BEMGUI: INTERFACE GRÁFICA PARA MODELAGEM VIA MÉTODO DOS ELEMENTOS DE CONTORNO

TITO ALBERNAZ GROSSI

MONOGRAFIA DE PROJETO FINAL EM ENGENHARIA CIVIL

DEPARTAMENTO DE ENGENHARIA CIVIL E AMBIENTAL

FACULDADE DE TECNOLOGIA

UNIVERSIDADE DE BRASÍLIA

UNIVERSIDADE DE BRASÍLIA

FACULDADE DE TECNOLOGIA

DEPARTAMENTO DE ENGENHARIA E AMBIENTAL

BEMGUI: INTERFACE GRÁFICA PARA MODELAGEM VIA MÉTODO DOS ELEMENTOS DE CONTORNO

TITO ALBERNAZ GROSSI

ORIENTADOR: PROF. DSc. GILBERTO GOMES

CO-ORIENTADOR: MSc. ÁLVARO MARTINS DELGADO NETO

MONOGRAFIA DE PROJETO FINAL EM ENGENHARIA CIVIL

BRASÍLIA - DF

2020

UNIVERSIDADE DE BRASÍLIA

FACULDADE DE TECNOLOGIA

DEPARTAMENTO DE ENGENHARIA CIVIL E AMBIENTAL

BEMGUI: INTERFACE GRÁFICA PARA MODELAGEM VIA MÉTODO DOS ELEMENTOS DE CONTORNO

TITO ALBERNAZ GROSSI

MONOGRAFIA DE PROJETO FINAL SUBMETIDA AO DEPARTAMENTO DE ENGENHARIA CIVIL E AMBIENTAL DA UNIVERSIDADE DE BRASÍLIA COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE BACHAREL EM ENGENHARIA CIVIL.

APROVADA POR:

_____

GILBERTO GOMES, D.Sc (UnB)

(ORIENTADOR)

_____

LENILDO SANTOS da SILVA, D.Sc (UnB)

(EXAMINADOR INTERNO)

_____

BRUNNO EMÍDIO SOBRINHO, M.Sc (UnB)

(EXAMINADOR EXTERNO)

BRASÍLIA, 18 de DEZEMBRO DE 2020

FICHA CATALOGRÁFICA

GROSSI, TITO ALBERNAZ

BEMGUI: Interface Gráfica para Modelagem via Método dos Elementos de Contorno [Distrito Federal] 2020.

xviii, 81p., 297 mm (ENC/FT/UnB, Bacharel, Engenharia Civil, 2020)

Monografia de Projeto Final - Universidade de Brasília. Faculdade de Tecnologia. Departamento de Engenharia Civil e Ambiental.

1. GUI	2. Modelagem Geométrica
3. Malha de Elementos de Contorno	4. Geometria computacional
I. ENC/FT/UnB	II. Título (Bacharel)

REFERÊNCIA BIBLIOGRÁFICA

GROSSI, T. A. (2020) BEMGUI: Interface Gráfica para Modelagem via Método dos Elementos de Contorno. Monografia de Projeto Final 2 em Engenharia Civil, Departamento de Engenharia Civil e Ambiental, Universidade de Brasília, DF, 81p.

CESSÃO DE DIREITOS

AUTOR: Tito Albernaz Grossi

TÍTULO DA MONOGRAFIA DE PROJETO FINAL: BEMGUI: Interface Gráfica para Modelagem via Método dos Elementos de Contorno

GRAU: Bacharel em Engenharia Civil ANO: 2020

_____

Tito Albernaz Grossi

SHIS QL 18 Conjunto 3, Casa 19, Lago Sul

71650-035 – Brasília/DF – Brasil

e-mail: tito.brossi@gmail.com

AGRADECIMENTOS

Agradeço aos meus pais, Jaqueline e José Artur, minha avó Cléia, minha tia Patrícia e meu irmão Davi, por todo o carinho e amor dedicados a mim e por me apoiarem e estarem comigo em todas as etapas da minha vida, e também por mostrarem, a cada encontro e conversas, como ser uma pessoa melhor.

Aos meus professores orientadores Gilberto Gomes e Alvaro Martins Delgado Neto, por me proporcionarem, por meio deste trabalho, um interesse em engenharia que eu sempre senti que não possuía, e pelos aprendizados ao longo do processo de desenvolvimento deste, o que me ajudou para o crescimento como profissional.

Aos amigos próximos, por todos os momentos de risadas e diversões, e pelas conversas, que me ajudaram a enfrentar os desafios com mais leveza no coração. Em especial: Fernanda Machado, José Eduardo e Nathália Rezende.

Aos colegas da 102ª turma de Engenharia Civil da UnB, pela companhia durante todos os anos de graduação. Mesmo com caminhos distintos, vocês se tornam a cada dia uma inspiração maior para mim.

RESUMO

O desenvolvimento e a adaptação de métodos computacionais para problemas de engenharia, especialmente o Método dos Elementos de Contorno (MEC), tem sido uma área importante de pesquisa nos últimos tempos, tornando cada vez mais útil o uso da computação geométrica aplicada a problemas físicos de modelagem e simulação. O uso correto de estruturas de dados e algoritmos eficientes, no ambiente acadêmico, ainda é escasso, especialmente ao se tratar de projetos manejáveis por um longo período de tempo e que sigam a transparência de projetos de *software* de código aberto. A presente monografia tem como objetivo a implementação de um conjunto de estruturas de dados baseados em grafos e na *doubly conected edge list* (DCEL) que consiga, de forma eficiente, possibilitar a modelagem de zonas bidimensionais de maneira automatizada. Procura também desenvolver uma interface gráfica de usuário (*Graphical User Interface*, GUI) em ambiente computacional que permita o desenho do modelo e a geração da malha de elementos de contorno, bem como a comunicação com programas processadores baseados no MEC, que gere simulações de fatores de intensidades e propagações de trinca. O programa é implementado na linguagem de programação *Python*, com uso da biblioteca externa *PyQt* e substanciado no paradigma de Programação Orientada a Objetos. O código do programa é organizado em diversos modos de compartimentação, a fim de apresentar componentes que façam sentido lógico a partir do uso de *design patterns* e interfaces de classes bem definidas, o que facilita o aprimoramento do programa de maneira sistemática. O uso do conceito de grafos e DCEL permite que a topologia do modelo seja mantida durante o processo de modelagem, tornando-o rápido e flexível. Com a geração de arquivos que descrevem a topologia do modelo e os materiais que compõem as zonas, esses podem ser lidos por *softwares* que realizem simulações, tornando o programa uma ferramenta auxiliar de estudos do MEC, tanto para graduação quanto pós-graduação.

Palavras-chave: GUI, Modelagem geométrica, Malha de Elementos de Contorno, Geometria Computacional, Orientação a objeto.

ABSTRACT

Developing and adapting computational methods for engineering problems, especially the Boundary Element Method (BEM), has been an important area of research in recent times, making the use of computational geometry applied to real physical problems of modeling and simulations more useful than ever. The correct use of data structures and efficient algorithms in the academy is still scarce, especially considering long term management projects and the transparency of open-source software. This monography has the objective of implementing a set of data structures based on graphs and the doubly connected edge list (DCEL) which enables two-dimensional zones modeling in an automated way efficiently. It also aims to develop a Graphical User Interface (GUI) in a computational environment which allows model drawing and boundary element method generation, as well communication with processor programs based on BEM, making it possible to simulate behaviors such as deformations, intensity factors and crack propagation. The program is implemented in the Python language, using the PyQt external library, and based on the Object-Oriented Programming (OOP) paradigm. The code is compartmentalized in different levels in order to create components that make logical sense through the use of design patterns and well-defined class interfaces, which helps the improvement of the program in a systematic way. The use of graphs and DCEL also keeps the model's topology correct during the modeling process, making it fast and flexible. The generation of files that describes the model's topology and the materials that make up zones, these can be read by software that perform simulations, turning the program into an auxiliary tool for BEM studies, both for undergraduate and graduate levels.

Keywords: GUI, Geometric modeling, Boundary element mesh, Computational geometry, Object Orientation

SUMÁRIO

1. INTRODUÇÃO 1

1.1 Considerações Iniciais 1

1.2 Problemática 1

1.3 Justificativa 2

1.4 Objetivos 2

1.4.1 Objetivo geral 2

1.4.2 Objetivos específicos 2

1.5 Organização da Monografia 3

2. REVISÃO BIBLIOGRÁFICA 5

2.1 A geometria computacional 5

2.1.1 A eficiência de um algoritmo 5

2.1.2 Grafos, pilhas e *depth-first search* 6

2.1.3 *Doubly-connected edge list* 11

2.2 A Programação Orientada a Objetos (POO) 17

2.2.1 Abstração 19

2.2.2 Herança e composição 19

2.2.3 Classe 21

2.2.4 Objeto 21

2.2.5 Encapsulamento 21

2.2.6 Polimorfismo 22

2.3 O *Python* 22

2.3.1 Código aberto 22

2.3.2 Alto nível 22

2.3.3 Orientada a objetos 23

2.3.4 Sintaxe simples e clara 23

2.4 O Método dos Elementos de Contorno (MEC) 23

2.5 O BEMCRACKER2D 25

3. METODOLOGIA 27

4. ESTRUTURA DE DADOS PROPOSTA 34

4.1 Modelo Geométrico 34

4.2 Malha 41

4.3 Condições de Contorno 43

5. ORGANIZAÇÃO DO PROGRAMA 45

5.1 Pacote *view* 47

5.2 Pacote *model* 48

5.2.1 Subpacote *geometry* 48

5.2.2 Subpacote *meshgenerator* 49

5.2.3 Subpacote *boundaryconditions* 50

5.2.4 Subpacotes *elastostaticanalysis graphicalresults* 50

5.3 Pacote controller 50

6. A INTERFACE BEMGUI E SUAS FUNCIONALIDADES 54

6.1 *Geometry* (Geometria) 54

6.1.1 BOTÃO “PONTO” 55

6.1.2 BOTÃO “SEGMENTO RETO” 55

6.1.3 BOTÃO “SEGMENTO CURVO” 56

6.1.4 BOTÃO “ZONA” 57

6.2 *Mesh* (Malha) 58

6.3 *Boundary Conditions* (Condições de Contorno) 59

6.4 Elastostatic Analyisis (Análise Elastostática) 61

6.5 *Graphical Results* (Resultados Gráficos) 62

6.6 Parte Informativa do Programa 62

7. APLICANDO O BEMGUI 64

7.1 Exemplo de modelagem com passo-a-passo detalhado 64

7.2 Chapa Cruciforme com Trinca Inclinada 69

7.3 Placa retangular com trinca e furo 72

8. CONSIDERAÇÕES E RECOMENDAÇÕES PARA PESQUISAS FUTURAS74

8.1 Conclusões Gerais 74

8.2 Sugestões para Pesquisas Futuras 75

REFERÊNCIAS BIBLIOGRÁFICAS 77

APÊNDICE A 80

LISTA DE FIGURAS

Figura 2.1 – Simplificação da relação de fronteiras entre países a partir da abstração de um grafo. (Fonte: Dasgupta; Papadimitriou; Varizani, 2008). 7

Figura 2.2 – Representação de pilha e grafo quando novo nó encontrado durante DFS. (Fonte: Autor). 9

Figura 2.3 – Representação de uma divisão planar a partir de um grafo plano. (Fonte: De Berg; *et al*, 2008). 12

Figura 2.4 – Travessia de uma face de forma que esta esteja sempre a esquerda do caminho (a) no sentido horário para o componente externo e (b) no sentido anti-horário para componentes internos (furos). (Fonte: Adaptado de De Berg; *et al*, 2008). 13

Figura 2.5 – Definição das relações "próxima" e "anterior" entre semi-arestas por uma varredura angular do plano. (Fonte: Autor). 15

Figura 2.6 – Exemplo de estrutura completa da *doubly-connected edge list*. (Fonte: De Berg; *et al*, 2008). 15

Figura 2.7 – Conceitos que cercam a POO. (Fonte: Autor). 19

Figura 2.8 – (a, b): Duas tentativas erradas aplicar relações entre três entidades com simples herança e (b) Uma relação correta que cria uma nova abstração a partir de uma composição. (Fonte: Autor). 21

Figura 2.9 – Comparação entre malhas de elementos finitos e malhas de elementos de contorno. (Fonte: Brebbia; Domínguez, 1994). 25

Figura 2.10 – Representação de uma trinca: (a) destacando-se os dois lados opostos e (b): pela simplificação proposta por Portela e Aliabadi (2008). (Fonte: Autor). 26

Figura 3.1 – Processo de modelagem proposta pelo BEMGUI. (Fonte: Autor). 27

Figura 3.2 – Relação entre o BEMGUI e o BEMCRACKER2D, formando um pacote de análise de problemas via MEC. (Fonte: Autor). 28

Figura 3.3 – Casos de uso do programa. (Fonte: Autor). 29

Figura 3.4 – Ambiente do *Qt Creator*. (Fonte: Autor). 31

Figura 3.5 – Caso de uso "desenha modelo geométrico". (Fonte: Autor). 32

Figura 4.1 – Classes da estrutura de dados proposta para modelagem geométrica (a): Point; (b): Edge e (c): Zone. (Fonte: Autor). 34

Figura 4.2 – Tipos de arestas suportadas pela interface: (a) reta; (b) arco; (c) Bézier quadrática e (c) Bézier cúbica. (Fonte: Autor). 35

Figura 4.3 – Curvas de Bézier: (a) quadrática e (b) cúbica. (Fonte: Autor). 36

Figura 4.4 – Inserção de uma aresta ao modelo em que os pontos iniciais e finais não estão conectados a uma zona previamente: (a) uma nova zona pode ser criada e (b) nenhuma nova zona é inserida ao modelo. (Fonte: Autor). 36

Figura 4.5 – Inserção de uma aresta que divide uma zona previamente definida em duas. (Fonte: Autor). 37

Figura 4.6 – Inserção de novas arestas conectadas a componentes do grafo que definem zonas previamente: (a) uma nova zona é acrescentada a vizinhaça da zona previamente definida e (b): duas zonas se tornam compostas. (Fonte: Autor). 38

Figura 4.7 – Relação dos ângulos formados pelo polígono definido pelos pontos de controleuto de uma curva cúbica de Bézier: (a) curva sem intersecção e (b) curva com intersecção. (Fonte: Autor). 39

Figura 4.8 – Topologia entre arestas mantidas mesmo que uma nova aresta seja acrescentada ao modelo. (Fonte: Autor). 40

Figura 4.9 – Elementos de contorno discretizando uma aresta: (a) de forma contínua e (b) de forma descontínua. (Fonte: Autor). 42

Figura 4.10 – relação hieráquica entre modelo geométrico, elementos de controle (malha) e condições de contorno. (Fonte: Autor). 43

Figura 4.11 – Definição de eixos locais. (Fonte: Autor). 44

Figura 5.1 – Diagrama de pacotes do programa. (Fonte: Autor). 45

Figura 5.2 – Estrutura do programa organizada em diretórios (pacotes) e arquivos (módulos). (Fonte: Autor). 46

Figura 5.3 – Diagrama de classes do pacote *view* e um pouco de sua relação com o pacote *controller*. (Fonte: Autor). 47

Figura 5.4 – Diagrama de classes do pacote model. (Fonte: Autor). 48

Figura 5.5 – Modelo possível de se construir no BEMGUI. (Fonte: Autor). 52

Figura 5.6 – Limitação do BEMLAB2D. (Fonte: Autor). 52

Figura 5.7 – Tipos de zonas possíveis de serem trabalhadas pelo BEMCRACKER2D: (a) zona mestre com borda externa bem definida e número indefinido de furos e (b) zona mestre com borda externa indefinida (zona infinita) e um número indefinido de furos. (Fonte: Autor)..... 53

Figura 6.1 – Janela principal da interface gráfica. (Fonte: Autor). 54

Figura 6.2 – Seção *geometry*. (Fonte: Autor). 55

Figura 6.3 – Linhas editáveis referentes às coordenadas do *mouse*. (Fonte: Autor) 55

Figura 6.4 – Exemplos de objeto ponto: (a) em seu estado natural; (b) executando função de *snap* quando o *mouse* está sobre ele. (Fonte: Autor). 56

Figura 6.5 – Janela secundária para escolha de tipo de curva para ser adicionada ao modelo. (Fonte: Autor). 56

Figura 6.6– Diagrama de estados para a criação de aresta do tipo Bézier cúbica. (Fonte: Autor). 57

Figura 6.7 – Definição das características físicas de uma zona. (Fonte: Autor). 58

Figura 6.8– Seção *mesh* (Fonte: Autor) 58

Figura 6.9 – Janela secundária para escolha do tipo e quantidade de elementos: (a) quando se percorre uma aresta que não define uma trinca e (b) quando é identificada uma trinca. (Fonte: Autor). 59

Figura 6.10 – Seção *boundary conditions*. (Fonte: Autor). 60

Figura 6.11 – Seleção de elementos quadráticos para adição de condições de contorno: (a) destaque de elemento quando cursor está sobre este; (b) seleção de múltiplos elementos é possível e (c) as condições de contorno são aplicadas à todos os elementos selecionados..... 60

Figura 6.12 – Janelas secundárias para definição de condições de contorno: (a) deslocamento e (b) esforços. (Fonte: Autor). 61

Figura 6.13 – Seção *elastostatic analysis*. (Fonte: Autor). 61

Figura 6.14 – Caixa de diálogo para definição de últimos dados do problema. (Fonte: Autor). 62

Figura 6.15– Programa lidando com intersecção entre arestas 63

Figura 6.16 – Programa lidando com divisão de zonas 63

Figura 7.1: Representação do desenho a ser modelado. (Fonte: Adaptado de Brebbia; Domínguez, 1994) 64

Figura 7.2 – Disposição dos elementos de ponto para modelagem de hélice. (Fonte: Autor). 65

Figura 7.3 – Definição dos dois primeiros elementos do modelo (elementos quadráticos). (Fonte: Autor). 66

Figura 7.4 – Modelo geométrico de hélice. (Fonte: Autor). 67

Figura 7.5 – Exemplo de definição de discretização de elemento geométrico. (Fonte: Autor). 67

Figura 7.6 – Malha gerada para proposição de modelo de hélice. (Fonte: Autor). 68

Figura 7.7 – Exemplo de modelo final de hélice. (Fonte: Autor). 68

Figura 7.8 – Modelo geométrico da placa cruciforme. (Fonte: Autor). 69

Figura 7.9 – Disposição de pontos para formação de placa cruciforme trincada. (Fonte: Autor). 70

Figura 7.10 – Modelo geométrico de placa cruciforme trincada. (Fonte: Autor). 70

Figura 7.11 – Caixa de diálogo de discretização de trinca (elemento descontínuo). (Fonte: Autor). 71

Figura 7.12 – Malha para placa cruciforme trincada com destaque para trinca. (Fonte: Autor). 71

Figura 7.13 – Modelo final para placa cruciforme com trinca inclinada. (Fonte: Autor). 72

Figura 7.14 – Modelo geométrico da viga trincada com furo dentro do ambiente do BEMGUI. (Fonte: Autor). 72

Figura 7.15 – Malha gerada para viga trincada com furo. (Fonte: Autor.) 73

LISTA DE TABELAS

Tabela 2.1 – Exemplos de uso da notação *big-O* 6

Tabela 2.2 – Relações entre as entidades da figura 2.4. (Fonte: Adaptado de De Berg; *et al*, 2008). 15

Tabela 3.1 – Abstrações de organizações de código segundo os conceitos da POO em *Python* 32

LISTA DE ALGORITMOS

Algoritmo 2.1 – DFS. (Fonte: Adaptado de Dasgupta; Papadimitriou; Varizani, 2008). 9

Algoritmo 2.2 – DFSutil (Fonte: Adaptado de Dasgupta; Papadimitriou; Varizani, 2008). 10

Algoritmo 2.3 – Adaptação do DFS para descobrir se um nó está conectado a um ciclo. (Fonte: Adaptado de Dasgupta; Papadimitriou; Varizani, 2008). 11

Algoritmo 2.4 – Travessia da borda de um componente de zona. (Fonte: Adaptado de De Berg; et al, 2008). 16

Algoritmo 2.5 – Travessia completa da borda de uma zona. (Fonte: Adaptado de De Berg; et al, 2008). 17

Algoritmo 4.1 – são_conectados (Fonte: Adaptado de Dasgupta; Papadimitriou; Varizani, 2008). 37

LISTA DE ABREVIAÇÕES

GUI – *Graphical User Interface*

MEC – Método dos Elementos de Contorno

MEF – Método dos Elementos Finitos

CAD – *Computer Aided Design*

CAM – *Computer Aided Modeling*

GPL – *General Public License*

DFS – *Depth First Search*

DCEL – *Doubly-Edge Connected List*

POO – Programação Orientada a Objetos

UML – *Unified Modeling Language*

DRY – *Don´t Repeat Yourself*

MVC – *Model-View-Controller*

PEP – *Python Enhancement Proposal*

origem(he) – Representação da "origem" de uma semi-aresta he em pseudo-código

próxima(he) – Representação da "próximo" de uma semi-aresta he em pseudo-código

anterior(he) – Representação da "anterior" de uma semi-aresta he em pseudo-código

left(he) – Representação da "esquerda" de uma semi-aresta he em pseudo-código

componente_externo(f) – Representação do componente externo para uma face f em pseudo-código

componentes_internos(f) – Representação de lista de componentes internos para uma face f em pseudo-código

he.origin – Representação do atributo "origem" de uma semi-aresta he em notação de orientação de objetos

he.next – Representação do atributo "próximo" de uma semi-aresta he em notação de orietanção a objetos

he.previous – Representação do atributo “anterior” de uma semi-aresta he em notação de orietanção a objetos

he.right – Representação do atributo “direita” de uma semi-aresta he em notação de orietanção a objetos

1. INTRODUÇÃO

1.1 Considerações Iniciais

Um modelo pode ser entendido como uma visão simplificada da realidade, ou um mundo abstrato formado pela idealização de um sistema físico real (SAMPAIO, 1999). Essa forma de entender o mundo que nos cerca é muito útil para o estudo de sistemas, ideias ou objetos extremamente complexos, já que permite que estes sejam avaliados a partir de parâmetros de escolha do observador.

Dessa forma, é quase natural o avanço de ferramentas de modelagem em quase todas as áreas de conhecimento humano. Mais especificamente, dentro do conceito da mecânica, a modelagem de objetos tridimensionais e bidimensionais se tornou de grande importância para o estudo elástico de propriedades de materiais e conta atualmente com grande participação de sistemas computacionais (GOMES, 2000).

Nesse aspecto, o estudo e o avanço de técnicas de representação gráfica, principalmente aquelas inseridas no ramo da geometria computacional, são tratadas com muita esperança por pesquisadores, já que podem ser integradas a sistemas de mecânica computacional para resultados excelentes de simulações numéricas.

Para uma modelagem geométrica computacional, faz-se necessária também a aplicação de conceitos de representação de objetos da ciência da computação e a definição de processos bem delimitados para representação correta do que se quer modelar. Um dos meios mais comuns de se representar tais objetos e processos, em ambiente computacional, são as estruturas de dados, as quais armazenam os estados de objetos e permitem que ações sejam executadas sobre aquelas. No entanto, recentemente, novas formas de representação se provaram mais fáceis de manejar ao longo do tempo, especialmente em sistemas de grande complexidade.

Assim, este trabalho sugere a implementação de uma estrutura de dados relacionada a modelagem da geometria de objetos físicos em ambiente computacional somada a uma interface gráfica de usuário (*Graphical User Interface*, GUI), ambas implementadas através do paradigma de Programação Orientada a Objetos (POO).

1.2 Problemática

Um dos métodos de simulações mais úteis dentro da área de engenharia é o Método dos Elementos de Contorno (MEC). No entanto, o uso dessa técnica ainda não é tão amplamente

difundido dentro do ambiente comercial quando comparado a um outro grande método de análises numéricas, o Método dos Elementos Finitos (MEF). Ao se tratar de trabalhos focados em modelagem, a diferença no número de trabalhos entre os métodos é ainda maior (GOMES, 2000), o que dificulta a realização de a partir do MEC.

Ademais, muitos programas acabam sendo descontinuados por apresentarem uma maneira pouco clara de seu funcionamento. Isso ocorre, principalmente, em sistemas que possuem muitas relações entre seus componentes, o que torna o processo de alterar algo extremamente trabalhoso.

1.3 Justificativa

Para suprir a carência de ferramentas de modelagem citada, torna-se importante o desenvolvimento de trabalhos que consigam definir de forma robusta objetos no espaço para que possam ser modelados em ambiente computacional. Ademais, inserir essas definições em ambientes gráficos que possam interagir com programas de simulação numérica, formando um pacote de modelagem e simulação, pode ser considerado um passo importante no avanço dos métodos numéricos e da engenharia em si.

O presente trabalho se justifica por fazer parte de um desses pacotes, fazendo o papel de pré-processador em um aplicativo de MEC que seja totalmente desenvolvido em código aberto. Sendo assim, ela promove o desenvolvimento da ciência, já que torna o uso das ferramentas mais universais e acessíveis.

Por fim, o desenvolvimento dessas ferramentas a partir de conceitos que facilitem o seu entendimento como um todo, tornando-as mais receptivas a mudanças e mais capazes de acompanhar o desenvolvimento científico na área também se faz útil no ambiente acadêmico.

1.4 Objetivos

1.4.1 Objetivo geral

Este trabalho tem o objetivo geral de desenvolver uma GUI para a modelagem de zonas físicas bidimensionais a partir da sua representação geométrica e da geração de sua malha de elementos de contorno, seguindo-se os conceitos da POO.

1.4.2 Objetivos específicos

Como objetivos específicos buscou-se o desenvolvimento de um sistema computacional que deve:

- Implementar uma estrutura de dados capaz de automatizar a criação de zonas para modelagem geométrica bidimensional;
- Implementar uma forma de discretização dos elementos de borda das zonas, de maneira a adaptar modelos geométricos para modelos MEC;
- Considerar a adição de características físicas de zonas, como propriedades de materiais e condições de contorno;
- Exportar dados do modelo para a criação de uma interface com programas numéricos de MEC para simulações;
- Criar uma interface gráfica responsiva que permita que usuários possam usar os recursos acima descritos em um sistema computacional gráfico de maneira simples; e
- Tornar os conceitos, componentes e sistemas implementados reutilizáveis em diversos níveis para projetos futuros.

1.5 Organização da Monografia

Esta monografia é organizada em sete capítulos, descritos a seguir:

- O capítulo 2 trata da revisão bibliográfica e busca dar uma explicação abrangente das estruturas computacionais básicas que envolvem o programa e de como o conceito de POO é importante no desenvolvimento de sistemas grandes, além de explicar concisamente alguns outros conceitos usados no projeto;
- No capítulo 3, é descrita a metodologia do trabalho utilizada no processo de desenvolvimento do *software* e enumera as principais ferramentas utilizadas e suas vantagens;
- O capítulo 4 apresenta a estrutura de dados e sua validade para todos os casos propostos de zonas bidimensionais e suas simplificações em malhas, bem como a ideia da aplicação de condições de contorno;
- O capítulo 5 mostra a organização do sistema a partir de uma visão da fase de *design* e a divisão do código implementado em um sistema operacional;
- O capítulo 6 é composto pela apresentação da parte gráfica do pré-processador, incluindo algumas funcionalidades que facilitam o processo de modelagem;
- O capítulo 7 mostra a aplicação do programa a alguns exemplos propostos pela bibliografia utilizada e mostra o passo-a-passo das etapas a serem cumpridas até o final de um modelo; e

- Por fim, o capítulo 8 apresenta uma breve conclusão do projeto, resumindo e sumarizando o objetivo do trabalho, suas contribuições, suas limitações e sugerindo trabalhos futuros que possam surgir a partir dele.

2. REVISÃO BIBLIOGRÁFICA

2.1 A geometria computacional

Um grande passo para a criação de modelos geométricos em ambientes computacionais é a implementação de códigos para definir, computacionalmente, elementos geométricos, objeto de estudo da geometria computacional (BU-QING, 2014).

A importância desse método se dá, de forma extensiva, dentro do desenvolvimento de sistemas de desenho assistido por computador (*Computer Aided Design*, ou CAD), que podem ser integrados a aplicações de simulações numéricas, formado sistemas de manufatura assistida por computador (*Computer Aided Design*, ou CAM). Essa aplicação de geometria computacional permite testes de determinados modelos para verificar se é possível atingir-se determinadas especificações (DE BERG; *et al*, 2008).

