WPF, a utilização do XAML para a criação das janelas da interface gráfica e C# para codificação da lógica da ferramenta do pós-processador foi também de grande importância.

Foram vistas maneiras de tornar o código mais limpo e organizado a partir da utilização dos *generics* e o *binding* que, embora não seja intuitivo em um primeiro momento, acabou se tornando uma ferramenta extremamente útil que possui a capacidade de relacionar as diversas partes do programa com apenas um comando.

Entender mais a fundo sobre interface gráfica também foi muito vantajoso. O conceito de eventos está bem mais consolidado. É certo que os programas mais fáceis e intuitivos são os preferíveis de serem utilizados pelos usuários e em geral são os mais bem-sucedidos no mercado. Por conta disso, é essencial que um *software* seja escrito com a melhor lógica possível e que mostre os dados de modo elucidativo para os usuários.

Além disso, trabalhar com novas bibliotecas como o *OxyPlot* para plotagem de gráficos e o *NewtonSoft* para mexer com arquivos em JSON foi de grande valia. Entender que a utilização de arquivos em JSON facilita o uso de dados pelo programa e que é capaz de armazenar enorme quantidade de informação dentro de um único arquivo de fácil leitura às pessoas também foi engrandecedor.

Fazer a verificação e colocar mensagens de erro se mostrou de extrema importância para qualquer *software*. Isso evita que o programa "quebre" e deixe de funcionar devido a algum erro imprevisto no código. Além do mais, acostumar-se com o *debugger* foi um grande resultado dessa pesquisa, já que o mesmo pode (e deve) ser utilizado sempre em qualquer programação.

Por fim, foi verificada a importância de um pós-processador em um modelo de macroelementos finitos, pois sem um pós-processador, a leitura dos resultados obtidos a partir de um processador seria praticamente impossível, já que são obtidos inúmeros valores, se tornando árduo o trabalho de organizá-los todos eles à mão.

Pode-se concluir que a presente pesquisa foi de extrema ajuda e possibilitou o entendimento desses conceitos, trazendo um conhecimento muito importante na área de computação.

Agradecimentos

Agradeço ao meu orientador, Prof. Dr. Rodrigo Provasi, pela oportunidade e por todo o apoio que recebi durante o desenvolvimento da pesquisa nos anos de 2017 e 2018. Também agradeço à Beatriz Santin e à Bianca Faria, também orientandas do Prof. Rodrigo, por serem minhas companheiras nas reuniões sobre o andamento das pesquisas e estarmos sempre nos ajudando nesse período.

Referências

PETZOLD, Charles. Applications = Code + Markup: A Guide to the Microsoft Windows Presentation Foundation. [S.l.]: Microsoft Press, 2006.

PRESSMAN, Roger S. Software Engineering. 6. ed. New York: McGraw Hill, 2005. 880 p.

PROVASI, Rodrigo; MEIRELLES, Christiano. O. C.; et al. CAD Software for Cable Design: A Consistent Modeling Method to Describe the Cable Structure and Associated Interface. International Congress of Mechanical Engineering COBEM, 2009.

PROVASI, Rodrigo; MARTINS, Clóvis. A. CAD Software for Cable Design: A Three-Dimensional Visualization Tool. International Conference on Ocean, Offshore and Arctic Engineering OMAE2010, Shangai, China. 2010.

PROVASI, Rodrigo. Contribuição ao Projeto de Cabos Umbilicais e Tubos Flexíveis: Ferramentas de CAD e Modelo de Macroelementos. Tese de Doutorado. Escola Politécnica da USP, 2013.

SHARP, John. Microsoft Visual C# 2013 Step by Step. 1ª ed. Pearson Education, 2013, 825p.

SOMMERVILLE, Ian. Software Engineering. 8. ed. São Paulo: Addison Wesley, 2007.

Documentação do *Oxyplot*. Disponível em: http://docs.oxyplot.org/en/latest/. Acesso em 20 de dezembro de 2017.

Title – Post-processor for the finite macroelement model

Abstract – The design of flexible tubes and umbilical cables is a complex process. In order to make all the necessary validations, both global and internal mechanics must be taken into account. Although the model shows great advances, both the input and output of data are still made using text files and commercial tools, which make the process time-consuming and more susceptible to failure. To this end, pre and post-processing tools are proposed to make such a task simpler. This research deals specifically with the post-processing tool. The objective is to construct a tool using the C # language, integrated to the model, allowing the generation of tables and graphs that facilitate the understanding of the phenomenon. It looks for an intuitive tool that is easy to use.

Keywords – Finite macroelements; C#; Post-processor.

Bruna Sayuri de Souza Suzuki iniciou seus estudos em Engenharia Mecatrônica em 2015, pela Escola Politécnica da USP. Teve a oportunidade de apresentar os resultados dessa iniciação científica no 27º SIICUSP (Simpósio Internacional de Iniciação Científica da USP) em 2018. Já foi editora chefe da revista Mecatrone, em 2017.