A geometria computacional, em conjunto com a mecânica computacional, tem grande importância para a solução de problemas de engenharia, já que possibilita simulações em modelos geométricos a partir da descrição dos contornos desse (GOMES, 2000). Dessa maneira, elas são capazes de formar aplicações completas de modelagem bidimensionais e tridimensionais de zonas no espaço e análise numérica. Ainda há, no entanto, diversos problemas na integração desses conceitos para o uso correto da simulação computacional, o que torna interessante o desenvolvimento de *softwares* na área (GOMES, 2000) e (BU-QING, 2014). Nesse contexto, destaca-se o uso correto e eficiente de estruturas de dados e algoritmos.

2.1.1 A eficiência de um algoritmo

A eficiência de um algoritmo, para este trabalho, é baseada na notação *big-O*, que é definida por Dasgupta, Papadimitriou e Varizani (2008, p. 6):

> "Sejam *f(n)* e *g(n)* funções que definem uma relação entre inteiros positivos e reais positivos. Dizemos que *f = O(g)* (o que significa que '*f não cresce mais rápido que g*') se há uma constante c > 0 tal que $f(n) \leq c \cdot g(n)$."

Essa é uma simplificação para tornar possível a avaliação de eficiência de algoritmos de acordo com o número de dados que devem ser processados, sem se atentar para a velocidade de diferentes processadores ou formas de armazenamento entre máquinas diferentes. Pode-se também escrever essa relação a partir da ideia de limite, sendo as duas maneiras equivalentes:

$$f = O(g) \Leftrightarrow \lim_{n \to \infty} \frac{g(n)}{f(n)} = c, c \geq 0$$

De maneira geral, essa notação busca comparar a função que define o comportamento do algoritmo, que usualmente é difícil de ser analisado, com funções que tem um comportamento muito bem definido. Alguns exemplos comuns da simplificação que essa notação permite pode ser verificado na tabela 2.1.

Tabela 2.1 – Exemplos de uso da notação *big-O*

g	***O(g)***
$5n + 3$	n
$4n^2 + 27n – 3$	n^2
$3n \cdot \log(n^4)$	$n \cdot \log(n)$

Essa notação será usada nas seções abaixo e ao longo da monografia para expressar a eficiência temporal de alguns algoritmos.

2.1.2 Grafos, pilhas e *depth-first search*

Pela natureza interativa da aplicação, um conceito importante (que a princípio não possui relação direta com geometria computacional), é a travessia de um grafo. Um grafo é uma definição matemática que busca representar qualquer forma de relação entre entidades (SKIENA, 2008).

Formalmente, um grafo é escrito na forma G = (V, E), onde V é um conjunto de nós e E é um conjunto de pares de nós, também chamados de aresta. A existência de uma aresta e = (v_1, v_2) em E indica que existe uma relação entre os nós v_1 e v_2. Um exemplo de um grafo pode ser visto na figura 2.1-b, onde cada nó representa um país da américa do sul (figura 2.1-a) e uma aresta indica a existência de uma fronteira entre os países.

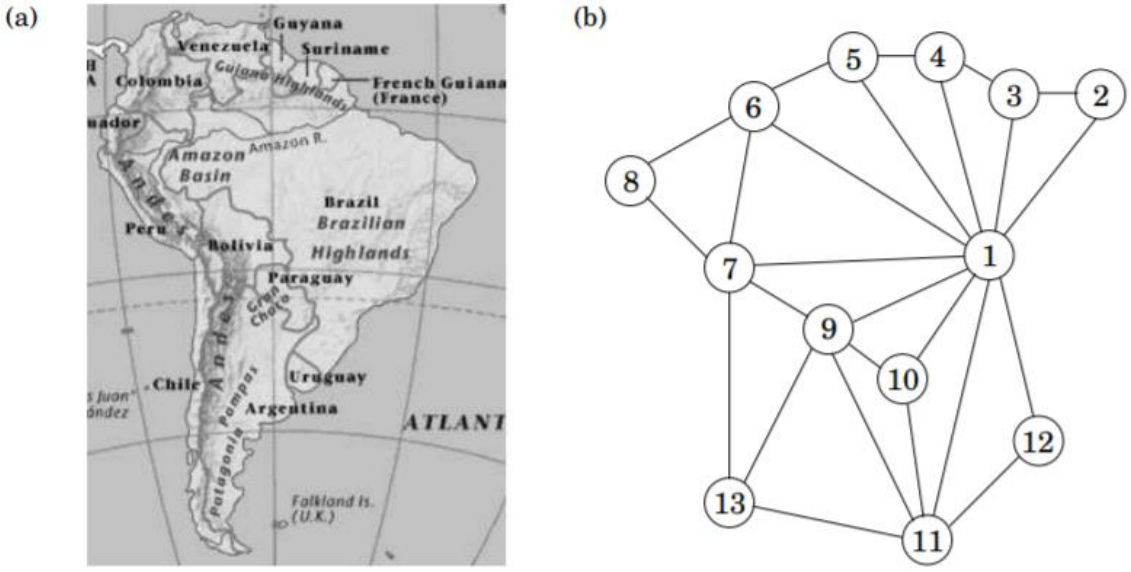

Figura 2.1 – Simplificação da relação de fronteiras entre países a partir da abstração de um grafo. (Fonte: Dasgupta; Papadimitriou; Varizani, 2008).

Segundo Dasgupta, Papadimitriou, Varizani (2008), tal representação permite a abstração de informações complexas como o formato da fronteira entre países e pontos de encontro entre três países vizinhos. Com isso, facilita-se a solução de problemas simples como descobrir o menor número de cores possíveis para colorir tal mapa.

Grafos podem ter várias representações diferentes, que buscam atender características específicas do problema a ser solucionado (SKIENA, 2008). Alguns exemplos de diferentes tipos de classificações de grafos estão listados a seguir:

- Direcionais e não direcionais – Para grafos direcionais, a existência de uma relação entre dois vértices não é, obrigatoriamente, simétrica (em outras palavras, a existência da aresta (v_1, v_2) em E não implica na existência de (v_2, v_1)). O contrário pode ser afirmado para grafos não direcionais. Os primeiros normalmente têm suas arestas representadas com uma seta direcionada ao destino da ligação, diferentemente dos não direcionais, como na figura 2.1-b.
- Cíclicos e não cíclicos – Um grafo é cíclico caso contenha ao menos um ciclo e não cíclico caso contrário. Um ciclo é formado por um caminho que tem origem e destino no mesmo nó, passando por cada aresta no máximo uma vez.
- Esparso e denso – Grafos densos contêm muitos de seus vértices conectados, se opondo aos grafos esparsos. De maneira geral, um grafo é considerado denso se o tamanho de E é igual ou maior que a combinação dos nós, dois a dois (SKIENA, 2008).

- Com ou sem peso – Grafos onde as arestas estão associadas a algum tipo de peso (como a distância entre duas cidades ou o custo de se fazer uma decisão financeira para uma empresa). Grafos sem peso consideram que toda aresta tem um peso unitário.

De maneira geral, há duas maneiras de se representar o conjunto de arestas de um grafo: através de matrizes de adjacência ou através de listas de adjacência. Segundo Dasgupta, Papadimitriou, Varizani (2008), a primeira é mais adequada para grafos densos, e a segunda para grafos esparsos. Essa diferença é causada pela quantidade de armazenamento que cada uma requer computacionalmente: enquanto uma matriz possibilita algumas operações em tempo constante, é necessário sempre um armazenamento proporcional ao quadrado do número de vértices; ao mesmo tempo, uma lista exige tempo linear para obtenção de uma aresta, mas exige também uma complexidade de espaço linear.

Para todos os tipos de grafos citados e para a grande variedade de problemas que sua representação possa vir a resolver (SKIENA, 2008), um dos mais importantes temas é a sua travessia, que pode revelar uma enorme quantidade de características da simplificação (DASGUPTA; PAPADIMITRIOU; VARIZANI, 2008).

Uma das maneiras mais versáteis de se realizar essa travessia é através do algoritmo *depth-first search* (DFS). A pergunta mais simples que se responde com ele é que partes de um grafo (quais nós) são possíveis de alcançar a partir de um nó inicial (DASGUPTA; PAPADIMITRIOU, VARIZANI; 2008).

O algoritmo descrito utiliza o conceito de outra estrutura de dados, a pilha. Por pilha, entende-se um registro ordenado de dados onde podem ser executadas duas ações: *push* (empurrar um novo elemento ao topo da pilha) e *pop* (retirar da pilha o elemento que está em seu topo). Essa descrição é bastante similar a uma pilha de pratos, onde quem a maneja faz sempre o trabalho mais fácil ao alterar somente o topo da pilha. Além disso, ela releva a natureza de operações sobre os elementos nela armazenados, em que o último objeto colocado é sempre o primeiro a ser retirado (comportamento *last in-first-out*, ou LIFO).

A sequência de passos a ser tomada no DFS pode ser definida a partir de uma pilha de nós, em que sempre que um nó começar a ser explorado, ele é posto em uma pilha, e o próximo passo é sacar o último nó empilhado e buscar novos a partir desse. Sempre que um novo nó é encontrado, o último procedimento é interrompido para que se comece a explorar a nova região

do grafo. Um nó só é retirado da pilha quando todos os nós adjacentes a ele já tiverem sido explorados. Uma etapa de um processo DFS está representada na figura 2.2

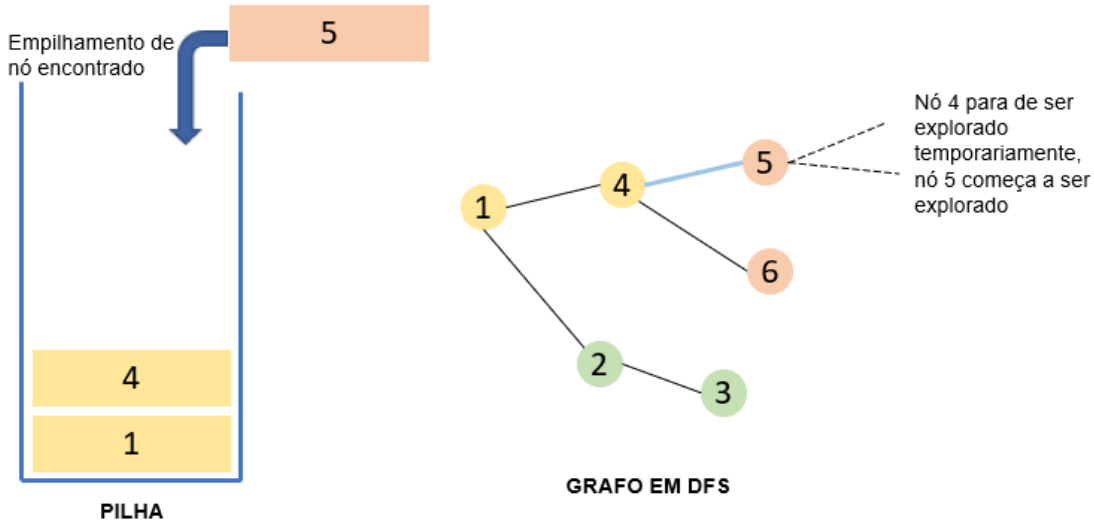

Figura 2.2 – Representação de pilha e grafo quando novo nó encontrado durante DFS. (Fonte: Autor).

Para tal atividade, faz-se necessário que, ao longo do processo, seja mantido um registro de quais vértices já foram visitados (para que uma parte previamente explorada não o seja novamente). Esse registro pode ter várias formas, e a mais convencional é um conjunto de objetos, onde são adicionados os nós encontrados. Com essa explicação, é obtido o algoritmo 2.1, descrito em formato de pseudo-código. Nele, ao invés de ser mantida uma pilha de nós real, ela é representada por recursão, que mantem implicitamente uma pilha de procedimentos e seus estados.

```
DFS (G, u, visitados)

Input: G = (V, E), G é um grafo e u ϵ V, visitados é
um conjunto de nós já visitados

Output: todos os nós possíveis de alcançar a partir
de u

visitados.adicionar(u)
para cada aresta (u, t) ϵ E:
        se t não está em visitados:
                visitados = DFS(G, t, visitados)
retornar visitados
```

Algoritmo 2.1 – DFS. (Fonte: Adaptado de Dasgupta; Papadimitriou; Varizani, 2008).

Executando-se o algoritmo 2.1 com, incialmente, uma lista vazia no lugar de visitados, consegue-se encontrar todos os nós alcançáveis a partir de u. Como cada nó é explorado apenas uma vez, esse algoritmo é de ordem de velocidade linear (DASGUPTA; PAPADIMITRIOU; VARIZANI, 2008).

É válido lembrar que, para uma travessia completa de um grafo, é possível que o procedimento do algoritmo 2.1 tenha que ser chamado múltiplas vezes. Isso é verdadeira em casos em que o grafo não seja completamente conectado. Portanto, para alguns autores, como em Dasgupta, Papadimitriou, Varizani (2008), o algoritmo descrito recebe o nome de *explore*, e o procedimento DFS é descrito como no pseudo-código 2.2, renomeado para *DFSutil*.

```
DFSutil (G)

Input: G = (V, E), G é um grafo
Output:

Visitados = conjunto vazio
para cada u ϵ V:
        se t não está em visitados:
                visitados = DFS(G, t, visitados)
```

Algoritmo 2.2 – DFSutil (Fonte: Adaptado de Dasgupta; Papadimitriou; Varizani, 2008).

No entanto, essa travessia completa de um grafo dentro, no contexto do programa, não se fez útil, já que se busca descobrir somente os nós conectados (formando assim componentes conectados). Portanto, o uso do nome DFS se restringe ao algoritmo 2.1.

Com isso, é possível retirar informações úteis das relações entre vértices, como a linearizabilidde (possibilidade de ordenação de nós em um grafo direcional) e a presença de ciclos (aciclicidade). Esta última, no contexto do programa, se faz extremamente útil, já que é necessário identificar quando uma zona com dimensões é “fechada” de maneira automática.

Um ciclo ocorre quando, a partir do DFS, se chega a um nó já explorado a partir de uma aresta que não tenha sido previamente atravessada, esta que é chamada na bibliografia de *back-edge* (DASGUPTA; PAPADIMITRIOU; VARIZANI, 2008). Em outras palavras, um ciclo ocorre quando um nó que já foi encontrado, mas que ainda não foi totalmente explorado (ainda há nós vizinhos há serem explorados) é encontrado novamente. Isso só ocorre quando o nó encontrado está presente na pilha montada.

Para o caso específico de grafos não direcionais, como as listas de adjacência são simétricas, o nó encontrado, quando já visitado também não deve ser o penúltimo da pilha, já que foi ele que levou ao nó atual. Chamando este nó de "pai" (estabelecendo uma relação de parentesco ou origem, como na estrutura de árvores), o algoritmo 2.3 para detecção de ciclos no grafo pode ser usado.

```
é_cíclico (G, u, p, visitados)

Input: G = (V, E), G é um grafo e u, p ϵ V, p é o
nó que revelou u pela primeira vez (pai), visitados é
um conjunto de nós já visitados
Output: verdadeiro se u estiver conectado a um
ciclo, falso se não

visitados.adicionar(u)
para cada aresta (u, t) ϵ E:
        se t está em visitados:
                se t ≠ p:
                        retornar verdadeiro
        se t não está em visitados: DFS(t):
                retornar(é_cíclico(G, t, u, visitados))
retornar falso
```

Algoritmo 2.3 – Adaptação do DFS para descobrir se um nó está conectado a um ciclo. (Fonte: Adaptado de Dasgupta; Papadimitriou; Varizani, 2008).

2.1.3 *Doubly-connected edge list*

De interesse especial para o programa, que busca representar zonas planares em um espaço dimensional, está uma estrutura de dados capaz de responder questões como "quais são os segmentos que definem uma região?", preferencialmente de maneira eficiente. Como busca-se tratar de múltiplas regiões dividindo o espaço, pode-se usar o termo subdivisão planar (DE BERG; *et al*, 2008), representada na figura 2.3.

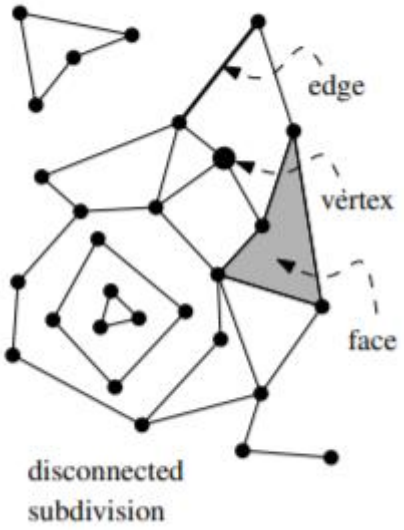

Figura 2.3 – Representação de uma divisão planar a partir de um grafo plano. (Fonte: De Berg; *et al*, 2008).

A figura acima mostra a representação de tal subdivisão a partir da ideia de um grafo. Nele, podem ser vistas anotações de arestas (*edge*), vértices (*vertex*) e faces. Algo importante de se notar é que o grafo observado é de um tipo especial: um grafo planar. Isso significa que o grafo pode ser desenhado de maneira que as arestas só se intersectam em seus pontos de extremidades. Ainda mais, a figura representa um grafo plano, que é uma das possibilidades desses desenhos.

Essas tarefas, assim como outras que se provaram (ou podem vir a se provar) necessárias, se tornam possíveis e simples computacionalmente a partir da estrutura de dados de nome *doubly-connected edge list*, ou DCEL (DE BERG; *et al.*, 2008).

Essa estrutura divide a abstração de uma zona em três simples entidades geométricas (e recorda cada instância destas): pontos, linhas e faces:

- Um ponto é um objeto geométrico que possui uma localização no espaço e é conectado a nenhuma, uma, ou mesmo várias semi-arestas,
- Uma semi-aresta representa uma aresta percorrida em um sentido e está conectada a um ponto (seu ponto inicial). Como esta pode ser percorrida em dois sentidos diferentes, cada semi-aresta possui uma contraparte, que percorre a mesma aresta no sentido contrário e,
- Por último, uma face é um formado por um conjunto de semi-arestas que formam um (ou mais) caminho(s) fechado(s), isto é, cíclico (que começa e termina no mesmo ponto).

Dessa maneira, de acordo com De Berg, *et al* (2008), é condizente dizer que cada semi-aresta deve saber sobre seus pontos de origem e de destino (já que devem haver informações relevantes nestes, como outras semi-arestas que chegam e saem deles) e de sua contra-parte, já que é razoável imaginar que se queira atravessar rapidamente a aresta a que ela se refere no sentido inverso.

No entanto, se cada semi-aresta já possui a informação de sua contra-parte, para acessar seu destino basta acessar a origem desta. Portanto cada semi-aresta deve possuir em sua memória somente sua contra-parte (chamada a partir de agora de gêmea(e) para a semi-aresta e) e sua origem (chamada a partir de agora de origem(e) para a semi-aresta e). Por último, também é interessante guardar a face à qual ela é incidente.

Por convecção (DE BERG; *et al*, 2008), um caminho deve ser percorrido de maneira que a face que ele define esteja a sua esquerda. Portanto, a face ao qual uma aresta é adjacente é chamada esquerda(e). Como representada na figura 2.4-a uma zona tem sua borda atravessada de tal maneira caso a travessia siga um sentido anti-horário. No entanto, também é possível que a zona possua furos. Nesse caso, ao atravessar a borda de um buraco, deve-se percorrer um caminho horário, assim como mostrado na figura 2.4-b

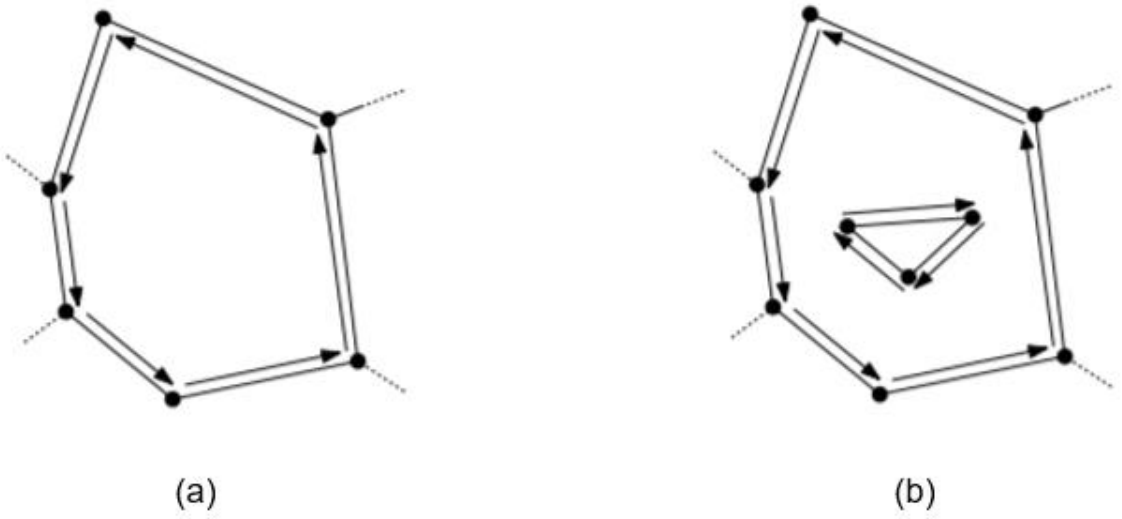

Figura 2.4 – Travessia de uma face de forma que esta esteja sempre a esquerda do caminho (a) no sentido horário para o componente externo e (b) no sentido anti-horário para componentes internos (furos). (Fonte: Adaptado de De Berg; *et al*, 2008).

Pela descrição do último parágrafo e como observado na figura 2.4, a zona deve obviamente ser informada sobre pelo menos uma semi-aresta de seu componente externo, a qual será chamada componente_externo(f) para a face f. Com isso, espera-se ser possível obter

informações sobre outras semi-arestas que se conectam à que a zona mantém em sua memória. Porém, caso a zona possua furos, isso não é suficiente, e ela deve armazenar também uma lista (sem um número definido de elementos) que a informam sobre cada um de seus furos (componentes internos). A essa lista dar-se-á o nome componentes_internos(f) para a face f.

Para completar a representação da DCEL, falta descobrir a relação entre as arestas (qual arestas vem depois de qual). Essa informação é facilmente adquirida ao se fazer uma varredura angular das arestas incidentes ao ponto de destino da semi-aresta. Se essa varredura é feita a partir da aresta a que a semi-aresta que está sendo tratada e retornar os as outras arestas em ordem crescente do ângulo dado pela varredura, a semi-aresta que possui o ponto de destino da semi-aresta que se deseja encontrar a vizinhança como ponto de origem e que se refere a primeira aresta encontrada na varredura é a próxima aresta a se percorrer. Vale notar que essa relação, demonstrada na figura 2.4-a, se inverte caso o caminho percorrido seja um furo.

Caso seja interessante atravessar a zona no sentido inverso ao original, também é possível guardar a informação de qual semi-aresta foi atravessada anteriormente. Isso também é adquirido através da varredura proposta: uma vez que se encontra a próxima semi-aresta p de uma dada semi-aresta t, a anterior de t é p (em outras palavras, anterior(próximo(e)) = e, para uma dada aresta e).

Um último exemplo da estrutura completa pode ser visto na figura 2.5. A definição explícita das relações das entidades pode ser vista na tabela 2.2.

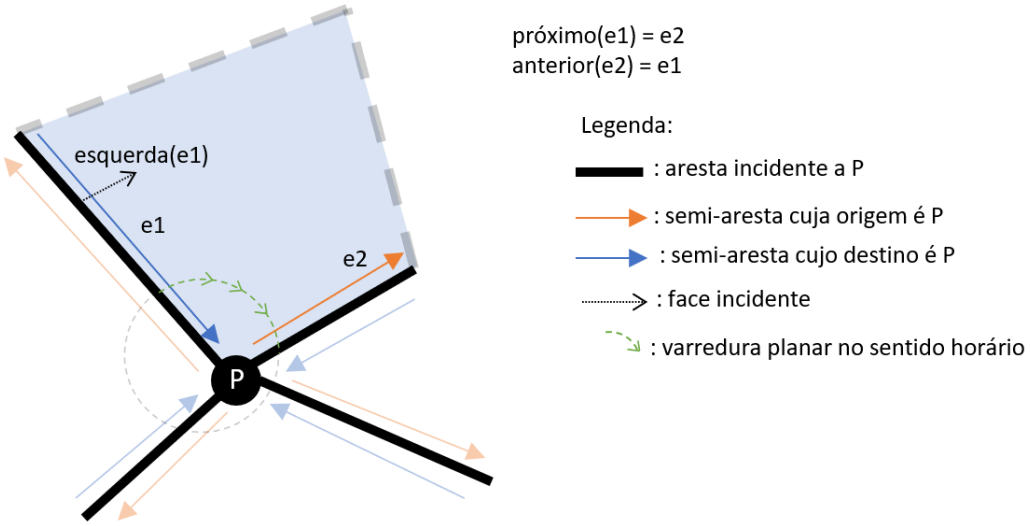

Figura 2.5 – Definição das relações "próxima" e "anterior" entre semi-arestas por uma varredura angular do plano. (Fonte: Autor).

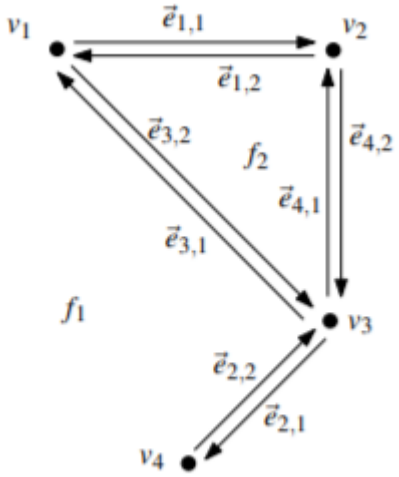

Figura 2.6 – Exemplo de estrutura completa da *doubly-connected edge list*. (Fonte: De Berg; *et al*, 2008).

Tabela 2.2 – Relações entre as entidades da figura 2.4. (Fonte: Adaptado de De Berg; *et al*, 2008).

Semi-aresta	**Origem**	**Gêmea**	**FaceIncidente**	**Próxima**	**Anterior**
$e_{1,1}$	v_1	$e_{1,2}$	f_1	$e_{4,2}$	$e_{3,1}$
$e_{1,2}$	v_2	$e_{1,1}$	f_2	$e_{3,2}$	$e_{4,1}$

$e_{2,1}$	v_3	$e_{2,2}$	f_1	$e_{2,2}$	$e_{4,2}$
$e_{2,2}$	v_4	$e_{2,1}$	f_1	$e_{3,1}$	$e_{2,1}$
$e_{3,1}$	v_3	$e_{3,2}$	f_1	$e_{1,1}$	$e_{2,2}$
$e_{3,2}$	v_1	$e_{3,1}$	f_2	$e_{4,1}$	$e_{1,2}$
$e_{4,1}$	v_3	$e_{4,2}$	f_2	$e_{1,2}$	$e_{3,2}$
$e_{4,2}$	v_4	$e_{4,1}$	f_1	$e_{2,1}$	$e_{1,1}$

Mantendo essas informações, atravessar um componente de uma zona é tão simples como a seguinte seção de procedimentos: ao iniciar uma travessia a partir da semi-aresta desse componente guardada pela zona, percorrer a próxima semi-aresta até que se chegue novamente a mesma aresta. Essa sequência de passos está representada no algoritmo 2.4.

```
atravessar_componente (sm, borda)

Input: sm é uma semi_aresta
Output: lista ordenada de semi-arestas que definem
o componente ao qual sm pertence

se próxima(sm) = sm:
        borda.adicionar(sm)

sm_inicial = sm
enquanto próxima(sm) ≠ sm_inicial:
        borda.adicionar(sm)
        sm = próxima(sm)

retornar borda
```

Algoritmo 2.4 – Travessia da borda de um componente de zona. (Fonte: Adaptado de De Berg; et al, 2008).

Como cada semi-aresta é atravessada somente uma vez, esse algoritmo tem uma eficiência de O(n). Para obter a borda completa de uma zona, é necessário atravessar cada um de seus componentes, o algoritmo 2.5 mostra um método de obtenção da borda de uma zona. No entanto, como componentes diferentes são, por natureza, não conectados entre si, as semi-arestas da zona continuam sendo atravessadas uma única vez, mantendo a ordem linear de eficiência.

```
atravessar_zona (z)

Input: z é uma zona
Output: lista ordenada de semi-arestas que definem
a fronteira da zona z

borda = lista vazia
sme = componente_externo(z)
borda = atravessar_componente(sme, borda)

para smi em componente_internos(z):
        borda = atravessar_componente(smi, borda)

retornar borda
```

Algoritmo 2.5 – Travessia completa da borda de uma zona. (Fonte: Adaptado de De Berg; et al, 2008).

Por fim, como o conceito de grafo e o algoritmo *depth-first search* não têm uma relação direta com a computação gráfica e a geometria computacional, faz-se necessário explicar o porquê de esta sub-seção estar presente na seção referente àqueles temas. Por isso, o parágrafo seguinte faz uma ligação (a princípio não aparente) entre os dois temas.

Na realidade, a estrutura *doubly-connected edge list* (explorada na próxima sub-seção) é uma maneira especial de representar e explorar um grafo, que leva em conta também características geométricas para definir quais informações armazenar e diferentes funcionalidades para extrair informações importantes, mas particulares dessa área de domínio. De maneira geral, um ponto é similar a um vértice e uma semi-aresta é similar a uma aresta em um grafo direcional (um grafo não direcional tem o mesmo significado de um grafo direcional onde todas as arestas do primeiro são substituídas por duas arestas direcionais de sentido contrário no segundo).

2.2 A Programação Orientada a Objetos (POO)

“A Programação Orientada a Objetos é atualmente a maneira mais popular de analisar, projetar e desenvolver sistemas de aplicações, especialmente em sistemas grandes” (DATHAN; RAMNATH, 2015, p. 3). Isso é um resultado da capacidade da POO de possibilitar e promover a criação de sistemas que se sustentam em outros, facilitando o conhecimento modular que agiliza processos de desenvolvimento.

O que diferencia a orientação a objetos de sua contraparte, a programação funcional, é que esta é focada no desenvolvimento de programas de computadores para problemas extremamente específicos, nos quais se pode estabelecer uma sequência de passos para a solução, o que se pode chamar de algoritmos.

De maneira geral, a programação orientada a objetos foi desenvolvida ao longo dos anos como resposta à necessidade de uma possibilidade de reutilização de componentes de *software*, assim como visto em objetos físicos de outras áreas da ciência. Essa atividade, dentro da concepção de programação funcional é extremamente difícil ou mesmo impossível (DATHAN; RAMNATH, 2015).

Outra concepção importante que contribuiu para a evolução dos sistemas é uma linguagem padronizada de representação a partir de modelos, a Linguagem de Modelagem Unificada (*Unified Modeling Language*, UML). Essa linguagem está apresentada nos trabalhos de Rumbaugh, Jacobson, Booch (1999) e facilita a apresentação de ideias para pessoas de vários níveis de envolvimento com o desenvolvimento do *software*.

Por isso, o presente trabalho busca apresentar suas concepções e ideias a partir da UML, sempre que possível, com o auxílio da ferramenta *Lucidchart* (Lucid chart inc., 2008), com a exceção das vezes que se julgar que informações serão transmitidas mais claramente de outras maneiras. Algumas vezes serão utilizados ambos.

Grande parte do uso em larga escala da POO se deve ao conceito DRY (*don't repeat yourself*, ou não se repita), que traduz a ideia de reutilização do paradigma. Algumas características relevantes por trás das concepções da POO, presentes na figura 2.7 e listadas a seguir, são responsáveis por isso.

Figura 2.7 – Conceitos que cercam a POO. (Fonte: Autor).

2.2.1 Abstração

A abstração é o conceito que define a imaginação de um objeto, sistema ou processo real devido às suas principais características. A partir dele podemos descrever computacionalmente o que desejamos (STEFIK, 1985).

Pode-se descrevê-la também, de forma mais prática, como a falta de necessidade da compreensão dos diversos níveis de um sistema complexo: quando se utiliza um componente já desenvolvido, imagina-se que este forneça as funcionalidades nele descrita estejam implementadas e não há necessidade de compreender ou modificar comportamentos ou relações intrínsecas desse.

Dessa maneira, essa ideia torna possível preocupar-se apenas com o que o componente oferece de serviços, sob um olhar externo, que pode ser chamado de **interface**.

2.2.2 Herança e composição

A herança pode ser definida como a possibilidade de descrever objetos que são extremamente semelhantes entre si (STEFIK, 1985). Assim, é possível, dentro da POO, declarar que certos

trechos de código são compartilhados entre diferentes partes do programa, o que evita a necessidade de reescrevê-los.

Uma boa forma de entender o conceito de herança é através de uma árvore evolutiva na história biológica. Pode-se imaginar que a classe dos mamíferos possui certas características em comum, como a presença de pulmões, quatro membros e sangue, além das ações de amamentar seus filhotes através de glândulas mamárias e produzir barulho através de órgãos vocais em suas gargantas.

Quando tenta-se descrever uma baleia, no entanto, é costumeiro pensar em outros aspectos: elas conseguem nadar muito bem, vivem em ambientes marítimos e seus membros dianteiros são barbatanas. No entanto, as baleias não perdem as características de um mamífero, o que as mantem dentro dessa ideia de parentesco entre a classe *Mammalia*.

No contexto tratado, essa herança pode ser vista como a aplicação de métodos e atributos de objetos "pais" a objetos "filhos" sem a necessidade de reescrevê-los. Dessa forma, o conceito de herança é um dos pilares do alto nível de reutilização dentro do paradigma da POO.

Assim, ela é, em grande medida, responsável pela reutilização do código que pode ser feita dentro de programas orientados a objetos (STEFIK, 1985). Com isso, os códigos são mais manejáveis a longo prazo e mantem-se mais curtos, já que cada classe que é definida como "filha" de outra e herda desta seus atributos e métodos.

Mais recentemente, observou-se que uma simples relação de herança impossibilita a definição de vários modelos. Como exemplo disso, é possível estender com o exemplo do reino animal descrito e tentar modelar uma generalização de um cavalo, animal quadrúpede da classe *Mammmalia*.

Caso queira-se imaginar um sistema de relações herdadas entre as entidades "quadrúpede", "*Mammalia*" e cavalo, no entanto rapidamente percebe-se que isso não é possível: um cavalo é quadrúpede e mamífero (e deve, portanto, herdar características daqueles), porém nem todos os animais quadrúpedes são mamíferos e certamente nem todos os animais mamíferos são quadrúpedes.

Dessa forma, foi desenvolvida a concepção de composição, na qual uma entidade pode herdar de mais de uma classe. Essa ideia é implementada em várias linguagens modernas que suportam orientação a objetos de maneiras diferentes e está representada na figura 2.8-c

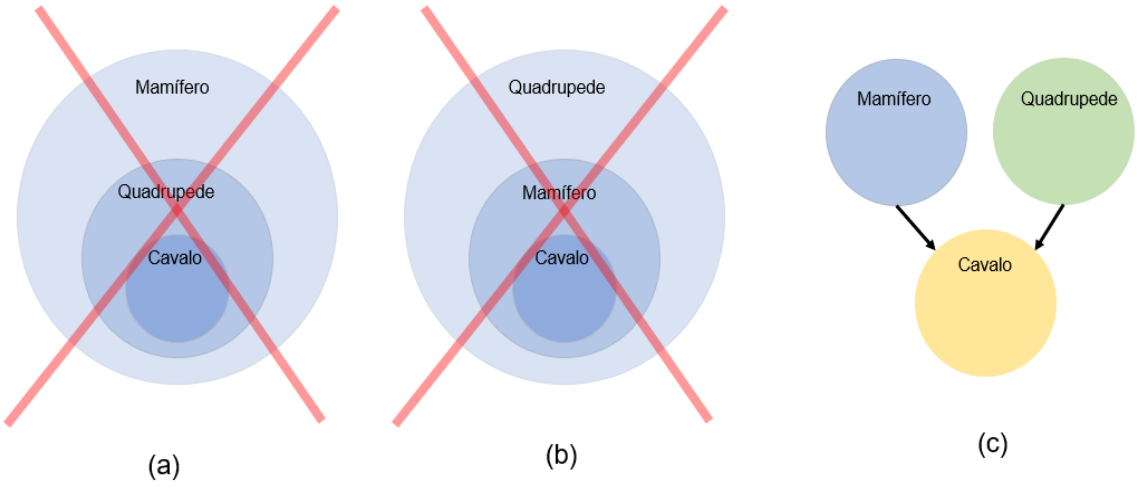

Figura 2.8 – (a, b): Duas tentativas erradas aplicar relações entre três entidades com simples herança e (b) Uma relação correta que cria uma nova abstração a partir de uma composição. (Fonte: Autor).

2.2.3 Classe

Uma classe, dentro da POO, é onde se implementam os conceitos acima listados. Nela, busca-se descrever um objeto a partir do que ele possui (atributos) e do que ele faz (métodos), ou, pelo menos, a partir dos atributos e métodos considerados importante para o que se deseja.

Pode-se descrever uma classe como um espaço em branco, uma forma de se imaginar um objeto de maneira válida para todos (o que esse objeto deve sempre apresentar para o sistema, independente do seu estado).

Nas classes, podemos aplicar o conceito de herança, criando subclasses (classes “filhas”) e alterando levemente suas características e seus comportamentos. Além disso, a concepção de uma classe parte do princípio da abstração (STEFIK, 1985).

2.2.4 Objeto

Um objeto é uma instância de uma classe, ou seja, uma opção de objeto entre outras infinitas opções. Se a classe é uma ideia geral sobre um objeto com espaços a serem preenchidos, um objeto é uma concretização daquela.

Os objetos são a base da POO, e todos os dados e procedimentos aplicados em programas são guardados por esse (STEFIK, 1985). Dessa forma, eles são a parte real de todas as concepções descritas nessa seção, ou seja, o que o computador irá codificar e interpretar.

2.2.5 Encapsulamento

O encapsulamento se refere a proteção do estado interno de um objeto. Para isso, cada atributo e método é classificado entre alguns níveis de acesso, sendo os principais:

- "Privado": apenas o próprio objeto pode acessar esses dados, e o acesso é restringido para outros objetos do sistema e
- "Público": qualquer objeto do sistema pode acessar esses métodos e atributos de forma irrestrita.

Usualmente métodos de uma classe são classificados como públicos e atributos como privados, mas isso pode ser alterado, dependendo do uso que se deseja.

2.2.6 Polimorfismo

O polimorfismo se refere à capacidade de diferentes objetos, mesmo que sejam da mesma classe, apresentarem comportamentos diferentes. Um exemplo seria uma pessoa, que possui uma altura, mas cada pessoa apresenta uma altura diferente. Esse conceito é importante para maior flexibilidade de sistemas.

2.3 O *Python*

O *Python* é uma linguagem de programação de propósito geral, tendo a capacidade de ser aplicada em qualquer meio ou para qualquer objetivo. Apesar de seu maior uso ser como uma linguagem de *scripting*, suporta diversos paradigmas, incluindo a Programação Orientada a Objetos.

2.3.1 Código aberto

A licença do uso do *Python* é baseada na GPL (*General Public License*) com baixa restrição, o que permite que programas escritos nessa linguagem sejam incorporados inclusive em produtos proprietários (BORGES, 2014). Isso permite acesso irrestrito ao código fonte do programa e torna o desenvolvimento científico através da linguagem mais palpável.

2.3.2 Alto nível

Uma linguagem de alto nível é uma linguagem próxima da linguagem humana e distante da linguagem computacional, o que faz com que os programadores consigam escrever códigos complexos naquela sem perder muito tempo com problemas de sintaxe e tempo de atraso de compilação (OLIPHANT, 2007).

Outro fator importante de linguagens desse nível (especialmente da que está sendo tratada) é a grande velocidade na curva de aprendizado, o que ajuda a atrair cada vez mais usuários para o

uso do *Python*. Isso faz a torna uma solução bastante cômoda para aplicações práticas de propósito científico (LUTZ, 2001).

2.3.3 Orientada a objetos

Uma linguagem que suporta orientação a objetos possibilita a reutilização de código em diversos níveis, tornando o desenvolvimento de programas mais rápido e eficiente (LUTZ, 2001). Essa característica também permite a integração da linguagem com um sistema maior, mesmo que este seja implementado em outra linguagem (LUTZ, 2001).

Apesar de não apresentar uma barreira clara sobre o encapsulamento, a linguagem confia no programador em fazer o certo. Além disso, existem convenções sobre o nome de atributos e métodos para que seja possível distinguir entre diferentes níveis de acesso na linguagem.

2.3.4 Sintaxe simples e clara

Uma das maiores vantagens no uso do *Python* em aplicações é o ganho na qualidade do código quanto a legibilidade (LUTZ, 2001). A clareza no que está sendo executado é quase sempre natural (LUTZ, 2001), sem haver uma necessidade de esforços muito grandes para que a aplicação se torne facilmente entendida por outra pessoa.

Isso se torna uma questão importante em um programa em que o caminho crítico é a produtividade do programador ao escrever o código, ao invés da velocidade de execução do programa, como ocorre em grande parte das aplicações atualmente (BORGES, 2014).

2.4 O Método dos Elementos de Contorno (MEC)

O MEC é um método numérico bem estabelecido na comunidade de engenharia, e seus termos fazem referência a qualquer forma de solução por aproximação numérica das chamadas equações integrais de contorno (COSTABEL, 1986). Essas equações são assim chamadas por serem uma ferramenta de análise para obtenção de valores de problemas de contorno para equações diferenciais parciais (COSTABEL, 1986).

O uso deste método se faz importante em casos em que o outro grande método numérico-computacional de engenharia, o MEF, se torna pouco eficaz, com grandes gastos computacionais, pouco preciso ou mesmo incapaz de soluções (BREBBIA, 1989).

As vantagens sobre o MEF são claras quando se deseja tratar de domínios infinitos ou quando se busca uma melhor acurácia para situações de concentrações de carga (BREBBIA, 1989). Há

ainda a vantagem na possibilidade de construção da malha, que, no MEC, pode conter elementos descontínuos, o que não é possível para elementos finitos (BREBBIA, 1989).

A principal característica do MEC, no entanto é a necessidade de modelar somente o contorno dos elementos, o que torna códigos de MEC muito mais fáceis de serem implementados e computacionalmente leves em casos da necessidade de remodelagem de malhas (BREBBIA, 1989).

Essa característica permite que os usuários do método refaçam os processos de construção de malha de maneira mais rápida (já que o método envolve bem menos elementos do que o MEF). Além disso, também traz uma grande vantagem na mudança de *desing* de modelos, já que não há a necessidade de uma mudança total na malha a cada mudança geométrica (BREBBIA, 1989).

Com isso, além da maior adequação à processadores de geometria para geração de malhas de problemas originais, o MEC é mais eficiente, por exemplo, em situações de propagação de trinca, onde a cada passo incremental é necessária a utilização de um código muito pesado para adaptação de malha. Essa relação comparativa está definida em BREBBIA (1989).

Uma outra forma de visualizar as vantagens oferecidas pelo MEC em comparação ao MEF pode ser observada na figura 2.9, que mostra a simplicidade da malha oferecida pelos elementos de contorno em relação àquela oferecida pelos elementos finitos.

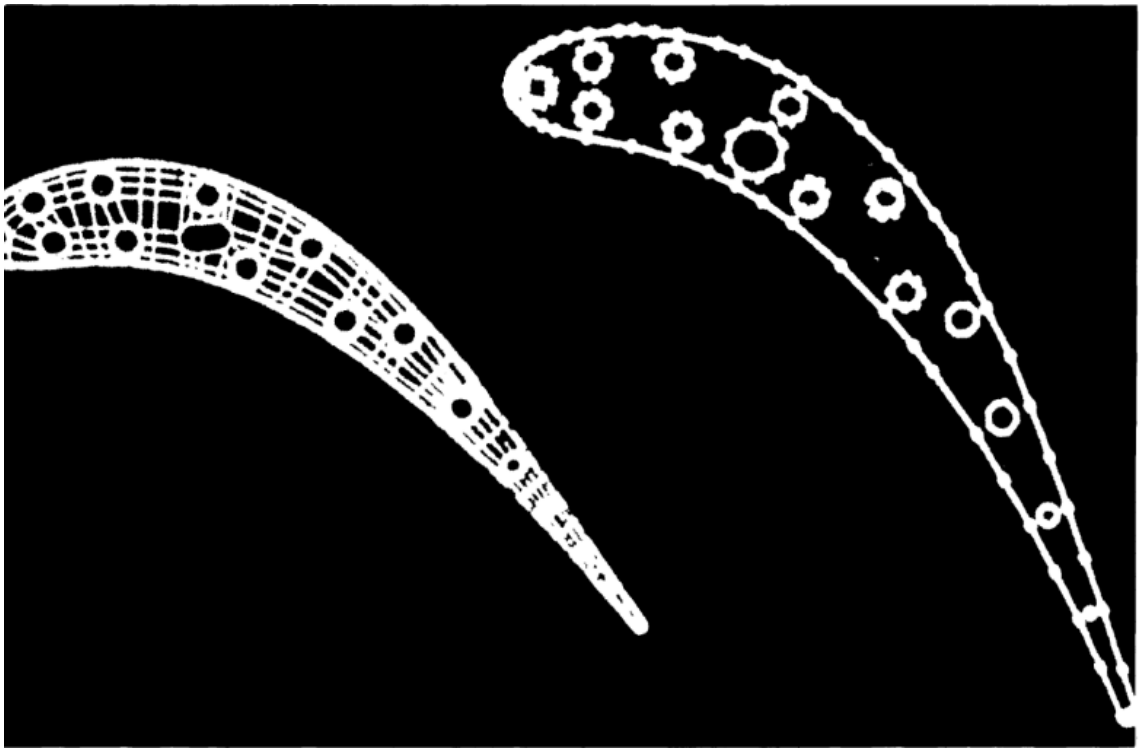

Figura 2.9 – Comparação entre malhas de elementos finitos e malhas de elementos de contorno. (Fonte: Brebbia; Domínguez, 1994).

2.5 O BEMCRACKER2D

O BEMCRACKER2D é um robusto programa acadêmico e científico desenvolvido por Gomes, Delgado Neto, Wrobel (2017) para análise de problemas elastostáticos bidimensionais por meio do MEC, em especial problemas com o aparecimento de trincas (GOMES; MIRANDA, 2018)

Sua construção é baseada na concepção de aplicação do MEC por Portela, Aliabadi (1992), conhecida como MEC Dual (MECD), que determina a formulação das integrais de contorno a partir de dois conjuntos de equações independentes: as equações integrais de contorno de deslocamento e as equações integrais de contorno de forças de superfície, uma para cada face de uma trinca (GOMES; MIRANDA, 2018).

Essa concepção define que, para uma trinca, os elementos de contorno de cada um dos lados desta devem ser definidos pelos mesmos pontos de extremidade (o mesmo caminho, com sentidos diferentes). Com isso, uma trinca é definida por um conjunto ordenado de elementos percorridos em dois sentidos diferentes, também na ordem definida pelo contorno da zona. Essa simplificação é mostrada na figura 2.10.

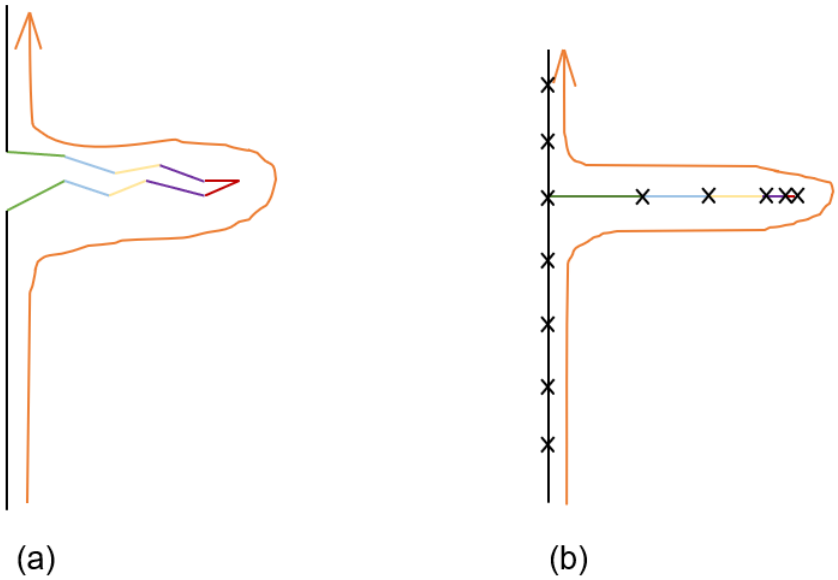

Figura 2.10 – Representação de uma trinca: (a) destacando-se os dois lados opostos e (b): pela simplificação proposta por Portela e Aliabadi (2008). (Fonte: Autor).

A figura 2.10 mostra a simplificação da concepção em termos de modelagem, e destaca a construção dos elementos de contorno compartilhando pontos de extremidade. Para melhor representação, cada elemento é destacado por uma cor nos dois modelos. A seta representa o caminho da zona da qual a trinca faz parte.

O programa BEMCRACKER2D apresenta 3 módulos principais, a saber: MEC padrão, para avaliação de problemas clássicos do método dos elementos de contorno sem a presença de trincas; MECD Sem Propagação, para análise de problemas com presença de trincas usando a formulação de Portela, Aliabadi (1992) sem a consideração de propagação e MECD Com Propagação, para análise de problemas em que se deseja considerar a propagação de trincas e o remanejaento da malha do problema original.

Neste último módulo, onde o programa é focado, há ainda quatro rotinas aplicadas para o estudo físico de regiões bidimensionais. São essas a Análise de Tensões com MEC, a Avaliação de Fatores de Intensidade, a Avaliação de Direção/Correção do Crescimento da Trinca a partir do critério de tensão máxima e a Avaliação de Vida à Fadiga (Lei de Paris) (GOMES; MIRANDA, 2018).

3. METODOLOGIA

A proposta do trabalho está em torno da automatização da criação de elementos e zonas geométricas e a geração de malhas e condições do contorno definidas pelo usuário a partir do uso da linguagem de código aberto, estruturas de dados e arquiteturas de código eficientes, seguras e que proporcionem velocidade e aprendizado ao usuário para a modelagem de problemas do MEC. Essa forma de modelagem adotada está ilustrada na figura 3.1.

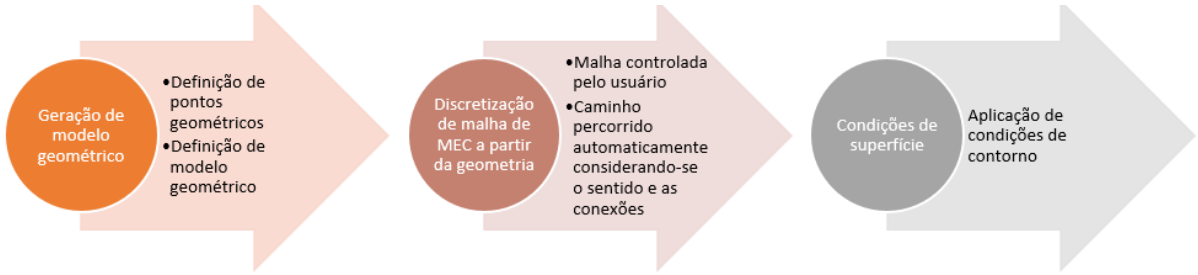

Figura 3.1 – Processo de modelagem proposta pelo BEMGUI. (Fonte: Autor).

O BEMGUI também faz parte de um pacote de soluções de problemas de elemento de contorno, tendo o papel de um pré-processador. Dessa forma, deve-se apresentar uma maneira de comunicação com processadores de maneira que sejam fornecidas a estes seus dados de entrada. Essa relação é principalmente forte com o programa BEMCRACKER2D e, por se tratar de uma interface, ela também pode ser utilizada para gerar visualizações gráficas dos resultados de simulações. Essa interdependência entre os programas pode ser observada na figura 3.2.

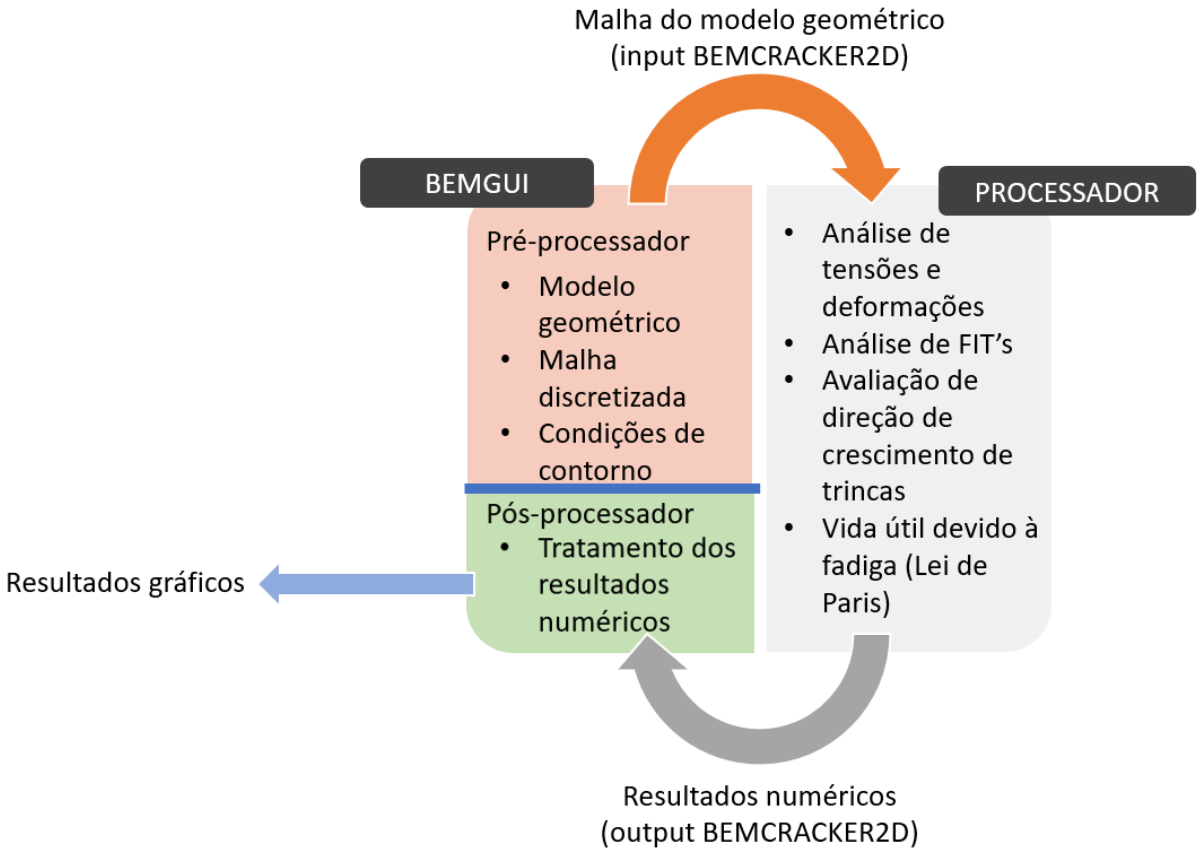

Figura 3.2 – Relação entre o BEMGUI e o BEMCRACKER2D, formando um pacote de análise de problemas via MEC. (Fonte: Autor).

A segunda seção sob BEMGUI na figura acima, no entanto, é ilustrativa para um pacote completo de análise de modelos MEC, já que não faz parte deste trabalho. A relação entre o programa e seu ambiente também pode ser explicada a partir do diagrama de casos de uso da figura 3.3.

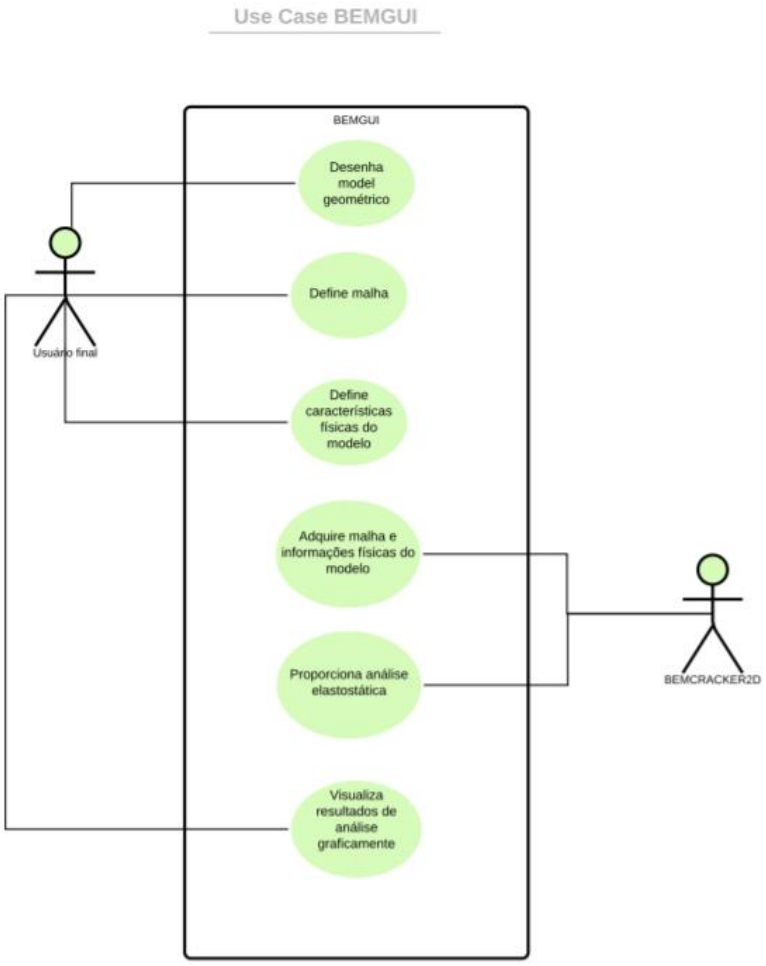

Figura 3.3 – Casos de uso do programa. (Fonte: Autor).

O *design* e a implementação do programa foram realizados a partir da concepção descrita por Sommerville (2011, p. 177),

> "Como todas as atividades criativas, o *design* não é um processo sequencial e bem definido. Você desenvolve um *design* adquirindo ideias, propondo soluções e refinando tais soluções conforme informações se tornam disponíveis. Você inevitavelmente deve retroceder e tentar novamente quando problemas aparecem."

Combinando essa informação com a ideia de que desenvolvimento e *design* de um *software* são atividades entrelaçadas (SOMMERVILLE, 2011), resultou-se em diversas mudanças na distribuição de componentes que o programa apresenta ao longo do trabalho. Para um melhor manejo de mudanças foi utilizada a ferramenta *git* (GIT, 2005), um sistema de versão de controles.

Ferramentas como essa buscam apresentar formas de alterar componentes do *software* de maneira que cada mudança possa ser traçada a uma data e a um autor. Isso permite uma

identificação mais fácil e rápida de erros no programa, e facilitam o retorno a uma versão funcional desse (SOMMERVILLE, 2011).

O uso de versões de controle no desenvolvimento de um sistema é otimizado quando há múltiplas pessoas (ou grupo de pessoas) trabalhando neste sistema de forma paralela. Isso se deve ao fato de evitar-se retrabalho, já que uma pessoa consegue ver tudo que está sendo ou já foi desenvolvido por todas as outras (SOMMERVILLE, 2011). Algo que facilita essa divisão de tarefas são interfaces bem definidas para os diferentes componentes do *software*.

Como o programa se propõem a utilizar dos conceitos da orientação de objetos, ao final buscou-se organizar o código a partir do maior nível de compartimentos que a linguagem utilizada apresenta: pacotes. Essa divisão proporciona um grande nível de abstração e entendimento do funcionamento do código, apesar de tornar o sistema menos eficiente (SKIENA, 2008).

Segundo Sommervile (2011), recentemente o bom desenvolvimento de sistemas computacionais também requer o máximo de reutilização de código possível. Esse reuso pode dar-se através de vários níveis de conhecimento, a saber: abstração, objeto, componente e sistema. O programa faz uso de dois desses níveis, o de componentes e o de abstração.

A reutilização a nível de componentes se traduz no uso das classes apresentadas pela biblioteca *PyQt*. Isso possibilita que comunicações com o a interface gráfica dos sistemas operacionais e a renderização de objetos gráficos não tenha que ser implementada e possa ser abstraída do escopo do código e do trabalho.

A biblioteca usada foi desenvolvida pela empresa *Riverbank Computing* (Riverbank Computing, 2020) e é distribuída pela *General Public Lincense* (GPL), uma licença de código aberto que permite o uso das ferramentas em projetos de terceiros desde que o *software* desenvolvido use o mesmo tipo de licença. Há também licenças comerciais, mas para este trabalho foi usada a verão livre, por se tratar de parte de um projeto acadêmico.

O *PyQt*, por sua vez, foi baseado no *framework Qt*, criado pela empresa *The Qt Company* (2019), e ambos apresentam uma enorme quantidade de *widgets* e funcionalidades para o desenvolvimento de interfaces gráficas e são multiplataformas, rodando em várias distribuições baseadas em *linux*, *unix* e *microsoft*. Por fim, para a criação da parte gráfica das janelas foi utilizada a ferramenta *Qt Creator*, ferramenta gráfica para a criação *designs* gráficos diferentes. O ambiente do *Qt Creator* pode ser visualizado na figura 3.4.

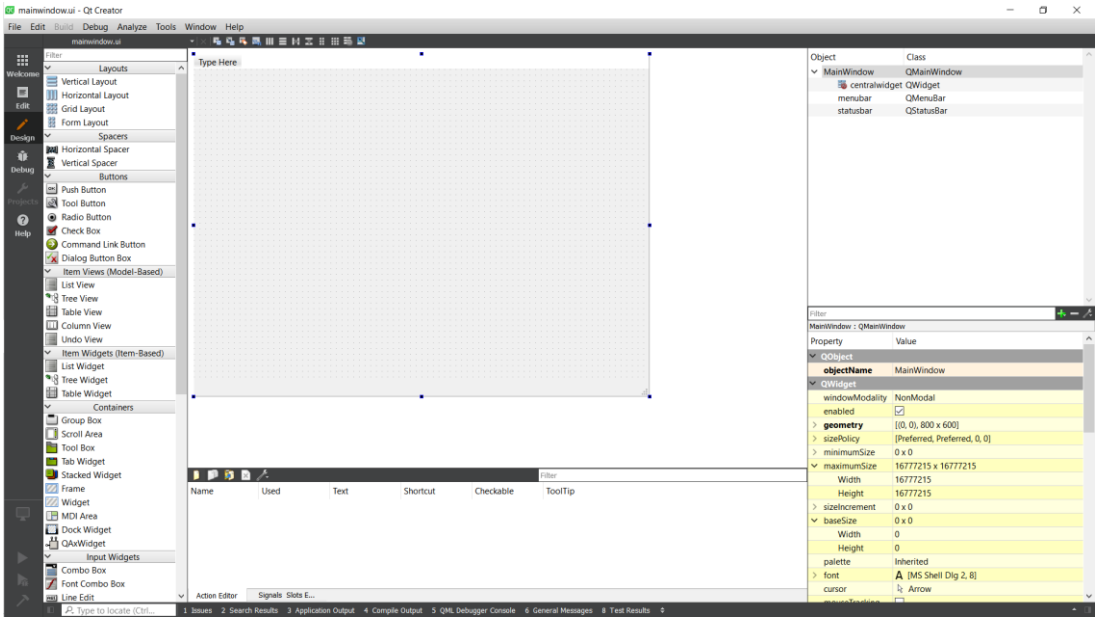

Figura 3.4 – Ambiente do *Qt Creator*. (Fonte: Autor).

A reutilização a nível de abstração, por sua vez refere-se a padrões bem definidos e amplamente estudados e testados (SOMMERVILLE, 2011). Esses padrões são normalmente uma guia para a produção de novos sistemas, facilitando novos *designs*, que podem ser adequados de acordo com a necessidade, e um sistema pode ser composto por vários padrões.

No geral, destaca-se o *model-view-controller pattern* (KROSNER, POPE, 1988), que busca dividir o código entre seções lógicas e é atualmente muito utilizado para sistemas de interface gráfica e aplicações na *web*. As seções são dividas para que cada uma tenha uma responsabilidade sobre o programa, e possa se concentrar nisso:

- Modelo (*model*): Mantem informações sobre os dados puros utilizados no sistema, e proporciona interface para que estes sejam manipulados e observados.
- Vista (*view*): Descreve o que o usuário final visualiza e interage com, isto é, a parte gráfica do sistema.
- Controle (*controller*): O controle age como uma conexão entre o modelo e a vista é onde se encontra a lógica da aplicação.

Com isso, pode-se explicar melhor o caso de uso “desenho do modelo geométrico” da figura 3.3 com o diagrama de sequências, um modelo dinâmico da UML, na figura 3.5.

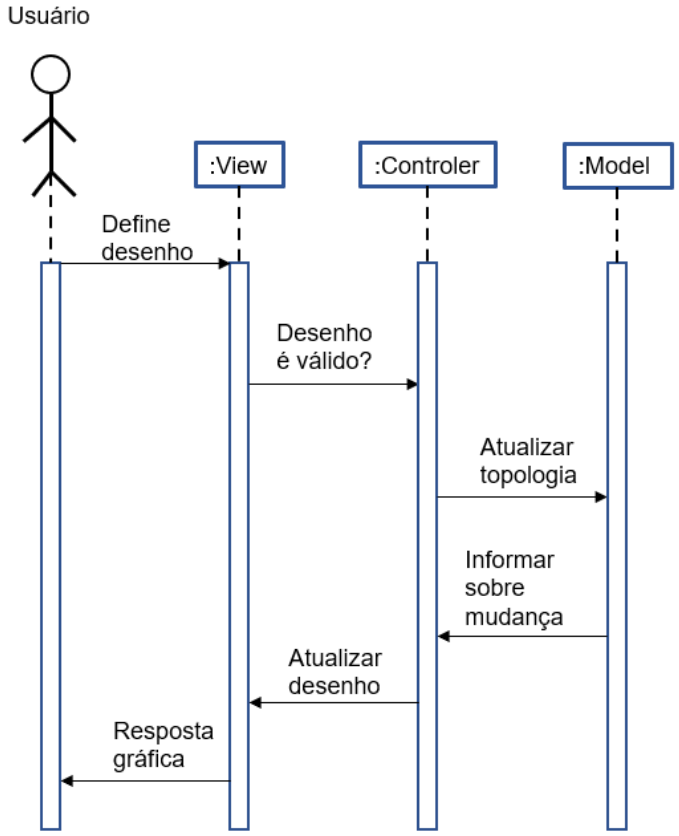

Figura 3.5 – Caso de uso "desenha modelo geométrico". (Fonte: Autor).

Ao final do projeto, buscou-se implementar o código no formato da POO, utilizando-se os diversos níveis abstrações de suportados pela linguagem *Python*, a saber: classes, módulos e pacotes. Cada um desses pode ser entendido, dentro do sistema de organização de arquivos de um sistema operacional da maneira descrita pela tabela 3.1.

Tabela 3.1 – Abstrações de organizações de código segundo os conceitos da POO em *Python*

Conceito	**Descrição**
Classes	Pedaço de código localizado em um módulo e que define objetos
Módulo	Arquivo de extensão *py* localizado em um pacote
Pacote	Diretório/pasta que aloca diversos arquivos de extensão *py*, com um arquivo denominado __init__.py obrigatório

Por fim, para os nomes das diferentes abstrações seguiu-se a convenção de nominação PEP-8 (*Python Enhancement Proposal*) (Python Software Foundation, 1995).

4. ESTRUTURA DE DADOS PROPOSTA

4.1 Modelo Geométrico

Para a modelagem de zonas bidimensionais foram utilizadas três classes, capazes de representar as duas estruturas de dados descritas nas seções 2.2.2 e 2.2.3 deste trabalho (DCEL e grafo). São elas:

- *point*: essa entidade representa tanto os nós de um grafo quanto os vértices de uma DCEL. Para tanto, ele mantém internamente um conjunto de arestas adjacentes a eles. Com isso, todas as arestas podem ser recuperadas, e um ponto pode ordená-las a partir de suas angulações naquele ponto (proporcionando uma varredura angular). Essa classe está representada na figura 4.1-a;
- *edge*: uma aresta tem como atributos seu ponto inicial e seu ponto final. Além disso, cada aresta mantém armazenada em seu escopo uma das semi-arestas associadas a ela, a saber: a semi-aresta que tem origem no ponto inicial da aresta. A maneira que o programa trata semi-arestas utiliza a dinamicidade da linguagem *Python*: elas são atribuídas ao escopo do objeto em tempo de execução através da função *setattr()*. A figura 4.1-b mostra um modelo UML para essa classe. Essa representação de uma aresta permite que pontos possam acessar nós adjacentes a ele (como em um grafo) através de seu conjunto de arestas.
- *zone*: uma zona deve ser apenas atravessada. Portanto, com exceção de características físicas, a única informação necessária a ela é uma das semi-arestas que a definem, como pode ser visto na fiugra 4.1-c.

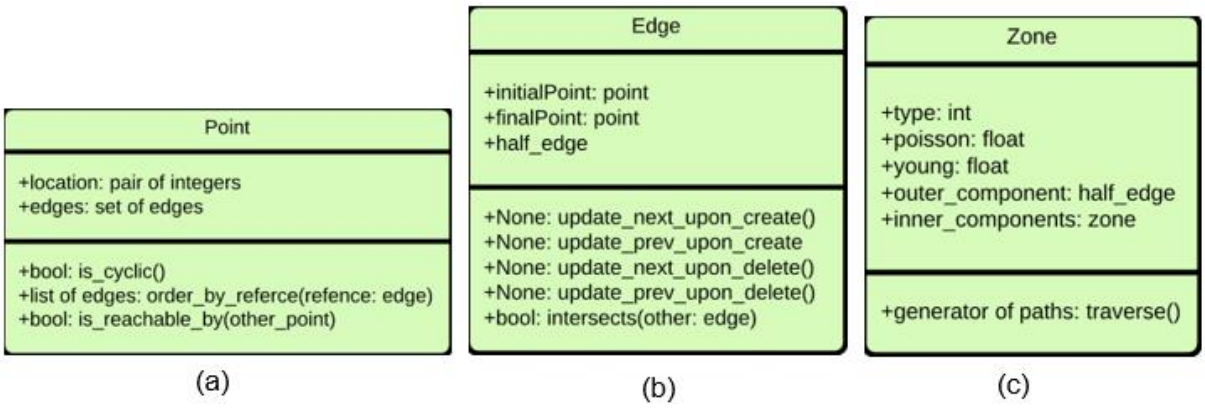

Figura 4.1 – Classes da estrutura de dados proposta para modelagem geométrica (a): Point; (b): Edge e (c): Zone. (Fonte: Autor).

Adicionando-se isso ao fato de essas classes serem herdeiras de objetos gráficos da biblioteca *PyQt*, e elas também podem ser representadas graficamente na interface. Mais especificamente, *point* herda todos os atributos e métodos da classe *QGraphicsEllipseItem*, utilizada para desenhar elipses. Já as classes *edge* e *zone* são herdeiras de *QGraphicsPathItem*, utilizadas no desenho de caminhos arbitrários. Esses caminhos suportam curvas do tipo linha reta, arcos e curvas de Bézier quadráticas e cúbicas.

Com isso, os desenhos suportados pela aplicação podem ser de cada um dos quatro tipos descritos. Isso indica que, caso queira-se fazer uma varredura angular em determinado nó necessita-se utilizar o ângulo inicial ou final das arestas. Um exemplo de cada um dos tipos de arestas apresentados pelo programa pode ser visto na figura 4.2.

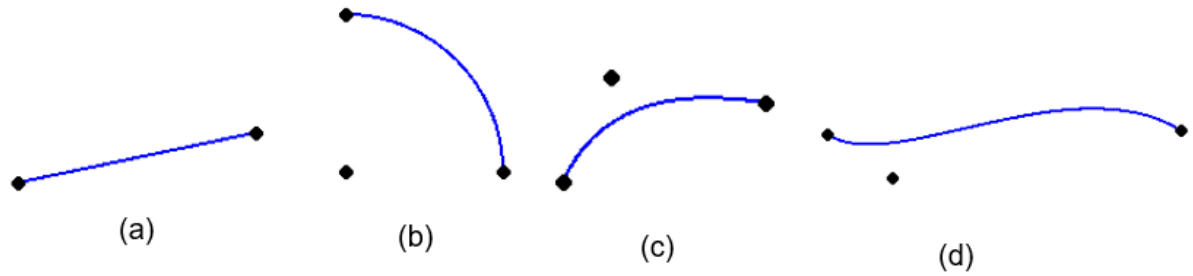

Figura 4.2 – Tipos de arestas suportadas pela interface: (a) reta; (b) arco; (c) Bézier quadrática e (c) Bézier cúbica. (Fonte: Autor).

As curvas de Bézier são curvas paramétricas do tipo *spline* definidas por um ponto final e um ponto inicial e um ou mais pontos de controles. Uma curva de Bézier quadrática possui apenas um ponto de controle e é definida pela equação abaixo.

$$B(t) = (1-t)^2 P_i + 2(1-t)tP_c + t^2 P_f, 0 \leq t \leq 1$$

Onde P_i é o ponto inicial da curva, P_f seu ponto final e P_c seu ponto de controle. Essa equação é aplicada nos dois eixos cartesianos para se obter as coordenadas de cada ponto contido na parametrização da curva. Usando a mesma notação para pontos iniciais e finais, e distinguido os dois pontos de controle da curva de Bézier cúbica por P_{c1} e P_{c2}, a equação dessa é definida pela equação abaixo. Uma representação de cada curva descrita, com seus pontos de controle, pode ser encontrada na figura 4.3.

$$B(t) = (1-t)^3 P_i + 3(1-t)^2 tP_{c1} + 3(1-t)t^2 P_{c2} + t^3 P_f, 0 \leq t \leq 1$$

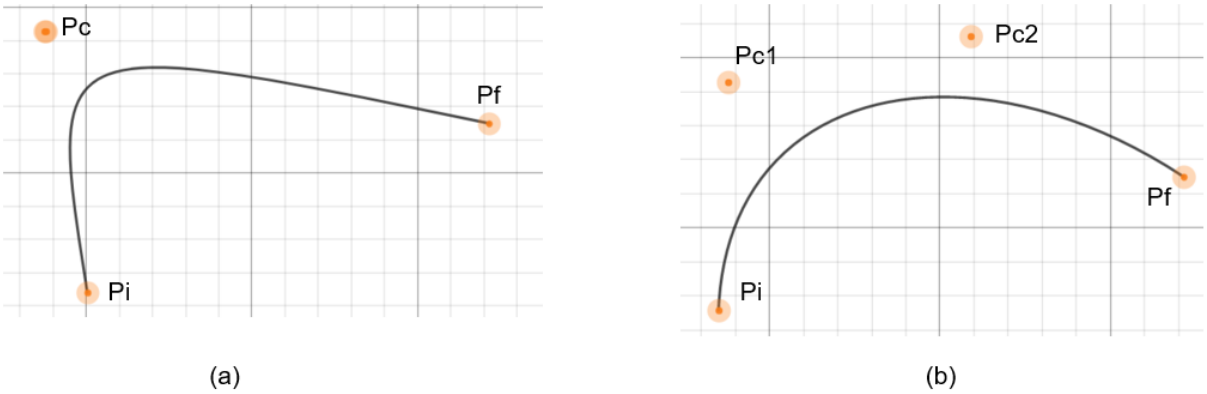

Figura 4.3 – Curvas de Bézier: (a) quadrática e (b) cúbica. (Fonte: Autor).

Os dois objetos que são ativamente desenhadas pelo usuário são os pontos e as arestas, já que o programa propõe automatizar a identificação de zonas. Para tanto, quando uma nova aresta é criada, algumas operações devem ser feitas para a atualização da topologia do modelo.

Nesse contexto, destaca-se quatro possibilidades que podem ocorrer, e essas cobrem exaustivamente todas as opções possíveis (nada além disso deve ocorrer, devido às estruturas de dados aplicadas e as verificações feitas ao se definir uma nova aresta).

A primeira é a criação de uma aresta que conecta dois pontos que não estão conectados a nenhuma zona. Para identificar esse evento, basta verificar se a semi-aresta próxima a semi-aresta armazenada pela aresta criada incide sobre uma face (se seu atributo *right* aponto para algum endereço qualquer que não nulo). Caso isso ocorra, tanto uma nova face pode estar sendo criada, como não. Para identificar se uma zona foi criada, pode-se usar o algoritmo 2.3. Caso seja identificado um ciclo, uma zona foi criada e um novo objeto do tipo *zone* deve ser criado. Se um ciclo não for identificado, basta-se atualizar as relações de tipologia dos nós e semi-arestas. Esses dois casos estão representados na figura 4.4.

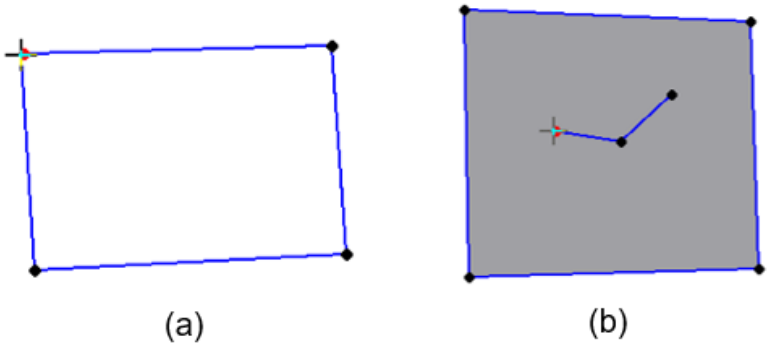

Figura 4.4 – Inserção de uma aresta ao modelo em que os pontos iniciais e finais não estão conectados a uma zona previamente: (a) uma nova zona pode ser criada e (b) nenhuma nova zona é inserida ao modelo. (Fonte: Autor).

Caso alguma das semi-arestas relacionadas à aresta recém criada possua seus atributos *next* e *previous* definindo o contorno de uma mesma zona, essa zona está sendo dividida em duas. Isso provem do fato de que o desenho do grafo não possui interseções fora dos vértices. Esse caso pode ser visto na figura 4.5.

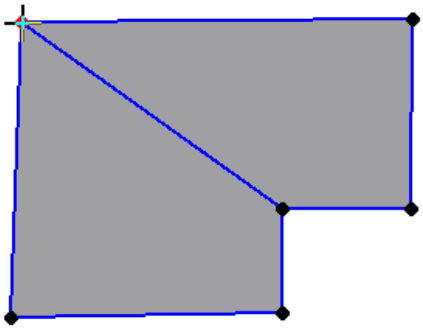

Figura 4.5 – Inserção de uma aresta que divide uma zona previamente definida em duas. (Fonte: Autor).

Os dois últimos casos se resumem na criação de uma nova aresta criada de forma que as semi-arestas anteriores e próximas das semi-arestas associadas àquela não façam parte de uma zona, mas que as gêmeas dessas façam. A partir disso, os pontos final e inicial da aresta podem fazer parte do mesmo componente do grafo desenhado ou não. A diferença entre esses dois casos é possível de identificar através de outra adaptação do DFS (Algoritmo 4.1).

```
são_conectados (G, u, v)

Input: G = (V, E), G é um grafo e u, v ϵ V
Output: verdadeiro se u e v pertencem ao mesmo
componente do grafo, falso caso contrário

visitados.adicionar(u)
para cada aresta (u, t) ϵ E:
        se t = v:
                retornar verdadeiro
        se t não está em visitados:
                retornar são_conectados(G, t, v)
retornar falso
```

Algoritmo 4.1 – são_conectados (Fonte: Adaptado de Dasgupta; Papadimitriou; Varizani, 2008).

Se o algoritmo retornar verdadeiro, os nós fazem parte do mesmo componente, e se o retorno for falso, os pontos fazem partes de componentes diferentes. Esse algoritmo é implementado

no método *is_reachable_from()* da classe *point* (figura 4.1). Dois modelos geométricos que exemplificam esses casos são verificados na figura 4.6.

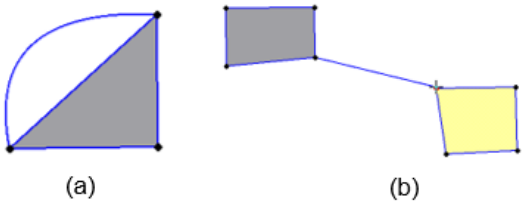

Figura 4.6 – Inserção de novas arestas conectadas a componentes do grafo que definem zonas previamente: (a) uma nova zona é acrescentada a vizinhaça da zona previamente definida e (b): duas zonas se tornam compostas. (Fonte: Autor).

Há ainda a preocupação de definir quais zonas são pertencentes à quais. Para isso, basta verificar se um ponto incidente a zona recém criada está contido na área definida por alguma outra zona. Caso isso seja verdadira, pode-se dizer que a nova zona está contida pela antiga. A partir daí, pode-se usar a lista de componentes internos das zonas para, iterativamente, definir quais zonas pertencem à quais. Se, ao contrário a zona recém definida não for englobada por nenhuma outra zona, deve-se verificar se aquela engloba alguma zona previamente definida.

Essa simples operação de verificar hierarquias de zonas através de um ponto ao invés de zonas extramente complexas de serem avaliadas decorre da propriedade de que o grafo não possui intersecções fora dos vétices. Portanto, um algoritmo muito preciso de verificação de intersecções se faz necessário (este que é baseado no método de intersecção de elementos da biblioteca *PyQt*) e um caso especial deve ser tratado: a auto-intesecção de curvas de Bézier cúbicas.

A solução para esse caso único a curvas definidas por equações de terceiro grau baseia-se no trabalho de Lasser (1989), que define que a soma dos ângulos formados pelas linhas do polígono definido pelos pontos de controle da curva não deve ser maior que 180^o para que não haja intersecções. Essa solução analítica está demonstrada na figura 4.7.

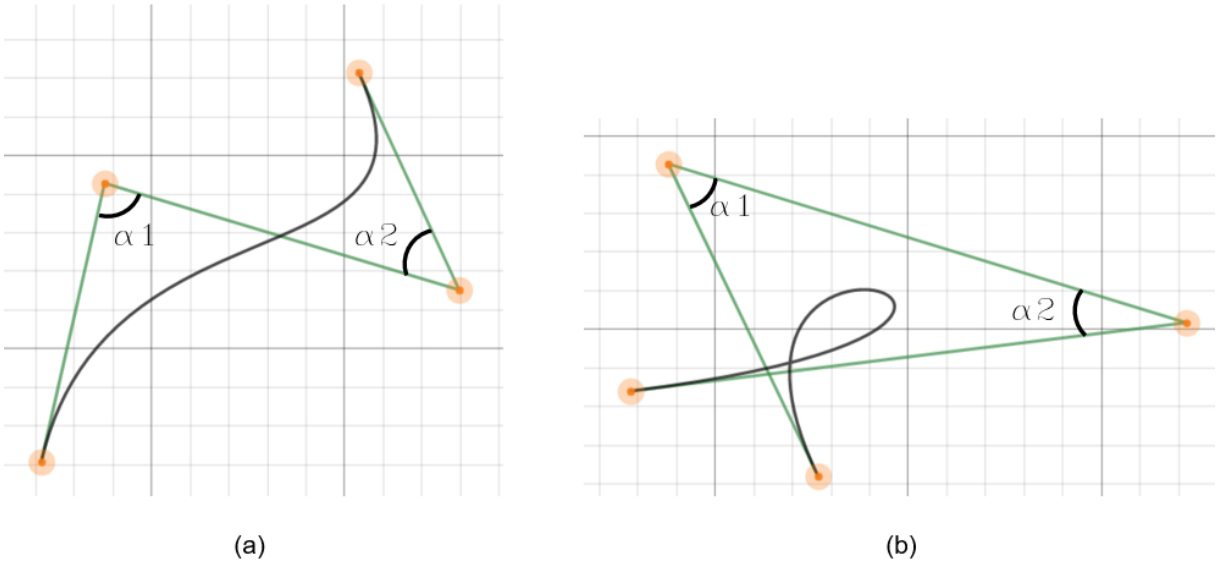

Figura 4.7 – Relação dos ângulos formados pelo polígono definido pelos pontos de controleuto de uma curva cúbica de Bézier: (a) curva sem intersecção e (b) curva com intersecção. (Fonte: Autor).

A última atividade com que os objetos do tipo arestas devem se preocupar é manter a sua topologia após a adição de uma nova aresta ao modelo, o que pode ser alcançdo utilizando-se o algoritmo de varredura para identificar as novas relações de anterior e próximo de cada semi-aresta com uma modificação: ordena-se as arestas em sentido anti-horário. Essa alteração se deve ao fato de que a direção normal a um elemento de contorno é, por definição, o contrário do padrão adotado pela DCEL. Dessa maneira, as orientações para se caminhar sobre uma borda deve ser no sentido horário para bordas externas e anti-horária para bordas internas. Isso também indica uma inversão na direção para a qual se olhar ao atraverssar uma zona, passando de esquerda para a direita.

A sequência de passos abaixo descreve o que deve ser feito (uma representação pode ser visualizada na figura 4.8):

- Os dois pontos finais devem ter a aresta adicionada aos seus respectivos conjuntos de arestas adjacentes;
- Uma semi-aresta he_1 deve ser atribuída a aresta definida. Essa semi-aresta representa o caminho definido pela aresta que tem sentido partindo de seu ponto inicial e se destina ao seu ponto final;
- Para essa semi-aresta, atribuir a seu parâmetro *twin* uma outra semi-aresta he_2 definida pelo método *toReverse()* de sua própria classe;
- Definir *twin* de he_2 como he_1;
- Definir *origin* de e_1 como o ponto inicial e *origin* de e_2 como o ponto final da aresta;

- Fazer uma varredura angular das arestas conectadas aos dois pontos de extremidade da aresta no sentido anti-horário. como proposto na figura 2.5. Essa varredura é feita pelo método *orderByHalfEdge* da classe *point*;
- A aresta mais distante retornada por essa função quando executada no ponto inicial da aresta indica a semi-aresta anterior a e_1 e atribui-se $e_1.previous$ e esta. A aresta mais próximas retornada por essa varredura dá a próxima aresta a e_2, atribuindo-se assim $e_2.next$ à essa primeira semi-aresta encontrada;
- Executa-se a mesma função no ponto final da aresta para se encontrar e_1.next e e_2.previous e
- Definir a(s) nova(s) zona(s) e atribuí-las ao atributo *right* de cada um de seus contornos enquanto isso é feito.

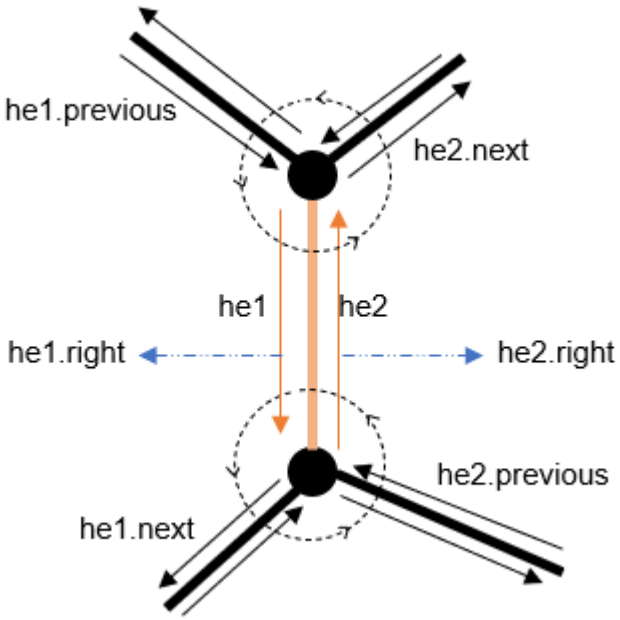

Figura 4.8 – Topologia entre arestas mantidas mesmo que uma nova aresta seja acrescentada ao modelo. (Fonte: Autor).

Com isso, a estrutura é capaz de representar todos os casos de ação do usuário com relação à criação de arestas e possíveis definições de zonas de forma automática, podendo descrever o modelo geométrico consistemente, dada que as condições de não intersecção de arestas seja seguida. Para isso, toda vez que uma aresta é criada verifica-se se ela apresenta alguma interseção com qualquer outra aresta.

Por fim, uma vez que um novo componente nova zona é definido, a primeira semi-aresta a ser percorrida (uma entre duas gêmeas) pode ser definido pela interface da classe *zone* de nome *is_clockwise()*. Esse método baseia-se na seguinte sequência de passos:

- Encontra-se o ponto com menor valores de coordenas x e y adjacente à zona a partir de uma travessia;
- Caso esse ponto tenha somente uma aresta adjacente a ele (ponta de uma trinca), considerar o próximo ponto;
- Encontra-se a primeira semi-aresta que possui esse ponto como destino;
- Simplifa-se esta semi-aresta e sua próxima em dois segmentos de retas, mesmo que aquelas sejam definidas por equações mais complexas
- O ângulo entre essas dois segmentos de reta determina o sentido da travessia: caso o ângulo seja anti-horário (a próxima semi-aresta está a esquerda), a travessia está sendo feita no sentido horário e
- O sinal deste ângulo pode ser obtido pelo determinante da matriz formada pelos dois vetores unitários com ângulo iguais aos das duas semi-arestas no ponto de encontro destas.

Essa solução, porém, só está correta caso o componente de zona seja adjacente a pelo menos três pontos (ou três semi-arestas). Caso aquele apresente apenas uma semi-aresta em sua borda, observa-se a existência de uma intersecção entre duas semiretas normais a essa em diferentes pontos. Se o componente for determinado por exatamente duas semi-arestas, verifica-se se alguma semireta normal a uma dessas intersecta a outra. Para os dois casos, a detecção de uma intersecção define que a zona está sendo atravessada no sentido anti-hoário.

4.2 Malha

A malha proposta pelo programa segue a definição de elementos quadráticos contínuos e descontínuos descrita por Delgado Neto (2017). Essa concepção tem seus fundamentos na equação de deslocamentos para a análises MEC padrão (GOMES, 2000) por elementos contínuos e no uso de análise incremental de Portela, Aliabadi (1992). Um exemplo de discretização por elemento contínuo pode ser visto na figura 4.9-a. Exemplos descontínuos são como os vistos na figura 4.9-b.

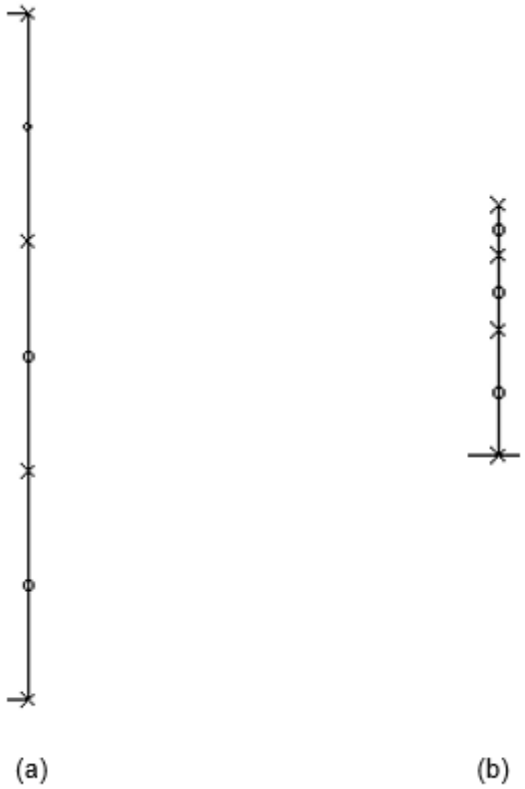

Figura 4.9 – Elementos de contorno discretizando uma aresta: (a) de forma contínua e (b) de forma descontínua. (Fonte: Autor).

O que está representado na figura 4.9 são os elementos de contorno que definem zonas e são utilizados pelas equações numéricas propostas pela bibliografia. Pode-se observar que um elemento está definido entre duas entidades em forma de cruz, e que cada elemento também possui um ponto médio (forma de círculo), formando assim um elemento quadrático.

Ao se restringir para o espaço de ação do programa, essas divisões foram adotadas como listas ordenadas que são atributos das semi-arestas definidas na seção 4.1. Com isso, na implementação, foi possível utilizar-se de uma das principais vantagens do MEC em comparação ao MEF: a reutilização de malhas já definidas quando há mudanças no modelo geométrico.

Além disso, para cada semi-aresta do modelo geométrico, a lista de elementos quadráticos de sua gêmea é composta por elementos que percorrem o caminho oposto de cada um dos elementos que definem a primeira, em ordem inversa. Isso permite um menor consumo de memória e confirma que, ao se percorrer uma trinca, o modelo proposto por Portela, Aliabadi (1992) e representado na figura 2.10 seja respeitado.

4.3 Condições de Contorno

As condições de contorno, por sua vez, são definidas por restrições e condições iniciais de deslocamentos, forças nodais e tensões aplicadas a elementos quadráticos. Essas condições são interpretadas como atributos dos elementos de contorno, formando assim uma relação hierárquica (como vista na figura 4.10) entre geometria, malha e condições de contorno.

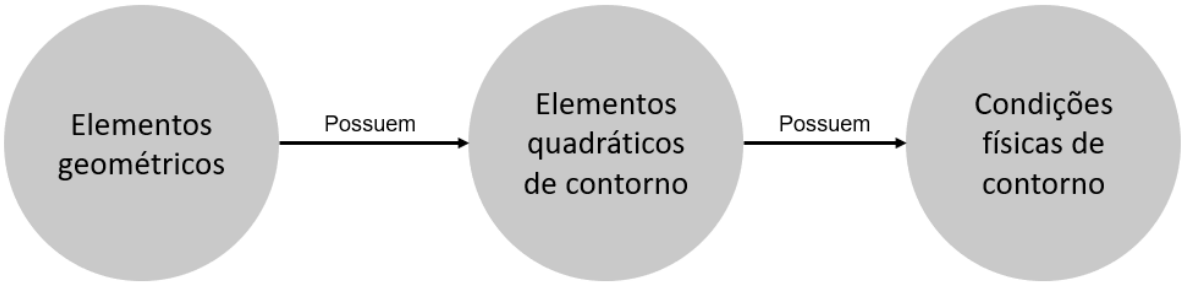

Figura 4.10 – relação hieráquica entre modelo geométrico, elementos de controle (malha) e condições de contorno. (Fonte: Autor).

As restrições e as condições iniciais de deslocamento, bem como a força aplicada em nós podem ser adicionadas ao modelo nos sentidos dos eixos principais do modelo (eixos globais das abscissas e das ordenadas). Já tensões podem ser aplicadas usando como referência tanto eixos globais como eixos locais. A definição de eixos locais está representada na figura 4.11 e pode ser escrita da seguinte forma:

- o eixo local x é definido pela direção e sentido do elemento quadrático e
- o eixo local y é perpendicular ao eixo local x.

Esses eixos, quando associados às ações sobre elementos, são equivalentes aos esforços cisalhantes e normais.

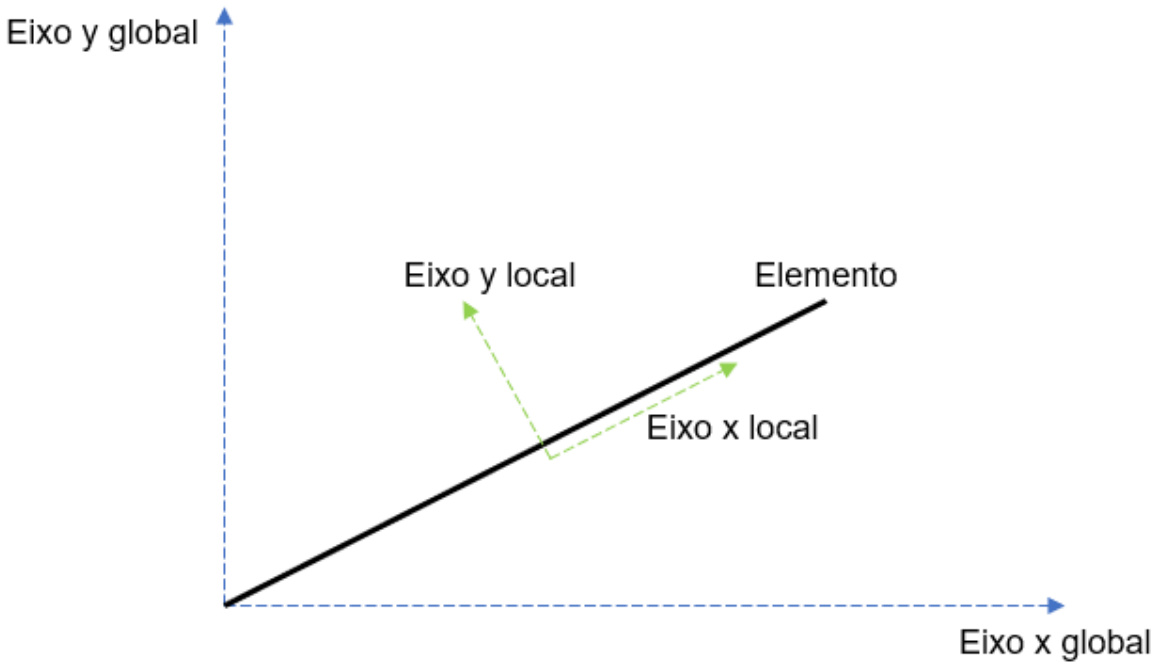

Figura 4.11 – Definição de eixos locais. (Fonte: Autor).

5. ORGANIZAÇÃO DO PROGRAMA

O programa, descrito a partir de uma visão global, é dividido em três pacotes, cada um referente a um conceito do padrão de código MVC (*Model-View-Controller*), como pode ser visto no diagrama de pacotes da figura 5.1. Como explicado, essa divisão permite que cada grande componente do código, o que deixa o programa mais manejável à medida que o tempo passa. Em outras palavras, caso queira-se alterar a aparência da interface gráfica, pode-se concentrar em alterar os conceitos pertencentes a *view* do programa, que estão logicamente e fisicamente separados dos outros conceitos.

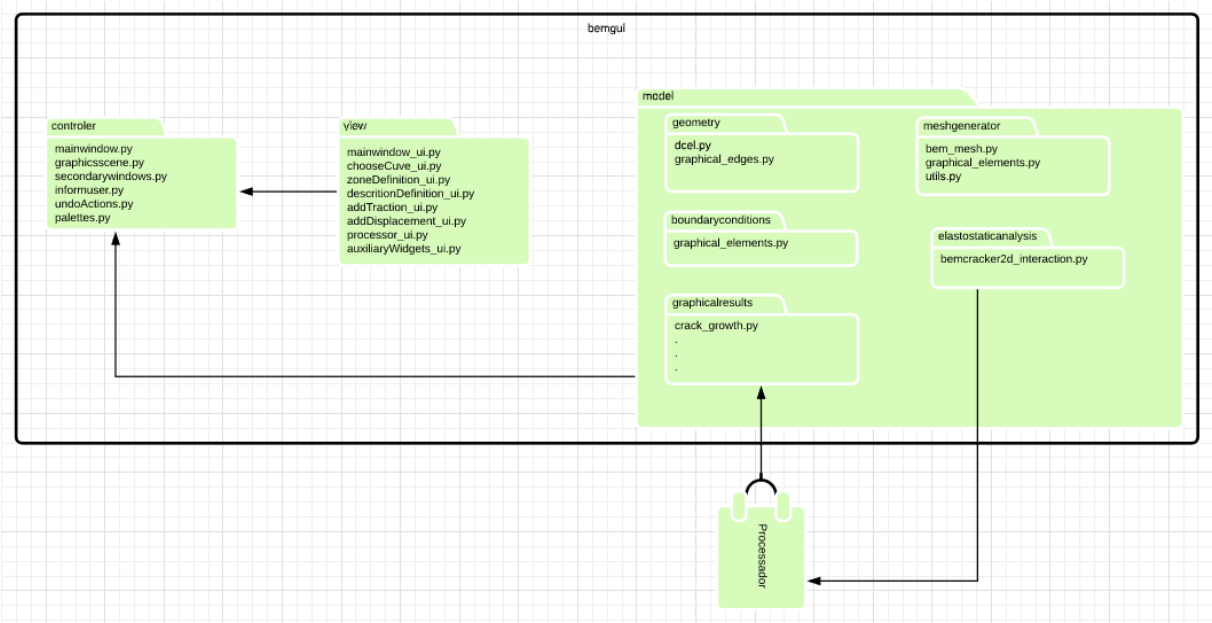

Figura 5.1 – Diagrama de pacotes do programa. (Fonte: Autor).

Por outro lado, caso queira-se adicionar ou alterar funcionalidades à interface, a pessoa responsável pela mudança pode-se concentrar em modificar os códigos presentes no pacote *controller* e considerar que as interfaces dos outros pacotes estão funcionando perfeitamente (SOMMEVILLE, 2008). Por fim, caso seja observado algum erro de lógica nas estruturas de dados utilizadas, é dedutível que se deve alterar funções presentes no pacote *model*.

A figura 5.2 mostra, ainda, a organização do programa em um sistema operacional.

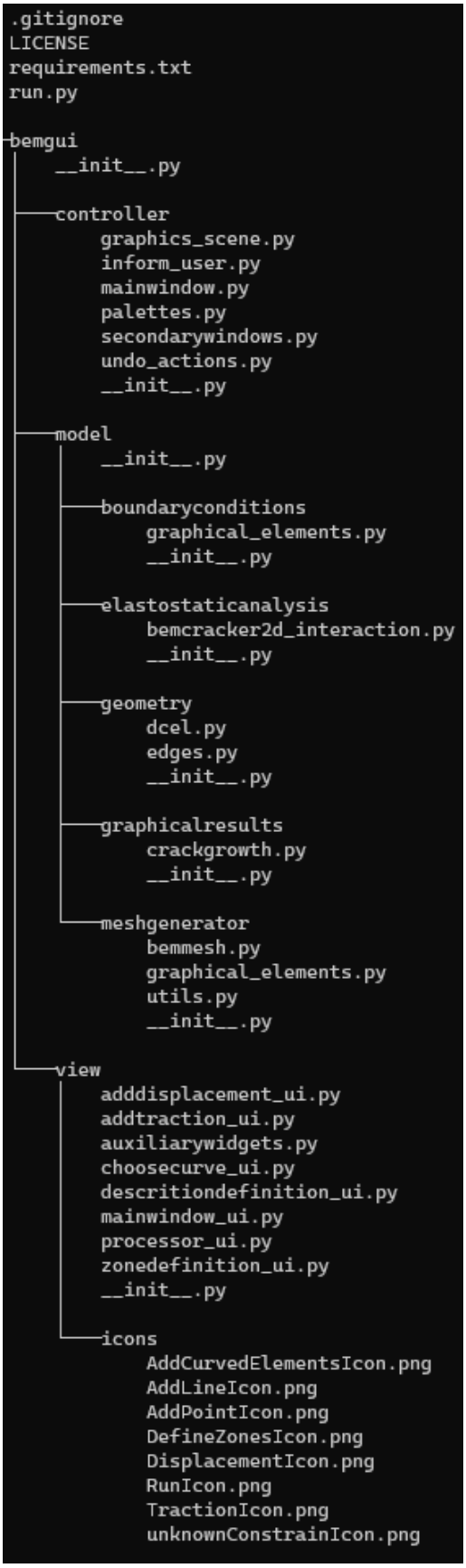

Figura 5.2 – Estrutura do programa organizada em diretórios (pacotes) e arquivos (módulos). (Fonte: Autor).

5.1 Pacote *view*

A figura 4.1 também mostra que o pacote *view* é formado por oito módulos, sete referentes a cada uma das janelas gráficas e um para *widgets* auxiliares. Os arquivos referentes às janelas contêm classes responsáveis por lidar com o aspecto visual de suas respectivas janelas. Mais especificamente, cada classe contém interfaces que, quando chamadas, criam objetos gráficos no formato de variáveis e definem localizações, nomes, *layouts*, ícones e outras características (puramente de visualização) desses objetos.

As funções apresentadas por essas classes, de nome descritivo, são duas: *setupUi* e *retranslateUi*. A primeira trata da disposição de objetos na janela a segunda define textos e *toolTips* (pequenas dicas responsivas ao movimento de cursor). As classes presentes neste pacote têm uma relação de parentesco por composição das classes presentes no pacote *controller*, como pode ser visto na figura 5.3.

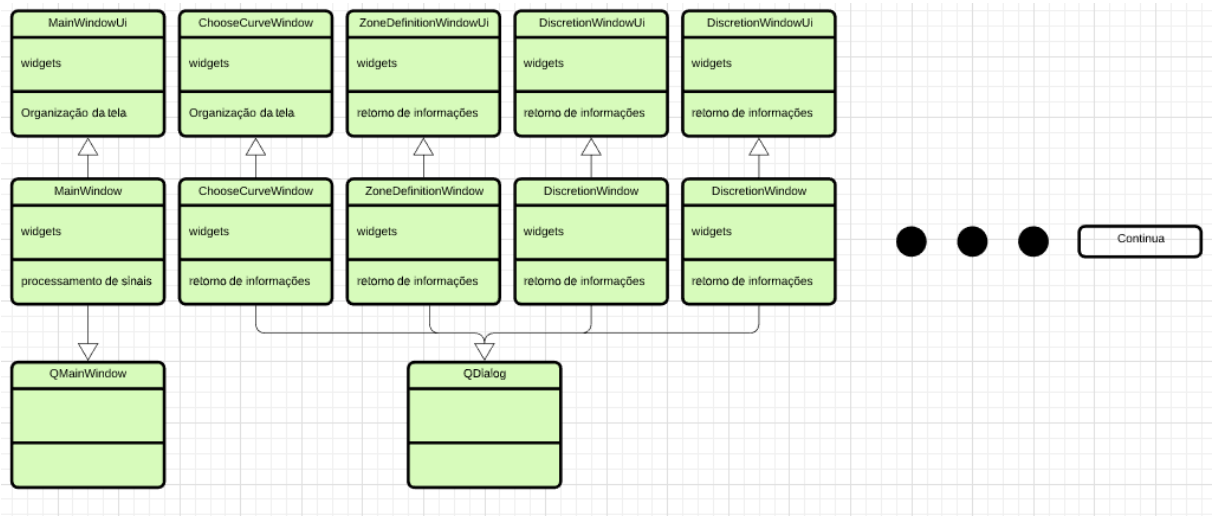

Figura 5.3 – Diagrama de classes do pacote *view* e um pouco de sua relação com o pacote *controller*. (Fonte: Autor).

A figura busca mostrar que cada janela é composta definida por uma parte gráfica (classes com final Ui) e a lógica, que faz parte do controlador. As classes que não terminam em Ui, portanto são uma composição entre as partes gráficas e de classes criadas pela biblioteca *PyQt*, acrescidas de lógica e atributos próprios.

O arquivo *auxiliaryWidgets.py* apresenta apenas duas classes que herdam comportamentos de *widgets* da biblioteca *PyQt* para uma melhor visualização em algumas janelas. Essas classes representam linhas editáveis que se comunicam diretamente para um melhor comportamento da interface (principalmente em janelas secundárias).

5.2 Pacote *model*

O pacote *model* é composto por 5 sub-pacotes, cada um contendo as estruturas e operações necessárias para as respectivas divisões da interface principal representada na figura 4.2. Nele são encontradas, por exemplo, os objetos que representam os elementos de discretização propostos para o MEC (DELGADO NETO, 2017).

5.2.1 Subpacote *geometry*

O primeiro sub-pacote implementa as abstrações descritas na seção 4.1. Assim, a partir dele, são apresentadas todas as ferramentas para a criação de um modelo geométrico, desde que obedeçam à restrição de que o grafo continue planar, o que também pode ser checado automaticamente. Para isso, o arquivo *dcel.py* contém as classes *Point*, *Edge* e *Zone*, que são os elementos básicos das duas estruturas descritas.

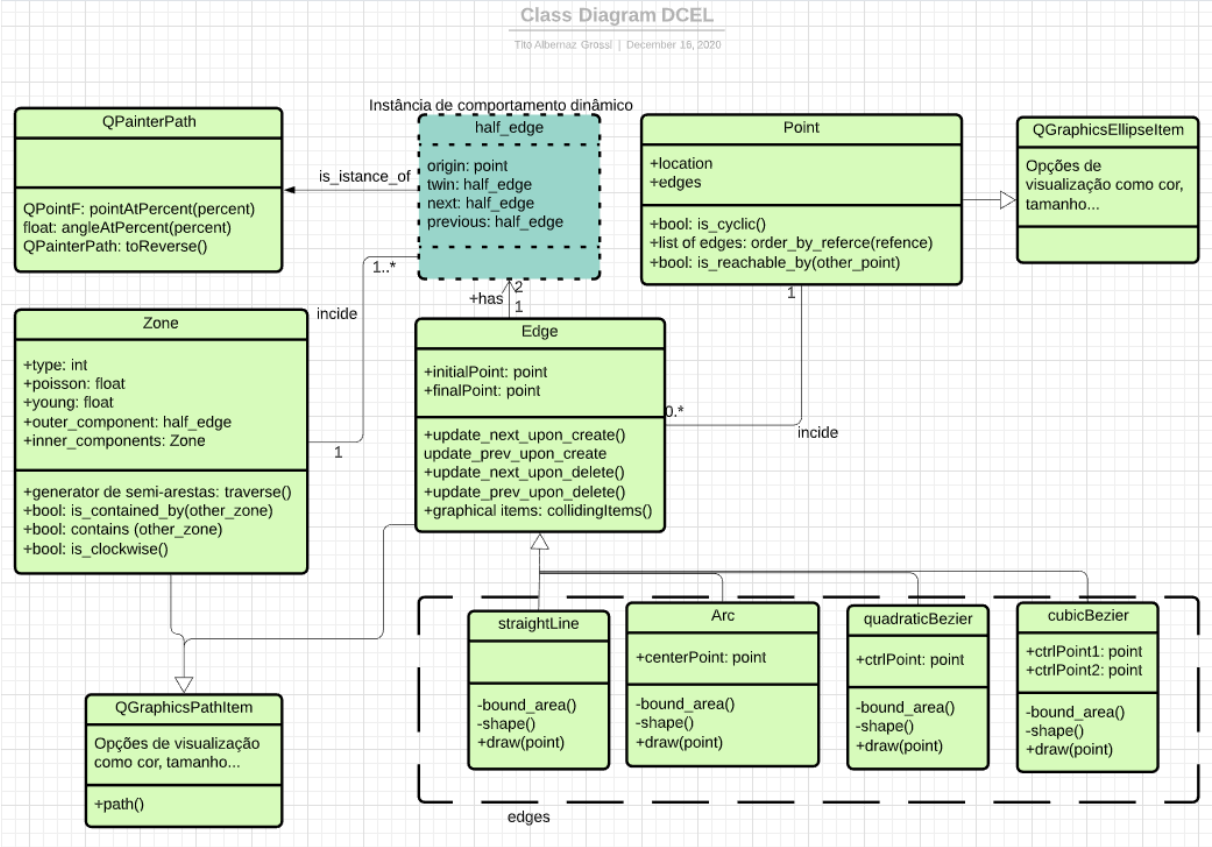

Figura 5.4 – Diagrama de classes do pacote model. (Fonte: Autor).

A lista de adjacência dos pontos usa um objeto padrão do *Python* chamado *set*, um conjunto de objetos. Essa representação foi escolhida por ser uma implementação de uma estrutura *hash*, que tem, as operações de inserir e retirar objetos e verificar a existência de determinado objeto

em seu interior em eficiência de tempo média constante, ou O(1) (DASGUPTA, VERIZANI, PAPADIMITRIOUS, 2008).

As classes *straightLine*, *Arc*, *quadraticBezier* e *cubicBezier* herdam todos os métodos e atributos da classe *Edge* e implementam suas próprias formas das funções *bound_area()* e *shape()*, utilizadas para identificação de intersecções com outras arestas (chamadas pelo método *intesects()* de sua classe pai). Além disso, cada uma dessas classes implementam um método *draw()* que possibilita a definição desses de maneira interativa, possibilitando o usuário de ver o que pode está sendo desenhado. Essa é outra mudança em relação ao programa BEMLAB2D, que não é responsivo desta maneira. Além disso, caso o caminho representado pela classe precise de mais pontos para sua definição (como um ponto central em um arco), a classe também é acrescida desses pontos em seu escopo.

As semi-arestas descritas na seção 2.2.2 não são implementadas por meio de uma classe, mas são instâncias (objetos) da classe *QPainterPath*, que buscam representar a trajetória percorrida por um *QGraphicsPathItem*, da qual as classes *Edge* e *Zone* são herdeiras. A diferença entre essas classes é que a primeira implementa um objeto gráfico que pode ser desenhado pela aplicação e a segunda implementa somente o caminho percorrido por essa, tornando-a natural para a ideia de uma semi-aresta assim como descrita. Apesar da classe *QGraphicsItem* possuir uma direção, ela é ignorada no contexto da aplicação para que represente um grafo não direcional.

5.2.2 Subpacote *meshgenerator*

O pacote responsável por definir o comportamento dos elementos de controle, bem como sua criação quando o modelo geométrico estiver definido é composto por três arquivos, a saber:

- *bemmesh.py* – adequa uma classe responsável pela criação da zona, através de determinados métodos de acordo com as zonas geométricas que são encontradas no modelo
- *graphical_elements.py* – mantém as classes de pontos e elementos que definem os elementos de controle. Os elementos são uma subclasse de *QGraphicsGroupItem* e mantem em seu grupo de itens duas linhas (conectando os pontos) e seu ponto do meio e ponto inicial. Como o ponto final de um elemento é o ponto inicial do próximo elemento na travessia da zona, esse ponto só está no escopo de um desses.

- *Utils.py* – Implementa algumas rotinas (funções) auxiliares para determinar localizações de pontos.

5.2.3 Subpacote *boundaryconditions*

Esse pacote contém apenas um módulo que abriga as classes para os objetos gráficos que implementam as condições de contorno do modelo. Essas classes são somente duas, *DisplacementConstrain* e *Traction*, as quais implementam métodos próprios para sua exibição gráfica e mantem informações sobre suas intensidades. É válido lembrar que cada elemento ou ponto definidor de elementos só pode manter uma instância de cada uma dessas classes (um ponto não pode estar submetido a uma força de 5kN e outra de 15kN, mas sim a uma de 20kN).

5.2.4 Subpacotes *elastostaticanalysis graphicalresults*

Os dois últimos pacotes devem apresentar métodos de interação com o processador em questão. O processador utilizado, BEMCRACKER2D, para recolher seus dados lê arquivos do disco rígido e faz o mesmo com os resultados gerados. Esses dois pacotes apresentam, portanto, funções que escrevem e leem arquivos. No entanto, como destacado no início deste trabalho, o segundo pacote desses não faz parte do escopo desta monografia.

5.3 Pacote controller

A lógica da aplicação e o controle de ações é processado no *controller*. Nele são encontrados arquivos que definem o comportamento do programa a partir da interação do usuário com a janela (*view*). O arquivo *mainwindow.py* apresenta uma classe filha de duas outras classes: *QMainWindow*

- *QMainWindow*: classe implementada pela biblioteca *PyQt*, responsável pelo envio e recebimento de sinais entre todos *widgets* “filhos” de uma instância daquela
- *MainWindowUi*: classe que representa os aspectos gráficos da classe, encontrada no arquivo *mainwindow_ui.py* do pacote *view*. Os métodos descritos na seção 5.1 são utilizados para que a janela possa ser renderizada.

Todos os *widgets* da aplicação, sejam eles janelas secundárias ou elementos simples como botões são filhos da janela principal e esta é o pai daqueles. Esse canal mútuo de comunicação permite que sinais possam ser passados de entre diferentes objetos gráficos. Quase todos os sinais envolvem a área de desenho central. Assim, quando um botão do módulo de geometria, por exemplo, é clicado, muda-se o estado da área de desenho, que passa a responder a eventos

de *mouse* correspondentes ao tipo de objeto sendo desenhado. Da mesma maneira, caso uma ação não possa ser realizada na área de desenho, esta envia um sinal para um *widget* presente na janela principal que aquela informação não pode ser realizada e o porquê. Esses *widgets* são implementados no arquivo *inform_user.py*.

A lógica das janelas secundárias está presente no arquivo *secondarywindows.py* e seguem o mesmo padrão de herança das janelas principal, possuindo um parentesco com forma gráfica presente no pacote *view* e com uma classe da biblioteca *PyQt*. Essa última, no entanto, passa a ser a classe *QDialog*, já que não há necessidade da presença de menus e barras de atividades, objetos construídos automaticamente para *QMainWindow*. De maneira simplificada, essas relações podem ser visualizadas na figura 5.3.

Essas classes apresentam, no entanto, um aspecto um pouco diferente: o uso de construtores alternativos, para que se possa habilitar ou não o uso de botões. Em *Python*, esse comportamento pode ser atingido por meio de *class methods*, que alteram alguma maneira o método construtor (método chamado ao se criar um objeto na memória) original da classe.

O arquivo *undo_action.py* implementa classes para armazenar representar comandos que podem ser armazenados em um histórico. Esse histórico permite ações de desfazer e refazer. Essa funcionalidade é uma vantagem se comparada aos trabalhos anteriores de Delgado Neto (2017) e Gomes (2000), onde um erro no processo de modelagem geométrica significa reiniciar o processo do zero.

Por fim, o arquivo *scene* contém a implementação da classe homônima *scene*, onde os objetos gráficos do modelo são validados e renderizados para serem visualizados no espaço visível de desenho da tela. Essa divisão de uma cena para armazenar os dados do modelo e uma janela gráfica para visualização desse é mais uma separação entre *view* e *controller*.

Apesar das estruturas de dados destacadas na seção 4.1 poderem se adaptar há basicamente qualquer modelo geométrico bidimensional e fornecerem ferramentas para o *controller* permitir ou não ações, o contexto do programa delimita a modelagem há alguns cenários. Isso se deve ao fato de o programa que trata do processamento do problema lidar apenas com uma zona definida por apenas um material.

Nesse sentido, o BEMCRACKER2D trabalha com a ideia de uma zona mestre que define o contorno ao qual todas as sub-zonas devem estar contidas. Ademais, o programa não trabalha com inclusões nesta zona mestre (já que isso definiria uma zona com mais de um material),

possibilitando-a ser definida apenas pela sua borda externa e por um número indefinido de furos.

Portanto, para a validação do modelo, dadas as condições da interface com os programas que o cercam, o controlador não permite a criação de inclusões através da desativação de certos botões. Além disso, convencionou-se que a primeira zona definida pelo usuário deve ser a zona mestre. A partir do momento que esta zona é definida, todas as outras devem estar contidas na zona mestre. Exatamente quando ela é definida também é verificada a existência de arestas fora de seu domínio. Essas são apagadas do modelo. Um exemplo de modelo válido para o programa de processamento pode ser visto na figura 5.5.

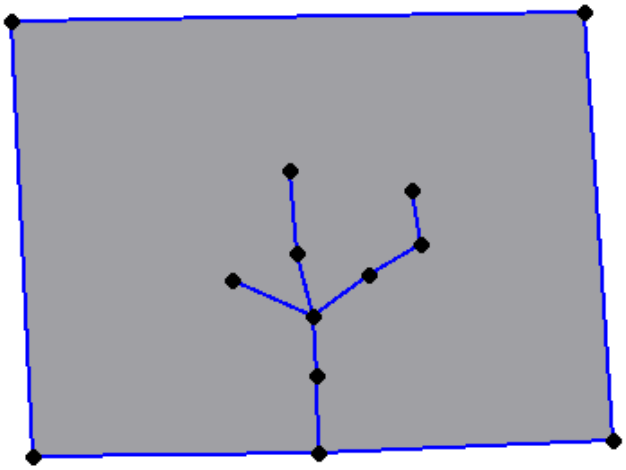

Figura 5.5 – Modelo possível de se construir no BEMGUI. (Fonte: Autor).

No entanto, pode-se observar na figura que o programa ainda permite a modelagem de trincas com múltiplas arestas, e que estas podem ser bifurcadas em quantos pontos e bifurcações se queira. Isso é uma vantagem sobre o programa BEMLAB2D, que possibilitava a modelagem de trincas com apenas uma aresta, como na figura 5.6.

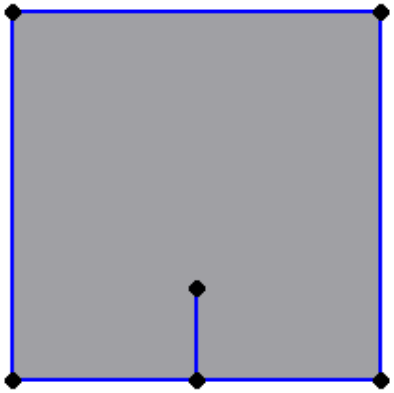

Figura 5.6 – Limitação do BEMLAB2D. (Fonte: Autor).

As outras zonas devem ser apenas do tipo furo. Portanto, nenhuma outra aresta deve estar contida em nenhuma zona que não a mestre, o que resulta em nenhum furo contido em um furo. Isso pode ser verificado automaticamente pelas ideias propostas na seção 4.1.

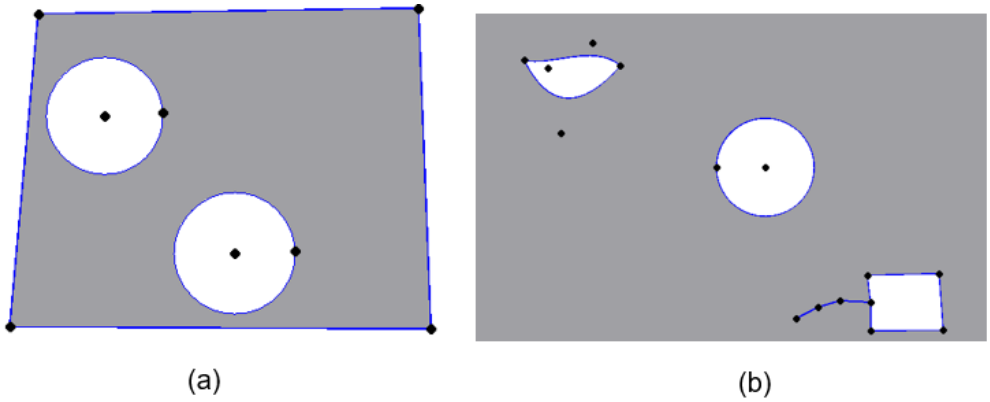

Figura 5.7 – Tipos de zonas possíveis de serem trabalhadas pelo BEMCRACKER2D: (a) zona mestre com borda externa bem definida e número indefinido de furos e (b) zona mestre com borda externa indefinida (zona infinita) e um número indefinido de furos. (Fonte: Autor).

Isso significa, simplesmente, que cada componente do grafo define, no máximo, um componente de zona. Portanto, modelos como os apresentados nas figuras 4.5, 4.6-a e 4.6-b não são permitidas pelo *controller*, apesar de ser possível modelá-los com a estrutura de dados implementada.

Devido a dinamicidade das interfaces apresentadas pelos objetos do pacote *model*, as ações não precisam ser tomadas com uma sequência definida, ao contrário do programa BEMLAB2D. No BEMGUI, o usuário pode definir pontos iniciais, adicionar arestas, e depois acrescentar ainda outros pontos, deixando o processo de modelagem mais flexível. Isso é ainda aprimorado considerando-se as funcionalidades de desfazer e refazer, que não descartam um modelo por completo quando um erro é cometido.

6. A INTERFACE BEMGUI E SUAS FUNCIONALIDADES

Como pode ser observado pela figura 6.1, a janela principal da aplicação é dividida em 5 seções, três localizadas à esquerda e duas localizadas à direita da área de desenho. Os três primeiros fazem parte, cada um, de uma etapa do processo de modelagem, e os outros dois têm como principal funcionalidade as interfaces de comunicação com programas de processamento. A aparência das janelas é inspirada principalmente no trabalho de Delgado Neto (2017).

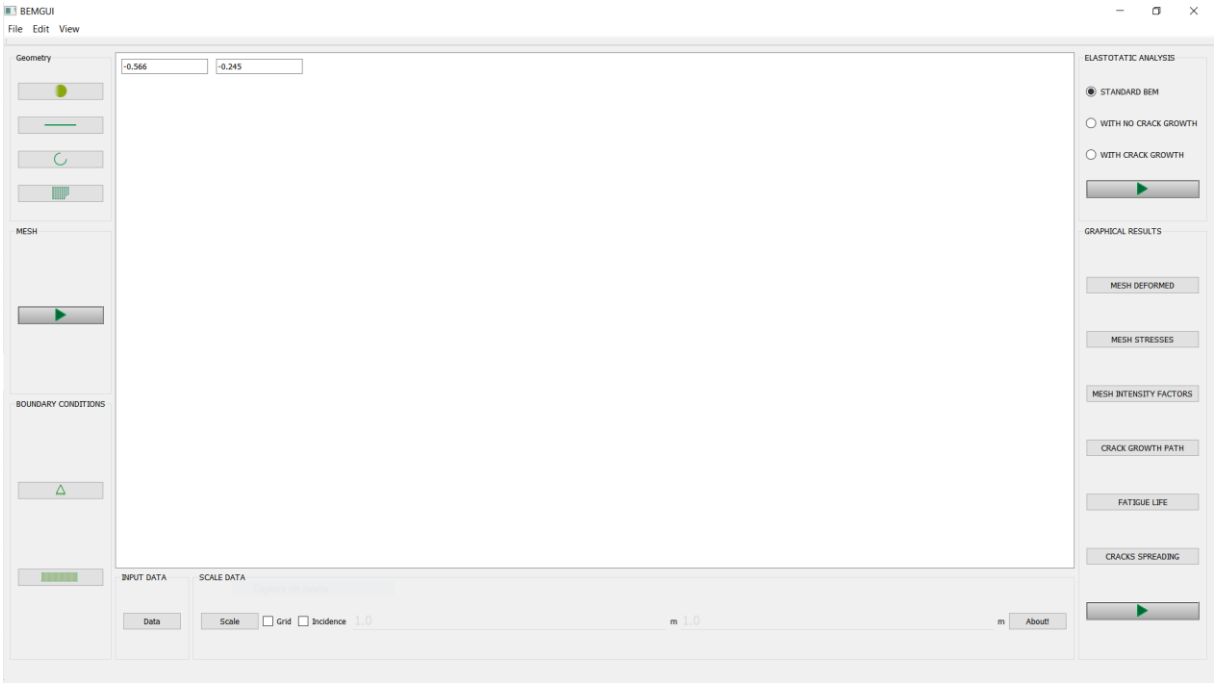

Figura 6.1 – Janela principal da interface gráfica. (Fonte: Autor).

6.1 *Geometry* (Geometria)

A primeira seção trata da geometria a ser definida e é composta por 4 botões, a saber: ponto, elemento reto, elemento curvo e zona. A figura 6.2 apresenta com maiores detalhes os botões que compõem esse primeiro módulo, bem como a *tooltip* (ferramenta de dica) presente no segundo botão dessa seção.

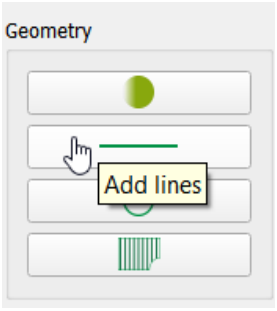

Figura 6.2 – Seção *geometry*. (Fonte: Autor).

Neste módulo os botões funcionam da seguinte forma: cada clique de botão envia um sinal da classe *pyqtsignal* para a área de desenho, que então muda o seu estado. Isso muda o que a cena realiza com cada interação, principalmente no que se refere ao *click* de *mouse*, que adiciona novos elementos ao modelo, desde que a adição desse seja válida.

6.1.1 BOTÃO “PONTO”

O botão ponto informa a cena em que, ao receber um *click*, um ponto deve ser acrescentado ao modelo. Esse ponto é definido pelas coordenadas do cursor, mas há um segundo modelo de adição de pontos, caso esta ação esteja sendo realizada: para uma maior precisão na hora da definição de localizações, o usuário pode digitar o valor com até três casas decimais e após definir as duas coordenadas com o uso da tecla *enter* (ou retorno), um ponto é adicionado à cena e ao modelo.

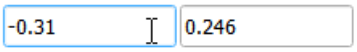

Figura 6.3 – Linhas editáveis referentes às coordenadas do *mouse*. (Fonte: Autor)

Para melhor visualização deste processo, as linhas editáveis que, usualmente, mostram a localização do cursor, reagem às teclas pressionadas (desde que numéricos) e mostram os valores digitados, primeiro o valor do eixo x e depois do eixo y (figura 6.3).

6.1.2 BOTÃO “SEGMENTO RETO”

As arestas que constituem um segmento reto podem ser adicionadas após esse botão ser clicado. Como previamente definido, o segmento reto deve conectar dois pontos. Dessa maneira, quando um elemento deste tipo estiver sendo adicionado e o *mouse* passar sobre algum botão, este é destacado por uma outra cor, como pode ser visto na figura 6.4.

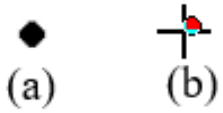

Figura 6.4 – Exemplos de objeto ponto: (a) em seu estado natural; (b) executando função de *snap* quando o *mouse* está sobre ele. (Fonte: Autor).

6.1.3 BOTÃO “SEGMENTO CURVO”

Com o *click* sobre este botão uma janela como a da figura 6.5, é apresentada ao usuário, que deve escolher qual tipo de segmento será desenhado. De maneira semelhante ao desenho de segmentos retos, pontos que estão posicionados sob o cursor são destacados. O desenho de cada uma das arestas também segue o movimento do *mouse* do usuário, facilitando a visualização do que está de fato sendo modelado. De maneira geral, os passos a serem seguidos pelo usuário ao se modelar uma nova aresta pode ser representado pelo diagrama de estados da figura 6.6.

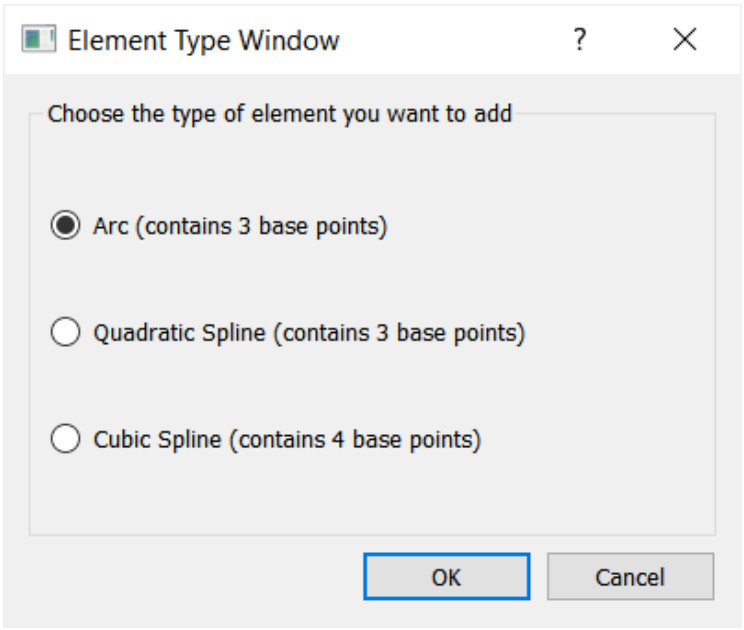

Figura 6.5 – Janela secundária para escolha de tipo de curva para ser adicionada ao modelo. (Fonte: Autor).

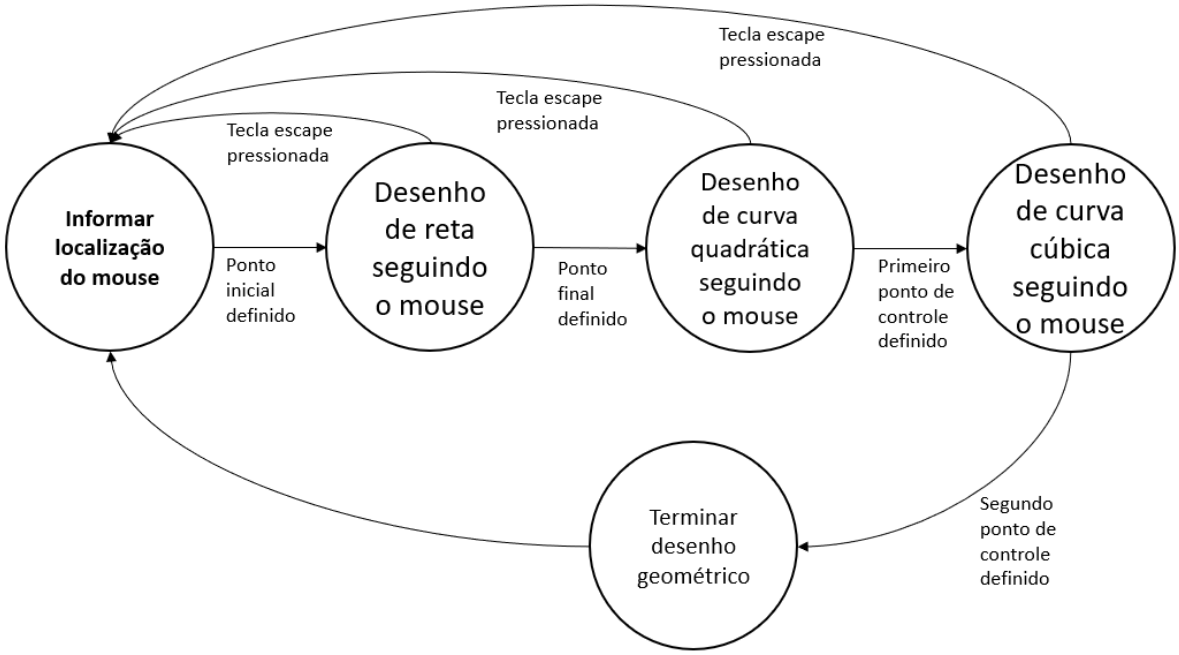

Figura 6.6– Diagrama de estados para a criação de aresta do tipo Bézier cúbica. (Fonte: Autor).

A figura acima mostra os diferentes caminhos a serem tomados para a definição de uma curva cúbica de Bézier, que exige quatro pontos. Diagramas semelhantes podem ser criados para a criação de cada um dos outros tipos de arestas, respeitando-se o número de pontos que cada uma exige. Nota-se que, a qualquer etapa do desenho o usuário pode desistir da criação da nova aresta pressionando o botão "*escape*", que retirará o elemento da cena e não fará mudanças na topologia do modelo.

6.1.4 BOTÃO "ZONA"

O botão zona indica que zonas serão selecionadas para a redefinição de algumas de suas propriedades. Caso uma zona seja selecionada, a janela da figura 6.7 se apresenta ao usuário, que pode redefinir as características físicas da zona (seu módulo de elasticidade e coeficiente de *poisson*) ou mesmo sua orientação ou tipo de zona.

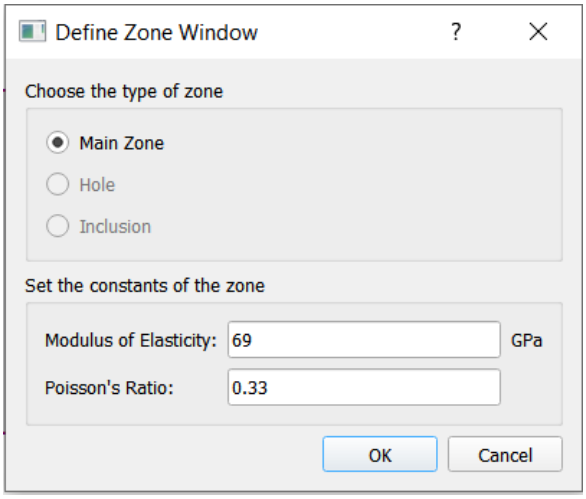

Figura 6.7 – Definição das características físicas de uma zona. (Fonte: Autor).

É importante ressaltar que essa janela também é apresentada toda vez que o contorno de um novo componente de zona é definido (novamente, identificado de maneira automática).

6.2 *Mesh* (Malha)

O módulo de malha trata da discretização da geometria criada na seção acima. Ele é composto por um apenas um botão que inicia o processo de criação de malha. Esse módulo pode ser visto na figura 6.8.

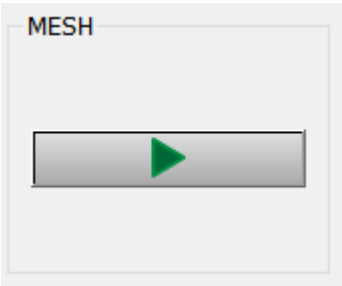

Figura 6.8– Seção *mesh* (Fonte: Autor)

Esse botão faz com que o programa peça que o usuário informe quantos elementos cada aresta terá. Para isso, são percorridas as zonas, por cada um dos seus objetos de contorno (que são destacados quando estão sendo discretizados) e, além da quantidade de elementos, o tipo de discretização também é definido a partir da caixa de diálogo da figura 6.9.

O tipo de discretização se refere à discretização contínua (primeira opção) ou a discretização descontínua (segunda opção). É válido lembrar que trincas também são identificadas de maneira automática pelo programa, e que só são aceitos elementos descontínuos para esses. Portanto,

caso seja identificado que a aresta discretizada seja referente a uma trinca, a janela aparece como a figura 6.9-b, permitindo somente a segunda opção.

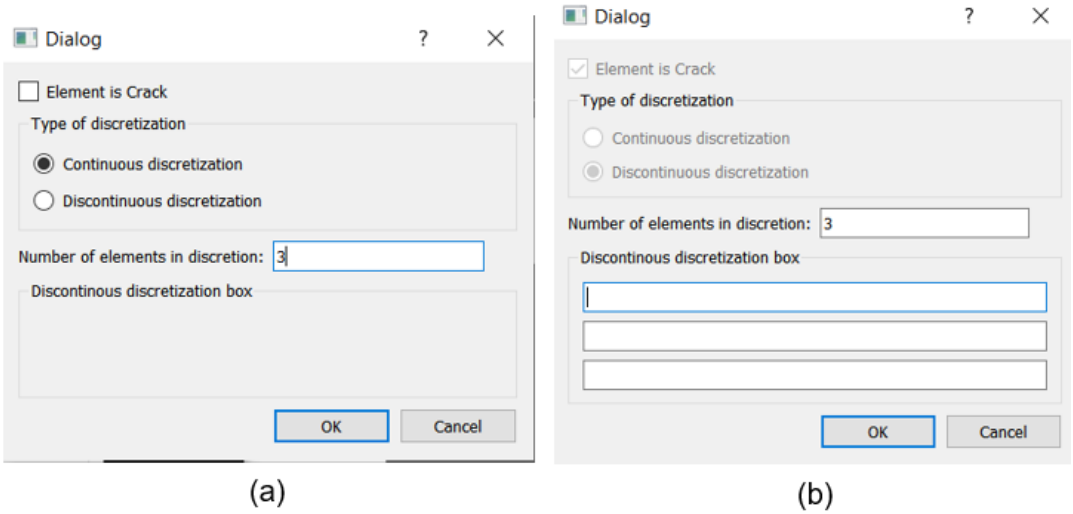

Figura 6.9 – Janela secundária para escolha do tipo e quantidade de elementos: (a) quando se percorre uma aresta que não define uma trinca e (b) quando é identificada uma trinca. (Fonte: Autor).

Nesse processo, são definidos elementos quadráticos, que são compostos por um ponto inicial, um ponto final e um ponto médio, e o usuário é capaz de definir se gostaria de discretizar cada objeto de contorno por elementos contínuos ou descontínuos, com exceção a trincas, que são obrigatoriamente elementos descontínuos.

Quando o objeto é discretizado de forma descontínua, o usuário deve definir as razões entre os elementos. Mesmo com essa liberdade, o programa garante que a soma das razões seja igual a um, deixando um valor mínimo para cada uma das entradas das razões.

6.3 *Boundary Conditions* (Condições de Contorno)

Esta seção é constituída pelos quatro botões mostrados na figura 6.10 e usados para a definição de condições de contorno, destacando-se os dois primeiros elementos. O primeiro define as restrições e condições iniciais de contorno e o segundo as forças e tensões a que cada elemento está submetido. Para que eles funcionem, no entanto, é necessário primeiramente selecionar

pelo menos um elemento da malha modelada. É obtido disso, portanto, que essa seção do programa só tem relevância uma vez que uma malha é definida.

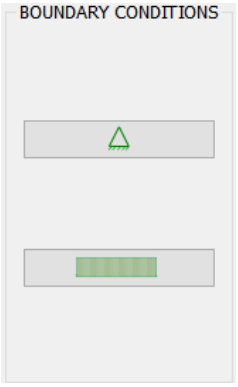

Figura 6.10 – Seção *boundary conditions*. (Fonte: Autor).

De maneira semelhante aos pontos do modelo geométrico, os elementos quadráticos são destacados quando o *mouse* está sobre eles (figura 6.11-a, com o elemento com uma cor vermelha). Quando o botão do *mouse* é pressionado com o cursor sobre um elemento, ele é selecionado e assume ainda outra cor. Caso o usuário queira selecionar múltiplos elementos, basta que se pressione a tecla "*control*" em conjunto com os botões do *mouse* (figura 6.11-b). Dessa forma, podem ser adicionadas as mesmas condições de contorno para vários elementos pelo mesmo comando (figura 6.11-c).

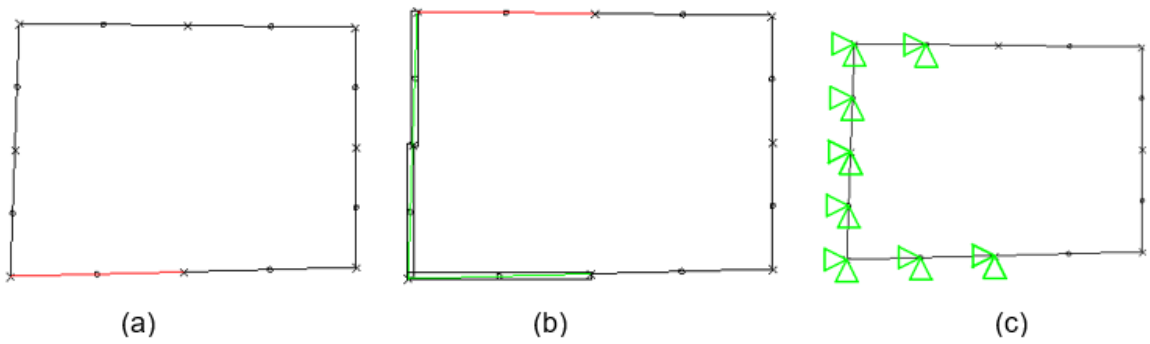

Figura 6.11 – Seleção de elementos quadráticos para adição de condições de contorno: (a) destaque de elemento quando cursor está sobre este; (b) seleção de múltiplos elementos é possível e (c) as condições de contorno são aplicadas à todos os elementos selecionados

A seleção de quais condições serão aplicadas aos elementos selecionados pelas janelas da figura 6.12-a (para restrições de deslocamentos e deslocamentos iniciais) e 6.12-b (para forças nodais e tensões). Ressalta-se que o usuário pode definir que as mesmas restrições podem ser definidas

para os pontos iniciais e "médios" dos elementos, selecionando-se a caixa de checagem "*Repeat for middle point*", o que dá uma facilidade maior ao usuário neste processo.

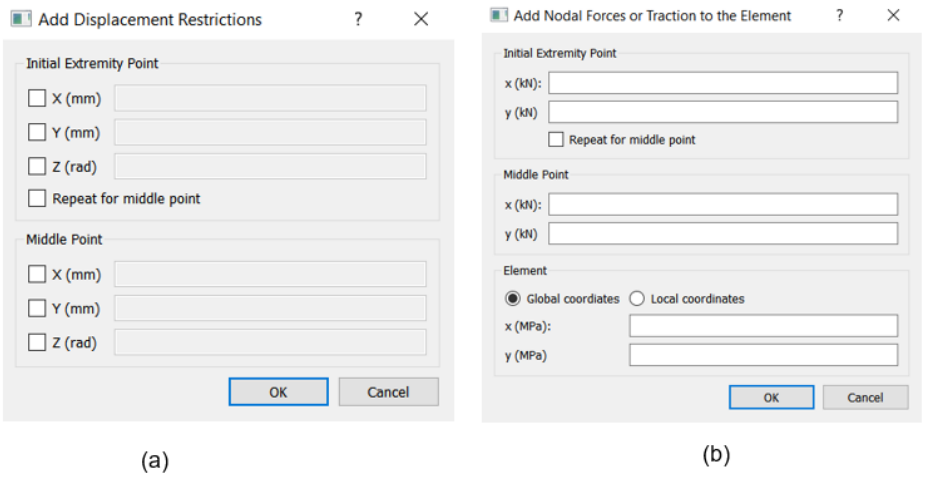

Figura 6.12 – Janelas secundárias para definição de condições de contorno: (a) deslocamento e (b) esforços. (Fonte: Autor).

6.4 Elastostatic Analyisis (Análise Elastostática)

O módulo de análise elastostática tem o propósito de gerar os resultados numéricos a partir da interação com o BEMCRACKER2D. Dessa forma, seus botões geram os arquivos de entrada daquele, informando o método de análise de escolha do usuário, a posição dos pontos definidos na malha e a topologia desses (as conexões de cada elemento) e as características elásticas e físicas das zonas modeladas.

Os métodos de escolha disponíveis são relacionados às aplicações do BEMCRACKER2D descritas na seção 2.5 deste trabalho: MEC padrão, Sem Propagação de Trincas e Com Propagação de Trincas (eles podem ser alterados por meio de *radio buttons* presentes no módulo). Cada uma dessas opções pode ser vista na figura 6.13.

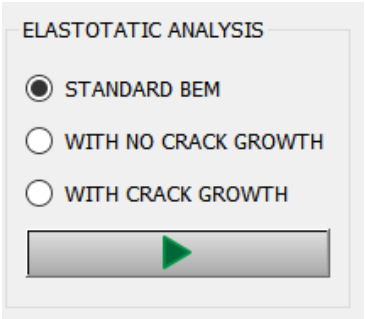

Figura 6.13 – Seção *elastostatic analysis*. (Fonte: Autor).

Já o botão final deste módulo chama a caixa de diálogo presente na figura 6.14 para que o usuário possa fornecer as informações faltantes no modelo (como a constante de Paris). Essa

por sua vez, retorna um arquivo que apresenta as informações necessária para a avalição de um problema elastoplástico via Método dos Elementos de Contorno. Este arquivo é o objetivo final do presente trabalho.

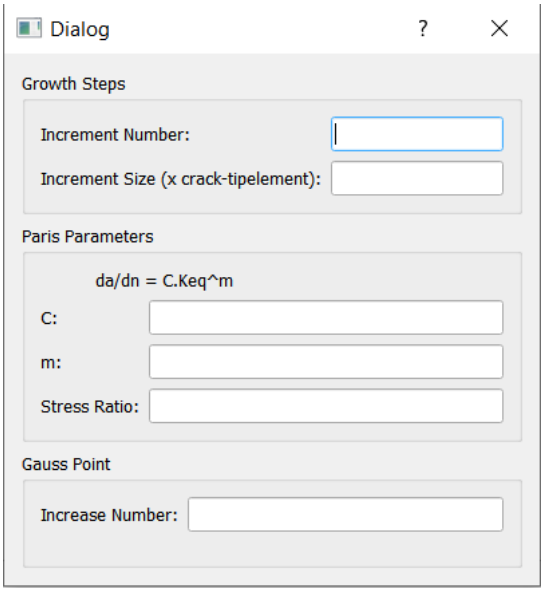

Figura 6.14 – Caixa de diálogo para definição de últimos dados do problema. (Fonte: Autor).

A caixa de diálogos da figura acima apresenta todos os objetos interativos apenas quando a opção “com crescimento de trincas” é selecionada. Caso contrário, o usuário define apenas o número de pontos de Gauss (último elemento na tela).

6.5 *Graphical Results* (Resultados Gráficos)

O módulo de resultados gráficos é o último dos presentes na interface gráfica, e também interage com o programa BEMCRACKER2D, mas de maneira contrária ao módulo anterior. Aqui, pretende-se usar os arquivos de saída do BEMCRACKER2D para gerar formas de visualização gráficas dos resultados numéricos apresentados pelo pacote.

Essas informações serão passadas por meio de gráficos interativos, animações e outros formatos de visualização de dados. No entanto, esta seção foge ao escopo desta monografia, e, com exceção da parte gráfica, não foi implementada.

6.6 Parte Informativa do Programa

Por fim, se o programa detectar que, ao usuário finalizar a definição de uma nova aresta, ela está quebrando as regras do programa (conectando dois componentes de grafos diferentes ou gerando intersecções fora dos pontos, por exemplo), avisos são fornecidos do porquê daquela ação não ser suportada.

Dessa maneira, o usuário pode saber a razão de uma resposta não esperada e não a repetir. As figuras a 6.15 e 6.16 mostram alguns exemplos disso.

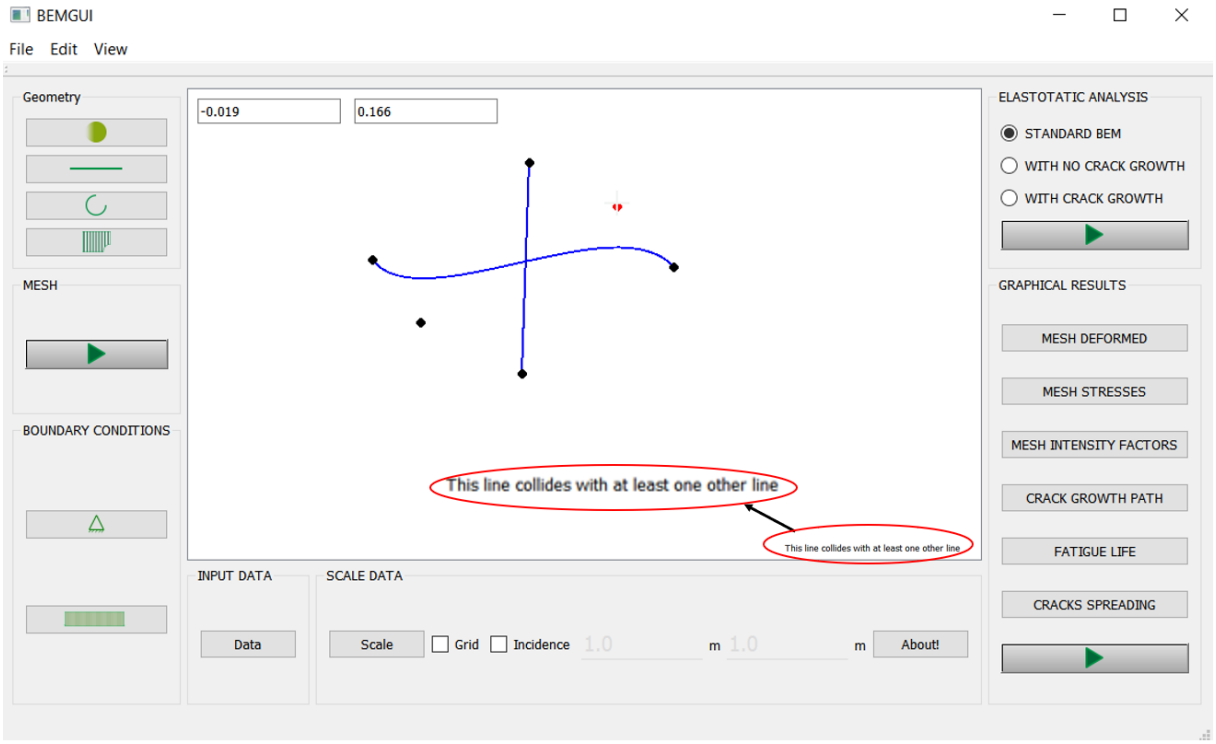

Figura 6.15– Programa lidando com intersecção entre arestas

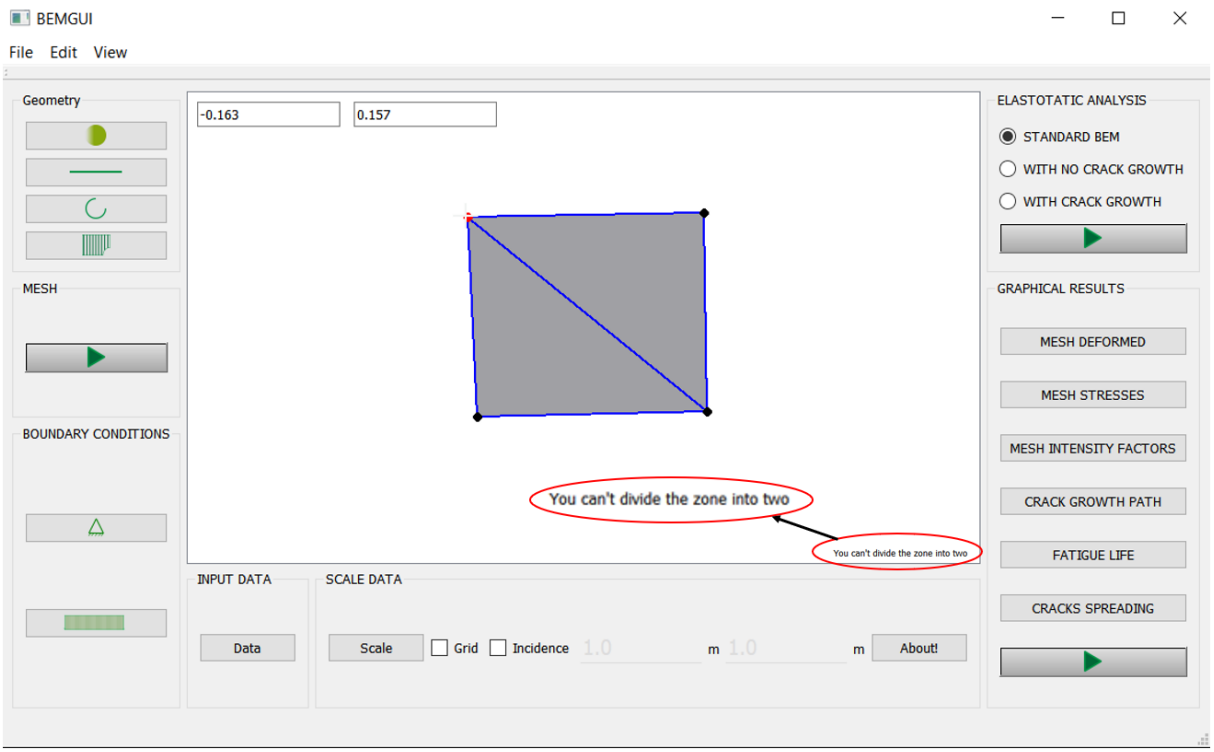

Figura 6.16 – Programa lidando com divisão de zonas

7. APLICANDO O BEMGUI

Nesta seção serão aplicados ao ambiente do BEMGUI alguns problemas previamente explorados na literatura, especialmente os propostos por Delgado Neto (2017). No entanto, o primeiro exemplo conterá um passo-a-passo do processo de modelagem de malha de elementos de contorno de um problema com o uso de *splines*, já que esses modelos ainda não podiam ser construídos previamente.

Deve-se notar, no entanto, algumas mudanças nas imagens em que a interface gráfica está representada, se comparadas com imagens das outras seções deste trabalho. Mais especificamente, ela apresenta botões que estavam presentes em versões anteriores do programa, quando os modelos foram construídos.

7.1 Exemplo de modelagem com passo-a-passo detalhado

Propõe-se aqui modelar a borda externa do desenho apresentado na figura 2.8 do presente trabalho, que pode ser melhor visualizada na figura 7.1 a seguir. Como essa é uma zona extremamente complexa de ser definida, faz-se necessário o uso de *splines*. Porém, como não se possui a descrição perfeita do contorno do desenho, este será modelada a partir de uma ideia considerada próxima o suficiente da realidade, apenas como demonstração do uso das ferramentas da interface.

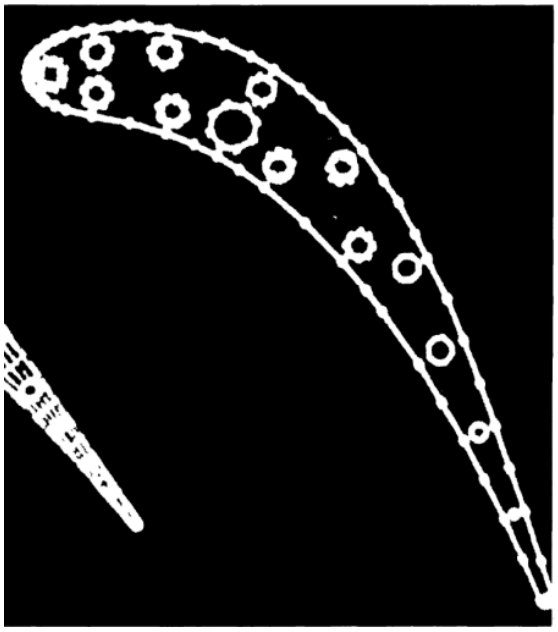

Figura 7.1: Representação do desenho a ser modelado. (Fonte: Adaptado de Brebbia; Domínguez, 1994)

Pretende-se definir o contorno da hélice a partir de dois segmentos cúbicos (que descreverão os elementos “laterais” maiores) e dois segmentos quadráticos (que descrevem as duas “pontas” do objeto real).

O processo de modelagem no BEMGUI começa com elementos do tipo ponto. Dessa maneira, primeiramente são definidos a localização dos pontos com o clique do *mouse* ou mesmo com o uso do teclado. Essa fase do modelo está ilustrada na figura 7.2.

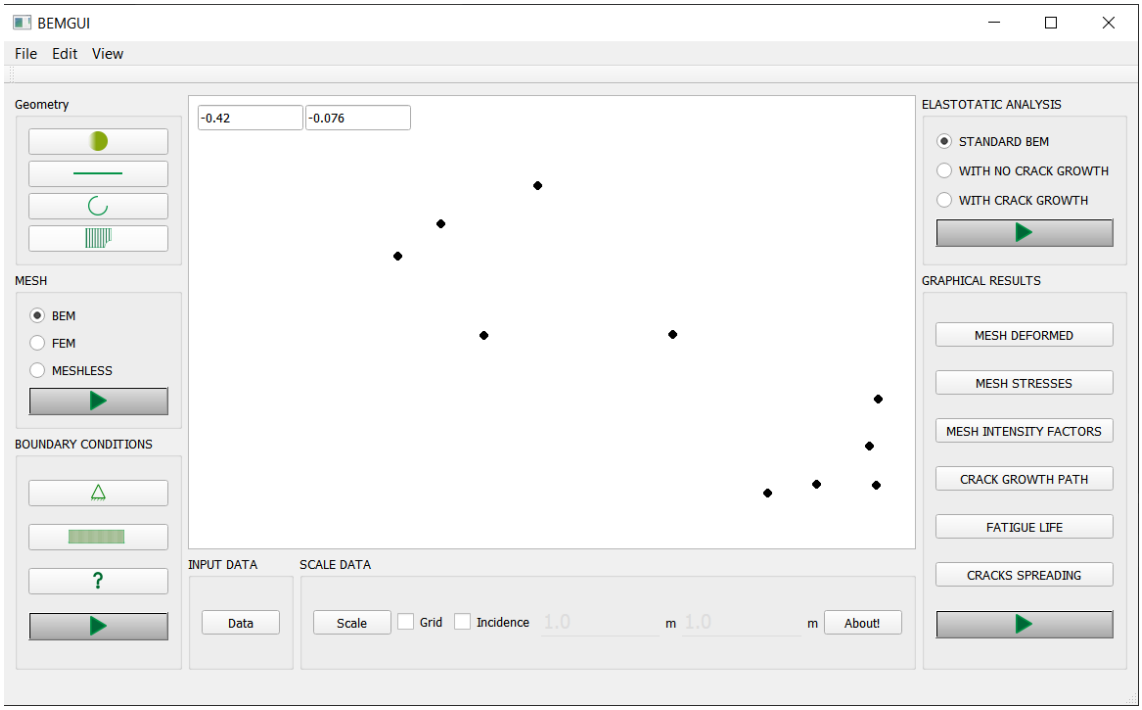

Figura 7.2 – Disposição dos elementos de ponto para modelagem de hélice. (Fonte: Autor).

Após isso, seleciona-se o botão “segmento curvo”, descrito no tópico 6.1.3, que oferece ao usuário a janela ilustrada na figura 6.6, onde escolhe-se a opção de elemento cúbico. A figura 7.3 mostra o modelo com os dois segmentos cúbicos adicionados.

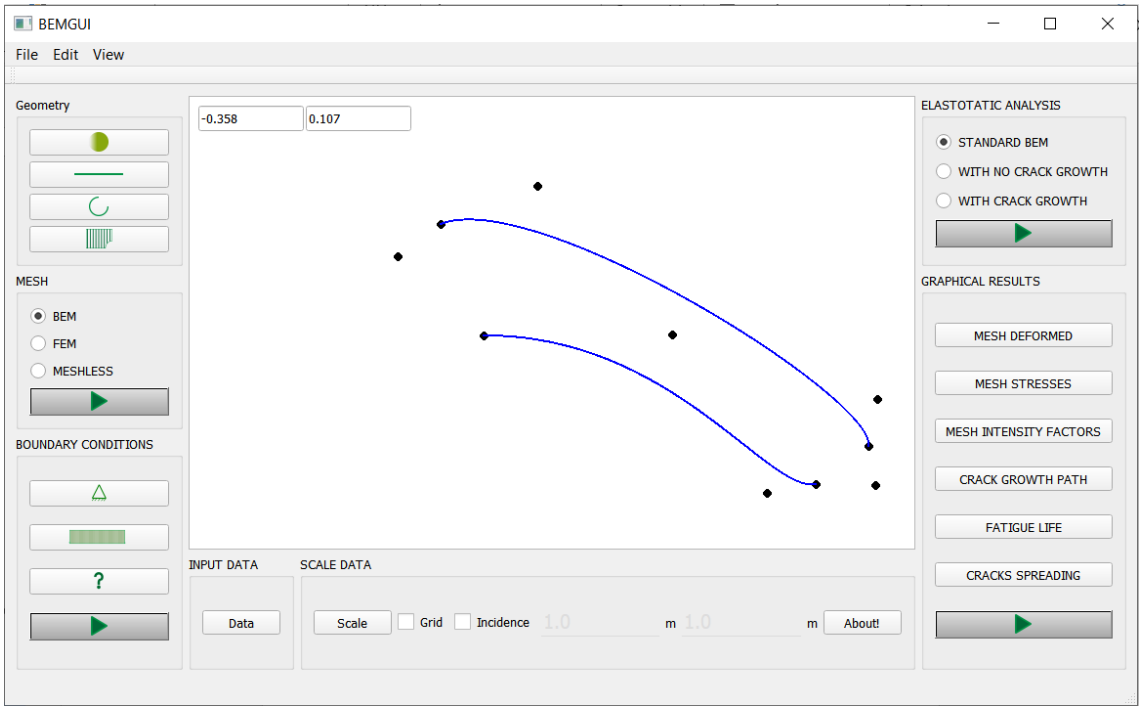

Figura 7.3 – Definição dos dois primeiros elementos do modelo (elementos quadráticos). (Fonte: Autor).

Deve-se, agora, voltar ao botão de elementos curvos, o que leva o usuário à caixa de diálogos. Escolhe-se, no entanto, a opção de elemento quadrático. Adiciona-se à área de desenho os dois elementos de ponta faltantes a partir de seus três pontos base.

Após a adição do segundo elemento, a janela de definição de propriedades de zonas será executada e é possível informar ao programa o módulo de Young e o coeficiente de Poisson do material que compõe a zona. É válido lembrar que, como esta é a primeira zona fechada, ela será considerada a zona mestre. Na figura 7.4 pode-se ver uma imagem da zona (modelagem geométrica completa).

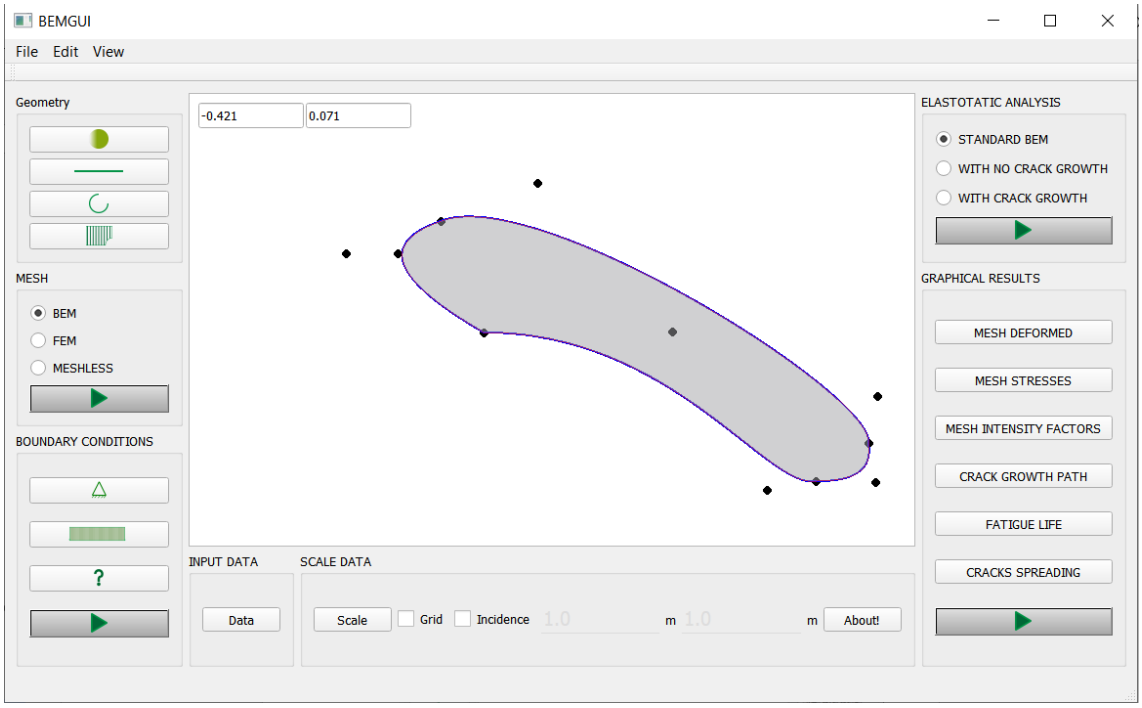

Figura 7.4 – Modelo geométrico de hélice. (Fonte: Autor).

Com isso tem-se a definição da zona desejada, e o processo de formação de malha pode ser iniciado. Para tanto, basta clicar no botão "*Run Mesh*" ("criar zona"), que a irá percorrer elemento de contorno por elemento de contorno, pedindo que o usuário indique a forma de discretização de cada um desses, a partir da caixa de diálogo presente na figura 7.5.

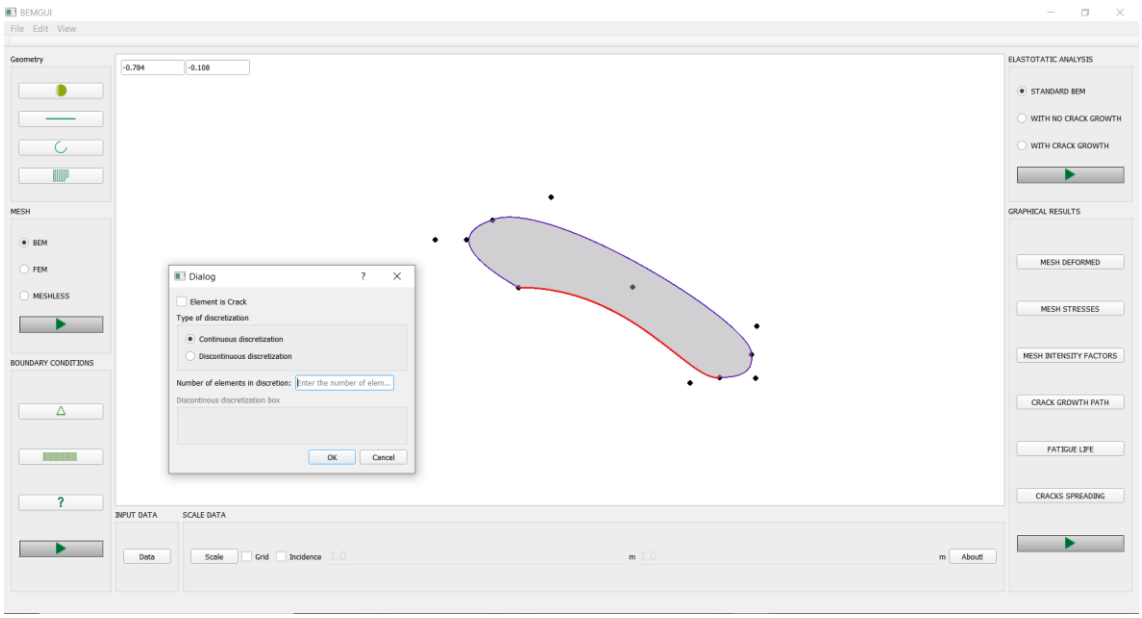

Figura 7.5 – Exemplo de definição de discretização de elemento geométrico. (Fonte: Autor).

Nesta caixa é possível definir se a discretização será contínua (seguimentos de espaçamentos iguais) ou descontínua (seguimentos de espaçamentos diferentes), caso em que o usuário deve

entrar com os valores das proporções de cada elemento, respeitando o valor mínimo de 0,1 e soma entre todos os valores igual a um.

A título de exemplo escolheu-se, para o modelo em questão, 5 elementos para as curvas cúbicas e 3 para as curvas quadráticas, todos com espaçamento contínuo. Isso faz com que a geração da malha seja finalizada. Sua visualização pode ser conferida na figura 7.6.

Figura 7.6 – Malha gerada para proposição de modelo de hélice. (Fonte: Autor).

A seguir, para intuitos de exemplificação, serão adicionados a elementos aleatórios algumas condições de superfície. Para isso, seleciona-se os elementos desejados e, a partir das caixas de diálogo da figura 6.12, adiciona-se restrições de deslocamento e esforços externos àqueles. O resultado pode ser visto na figura a seguir

Figura 7.7 – Exemplo de modelo final de hélice. (Fonte: Autor).

Ao final, pode-se clicar no botão "*Run Elastostatic Analysis*" para criação do arquivo de entrada do BEMCRACKER2D, que pode ser conferido no apêndice A. Esse apêndice foi adicionado ao trabalho com a finalidade de demonstrar como as informações topológicas são passadas ao programa processador. O usuário pode definir o nome e a localização do arquivo a partir de uma janela de navegação de pastas do sistema operacional utilizado.

7.2 Chapa Cruciforme com Trinca Inclinada

O modelo geométrico deste exemplo está representado na figura 7.8, e a zona é definida por apenas pelo seu contorno externo (sem furos), que tem a geometria de uma placa cruciforme, além da adição de uma trinca inclinada em uma de suas extremidades.

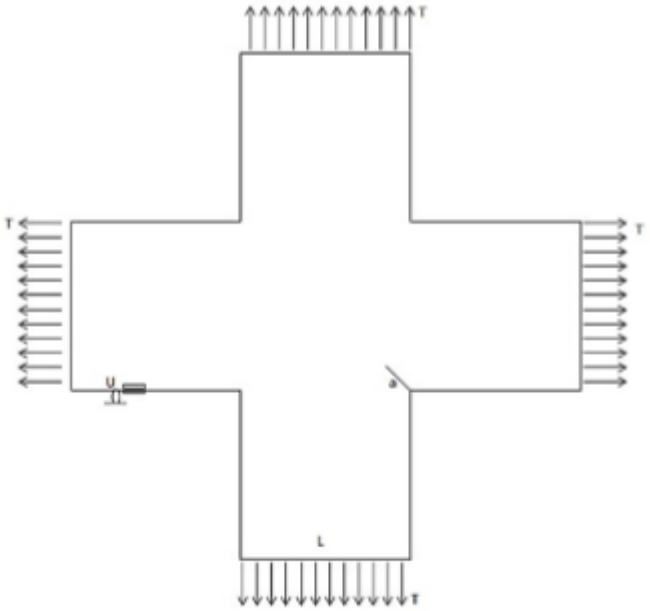

Figura 7.8 – Modelo geométrico da placa cruciforme. (Fonte: Autor).

Para a construção do modelo, cria-se, primeiramente os objetos pontos a partir das linhas editáveis na cena, para maior precisão. A disposição dos pontos pode ser verificada na figura 7.9.

Figura 7.9 – Disposição de pontos para formação de placa cruciforme trincada. (Fonte: Autor).

Após isso, foi reconstruído o modelo dentro do ambiente do BEMGUI. Essa representação pode ser observada na figura 7.10.

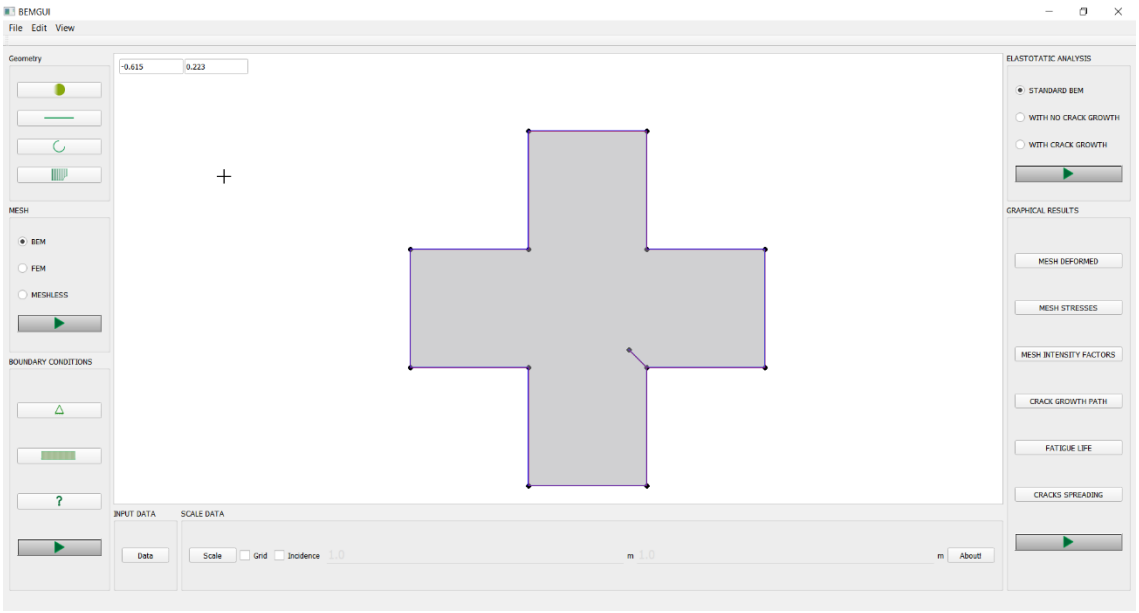

Figura 7.10 – Modelo geométrico de placa cruciforme trincada. (Fonte: Autor).

Após a definição do modelo geométrico, é construída a malha do modelo da mesma maneira que no exemplo anterior, com especial atenção para a trinca, que, por ser um elemento descontínuo, foi discretizado como mostrado pela caixa de diálogo da figura 7.11. Nesta, o usuário pode definir as proporções de cada elemento pelo seu comprimento total. A malha pode ser observada na figura 7.12.

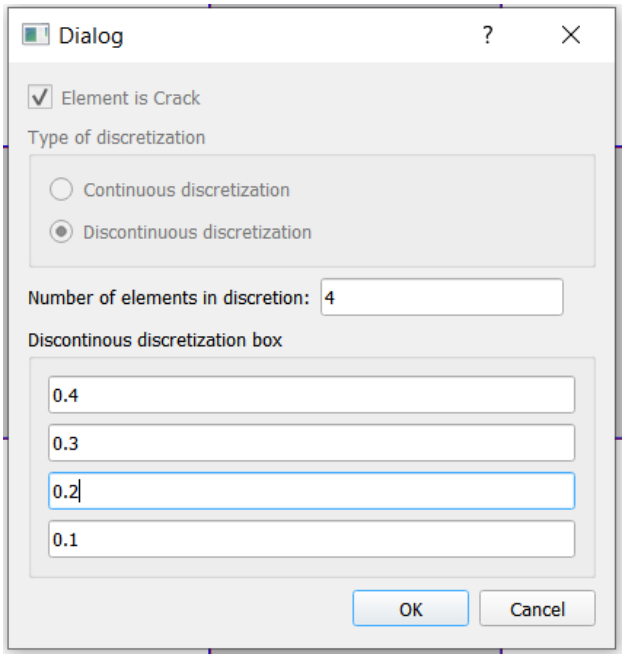

Figura 7.11 – Caixa de diálogo de discretização de trinca (elemento descontínuo). (Fonte: Autor).

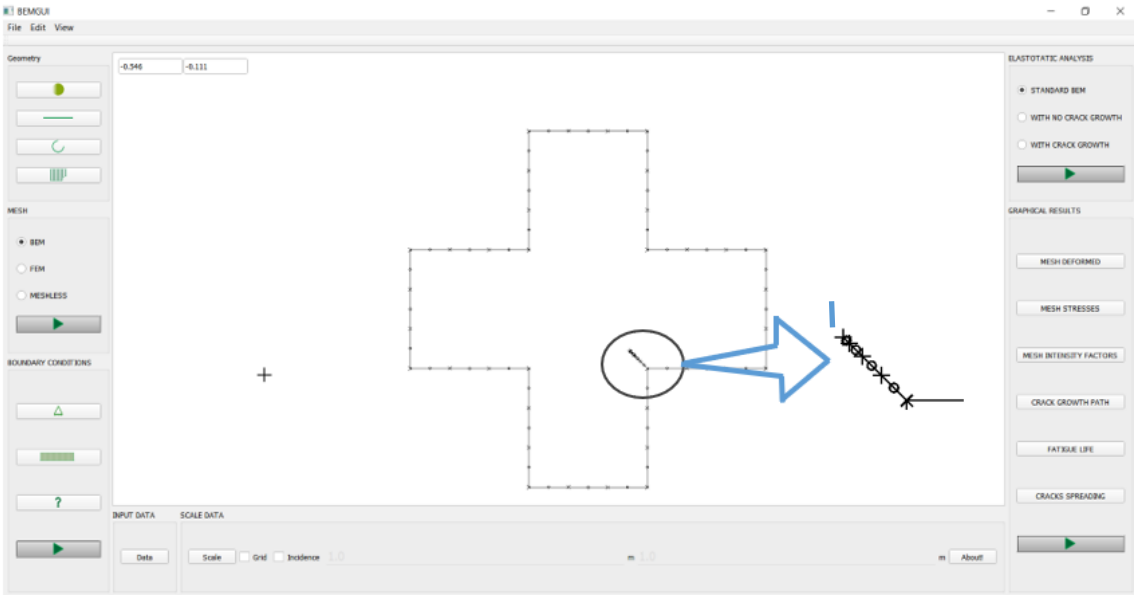

Figura 7.12 – Malha para placa cruciforme trincada com destaque para trinca. (Fonte: Autor).

Com a geração da malha, acrescenta-se as condições de contorno desejadas, e o arquivo de saída do problema pode ser criado. O modelo final está representado na figura 7.13.

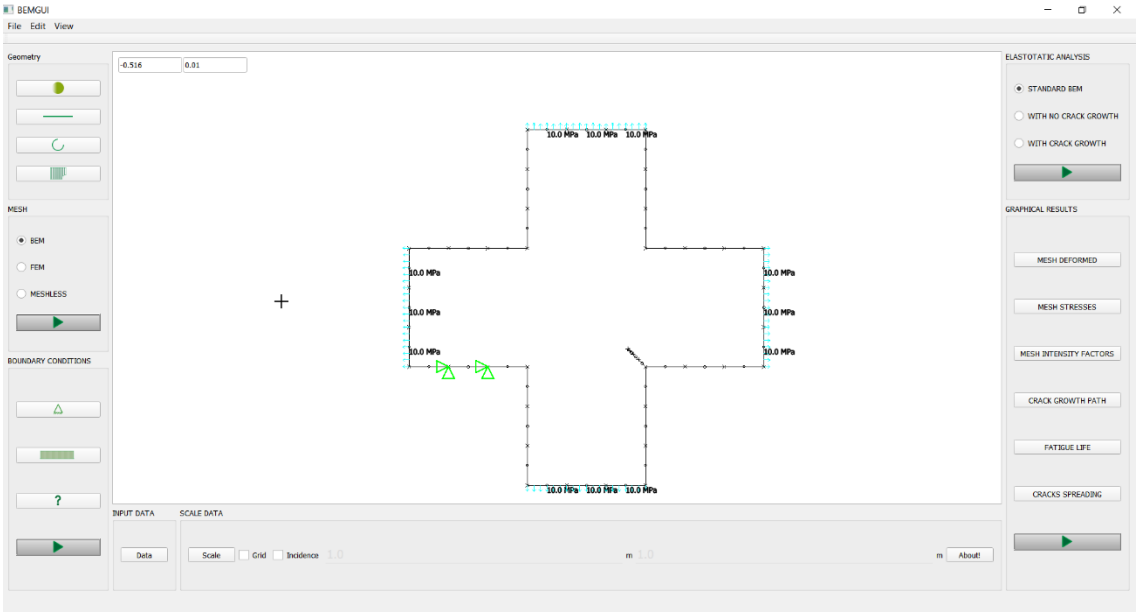

Figura 7.13 – Modelo final para placa cruciforme com trinca inclinada. (Fonte: Autor).

7.3 Placa retangular com trinca e furo

Nesse último exemplo será demostrada a discretização de uma viga com trinca e um furo no centro, como mostrada na figura 7.14 sem tratar-se do uso de condições de contorno.

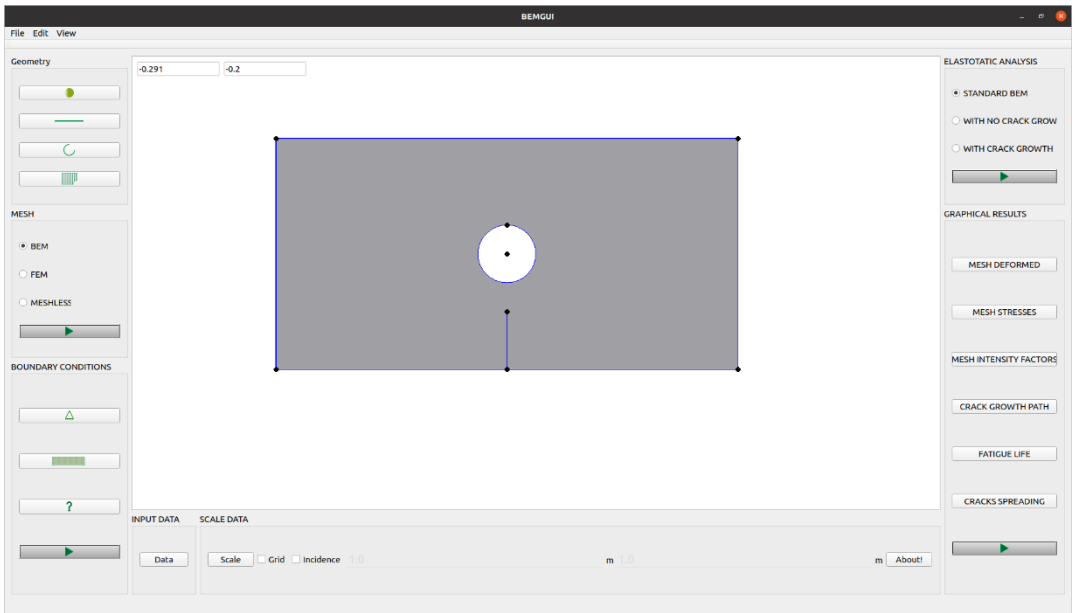

Figura 7.14 – Modelo geométrico da viga trincada com furo dentro do ambiente do BEMGUI. (Fonte: Autor).

Com o modelo geométrico desenhado, foi seguido o mesmo passo observado nos exemplos anteriores, que constrói a malha do modelo. Para tal, foram escolhidos 6 elementos contínuos para o segmento reto superior, 3 para cada um dos segmentos inferiores e 4 para os laterais. Além disso, o furo foi discretizado em 10 elementos contínuos e a malha em 3 elementos

descontínuos, com razões de 0,5, 0,3 e 0,2. O resultado dessa malha pode ser observado na figura 7.15.

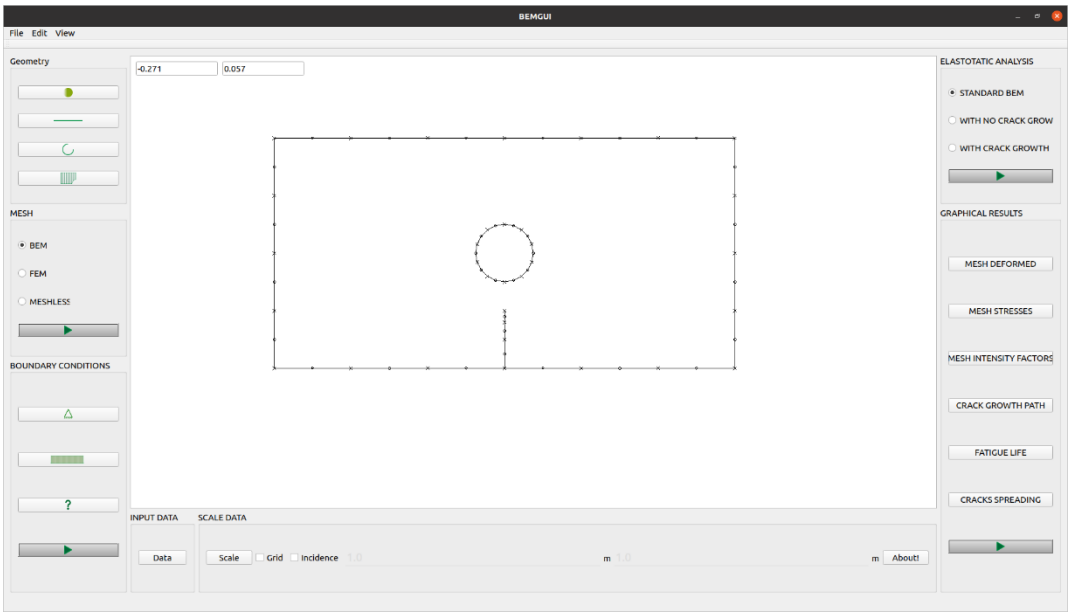

Figura 7.15 – Malha gerada para viga trincada com furo. (Fonte: Autor.)

8. CONSIDERAÇÕES E RECOMENDAÇÕES PARA PESQUISAS FUTURAS

8.1 Conclusões Gerais

Com as estruturas de dados descritas na revisão bibliográfica (grafos e DCEL), foi possível implementar classes que definem de forma consistente a topologia de um modelo de faces bidimensionais. As classes também apresentam interfaces bem definidas, que permitem um comportamento dinâmico ao longo do processo de modelagem, atualizando a topologia do desenho e possibilitando a verificação da validade de ações do usuário.

Essas funcionalidades são válidas tanto para segmentos retos quanto para arcos e curvas de Bézier definidas por equações quadráticas e cúbicas, o que possibilita um conjunto maior de contornos de zonas. Somando-se isso à ferramenta de histórico de comandos, o que possibilita que erros de modelagem possam ser reparados, e o processo de modelagem se torna mais rápido e flexível.

Além disso, atende o objetivo de possibilitar o desenho de modelos bidimensionais genéricos com a automatização da identificação de faces, as quais ainda são acrescidas de constantes de materiais e discretizadas, formando zonas simplificadas por elementos de contorno e condições físicas reais.

A caracterização dos elementos de contorno que discretizam a borda de uma zona respeitam a proposta de Portela, Aliabadi (1992) para a solução de problemas de trincas e permitem o uso de um dos maiores méritos do MEC, que é a reutilização de elementos de malha quando ocorre mudanças na geometria. Com a possibilidade de adição de características físicas aos modelos, o programa reúne dados suficientes para simulações pelo MEC.

Todas essas ideias estão implementadas em classes, presentes em módulos constituintes de pacotes, que compõem um pacote maior e que, por sua vez, divide o programa em três seções maiores, baseados em um padrão de design bem estruturado e verificado. Essa divisão em diferentes níveis de abstrações permite um entendimento sistêmico do software e um estudo específico de certas funcionalidades até o nível de detalhamento que se deseja. Isso tudo foi alcançado pelo uso da POO em diversos níveis de desenvolvimento, o que contribui para a robustez e extensibilidade do programa

Por fim, o programa utiliza o que seus objetos básicos oferecem para gerar arquivos de entrada de processadores e, com o uso de software livre, permite seu estudo e reutilização gratuitos, desde que os trabalhos desenvolvidos a partir dele tenham as mesmas finalidades acadêmicas.

8.2 Sugestões para Pesquisas Futuras

Segundo Sommerville (2011), *software* é um produto que sofre mudanças constantes ao longo de seu ciclo de vida. Essa mudança pode vir da realização da existência de comportamentos inesperados do programa, alteração de comportamentos de seus casos de uso ou da mudança das necessidades dos agentes que usam o *software*. Destaca-se aqui esse último caso, visto que o programa BEMCRACKER2D está em constante evolução. Caso este passe a tratar, por exemplo, zonas múltiplas e/ou zonas definidas por mais de um material, a interface deve conceder ao usuário a possibilidade de criar modelos assim.

Como as classes implementadas no sub-pacote *geometry* do pacote *model* já são capazes de lidar com esses casos, basta alterar comportamentos do pacote *controller*. Para o controle de ações e descrição correta da topologia ao longo da criação do modelo, indica-se como base a seção 4.1 deste trabalho, a qual descreve o funcionamento das interfaces das classes *point*, *edge* e *zone*.

Esse trabalho, no entanto, exige que novas funcionalidades sejam implementadas seguindo a lógica do programa. Em outras palavras, cada ação deve ser guardada em históricos de ações e ela pode ser desfeita e refeita. Além disso, a interface do programa com o processador numérico deve ser alterada, passando dados da maneira que o processador numérico os requer. Isso também é válido para a interação do programa com outros processadores, que podem, por exemplo, usar memória dinâmica ao invés de interagir por meio de arquivos. Em *Python*, isso é possível através de ligações chamadas *python bidings*, que definem tipos de dados que se comunicam com as linguagens C e C++. Para outras, existem outras opções.

A interface possui apenas uma escala pré-definida (em que cada unidade de tela representa 1 cm no mundo real). Isso delimita bastante a dimensão dos problemas a serem modelados. Como pode ser visto na figura 6.1, *widgets* interativos que definem a escala do modelo em seus dois eixos já existem no pacote *view*, mas eles não estão conectados a nenhuma funcionalidade do *controller*. Dessa forma, aprimorar o sistema para que seja possível trabalhar com dimensões do mundo real, a partir da definição de múltiplas escalas pelo usuário, definitivamente aumentaria o alcance do programa.

Além disso, programas de desenhos assistidos por computador são parte essencial da vida de um engenheiro e facilitam a criação e visualização de modelos inseridos em contextos maiores. Portanto, a importação e exportação desses como arquivos lidos por programas CAD se fazem relevantes para uma integração às atividades diárias da Engenharia, indo além da análise de problemas numéricos.

Por fim, a interface gráfica pode ser aproveitada para gerar representações gráficas dos resultados analíticos calculados por programas que seguem o MEC. Com isso, pode ser formado um pacote completo de pré-processamento, processamento e pós-processamento em um ambiente acadêmico.

REFERÊNCIAS BIBLIOGRÁFICAS

ALIABADI, Mohammad H. **The boundary element method, applications in solids and structures**. John Wiley & Sons, v.2, p 3., 2002

BORGES, Luiz Eduardo. **Python para desenvolvedores**: aborda Python 3.3. Novatec Editora, 2014.

BREBBIA, C. A.; DOMINGUEZ, J. **Boundary Elements. Computational Mechanics** Publ., 1989.

BU-QING, Su; DING-YUAN, Liu. **Computational geometry: curve and surface modeling**. Elsevier, 2014.

COSTABEL, Martin. **Principles of boundary element methods**. Techn. Hochsch., Fachbereich Mathematik, 1986.

DASGUPTA, Sanjoy; PAPADIMITRIOU, Christos H.; VAZIRANI, Umesh Virkumar. **Algorithms**. New York: McGraw-Hill Higher Education, 2008.

DATHAN, Brahma; RAMNATH, Sarnath. **Object-Oriented Analysis, Design and Implementation**. Springer International Publishing, Cham, Switzerland, 2015.

DE BERG, Mark; CHEONG, Otfried; Van KREVALD, Marc; OVERMARS, Mark. **Computational geometry algorithms and applications**. 2008.

DELGADO NETO, Álvaro Martins. **BEMLAB2D: interface gráfica de modelagem visualização e análise com elementos de contorno: uma aplicação e problemas elastostáticos**. 2017. xviii, 94 f., il. Dissertação (Mestrado em Estruturas e Construção Civil) – Universidade de Brasília, Brasília, 2017.

GOMES, G. **Estrutura de Dados para Representação de Modelos Bidimensionais de Elementos de Contorno**. Universidade de Brasília. [S.l.], p. 95. 2000. (Publicação 004A/2000).

GOMES, Gilberto; DELGADO NETO, Álvaro M.; WROBEL, Luis C. **Modelagem e visualização de trincas 2D usando equação integral de contorno Dual**. Revista Interdisciplinar De Pesquisa Em Engenharia, v. 2, n. 6, p. 120-133, 2017.

GOMES, Gilberto; MIRANDA, Antonio CO. **Analysis of crack growth problems using the object-oriented program bemcracker2D**. Frattura ed Integrità Strutturale, v. 12, n. 45, p. 67-85, 2018.

KRASNER, Glenn E; POPE, S. T. **A description of the model-view-controller user interface paradigm in the smalltalk-80 system**. Journal of object oriented programming, v. 1, n. 3, p. 26-49, 1988.

LASSER, Dieter. **Calculating the self-intersections of Bezier curves**. Computers in Industry, v. 12, n. 3, p. 259-268, 1989.

Lucid Software Inc., **Lucidchart**, 2008. Disponível em <https://www.lucidchart.com/pages/>. Acesso em: 15/11/2020.

LUTZ, Mark. **Programming python**. " O'Reilly Media, Inc.", 2001.

OLIPHANT, Travis E. **Python for scientific computing**. Computing in Science & Engineering, v. 9, n. 3, p. 10-20, 2007.

PORTELA, A, ALIABADI, MH, Rooke, DP., 1992. **The dual boundary element method: Effective implementation for crack problems**. International Journal of Numerical Methods in Engineering, 33, 1269-1287.

Python Software Foundation. **Python**, 1995 disponível em <https://www.python.org/>. Acesso em: 29/07/2019

RIVERBANK COMPUTING. **Riverbank Computing**, 2006. Disponível em: <https://riverbankcomputing.com/>. Acesso em: 01 de fevereiro de 2020.

RUMBAUGH, James; JACOBSON, Ivar; BOOCH, Grady. **The unified modeling language. Reference manual**, 1999.

SAMPAIO, Fábio Ferrentini. **Modelagem dinâmica computacional e o processo de ensino-aprendizagem: algumas questões para reflexão**. Relatório Técnico NCE, n. 3699, 1999.

SEDERBERG, Thomas W.; FAROUKI, Rida T. **Approximation by interval Bézier curves**. IEEE Computer Graphics and Applications, n. 5, p. 87-88, 1992

SKIENA, Steven S. **The algorithm design manual**. Springer Science & Business Media, 2008.

SOMMERVILLE, Ian. **Software engineering 9th Edition**. ISBN-10, v. 137035152, p. 18, 2011.

STEFIK, Mark; BOBROW, Daniel G. **Object-oriented programming: Themes and variations**. AI magazine, v. 6, n. 4, p. 40-40, 1985.

TORVALDS, Linus; HAMANO, Junio. **Git**, 2005. Disponível em: <https://git-scm.com/>. Acesso em: 06/06/2020.

The Qt Company, **Qt**, 1994. Disponível em <https://www.qt.io/company>. Acesso em: 29/07/2019.

UNHELKAR, Bhuvan. **Software engineering with uml**. CRC Press, 2017.

APÊNDICE A

Problema 1 -> Nome do arquivo
Stress -> Caso da análise
69000000.0 0.30 -> Módulo de Young e coeficiente de Poisson
32 16 10 5.00 10 -> Número de nós, número de elementos, e dimensões máximas
Nodal_Coordinates_(NODE,X,Y)
1 -0.099 -0.038
2 -0.124 -0.057
3 -0.138 -0.076
4 -0.141 -0.093
5 -0.133 -0.11
6 -0.114 -0.125
7 -0.084 -0.14
8 -0.054 -0.146
9 -0.014 -0.136
10 0.034 -0.115
11 0.086 -0.084
12 0.139 -0.049
13 0.19 -0.01
14 0.236 0.027
15 0.273 0.061
16 0.298 0.087
17 0.308 0.103
18 0.305 0.114
19 0.3 0.122
20 0.292 0.128
21 0.283 0.132
22 0.271 0.133
23 0.257 0.132
24 0.237 0.129
25 0.214 0.117
26 0.186 0.1
27 0.155 0.078
28 0.12 0.053
29 0.082 0.028
30 0.041 0.005
31 -0.003 -0.015

32 -0.05 -0.03

Mesh_Topology_(ELEMENT,G-NODE1,G-NODE2,G-NODE3)

1 1 2 3

2 3 4 5

3 5 6 7

4 7 8 9

5 9 10 11

6 11 12 13

7 13 14 15

8 15 16 17

9 17 18 19

10 19 20 21

11 21 22 23

12 23 24 25

13 25 26 27

14 27 28 29

15 29 30 31

16 31 32 1

Displacement_Boundary_Conditions_(ELEMENT,L-NODE,G-NODE)

0 1 0

1 1 1 0

Traction_Boundary_Conditions_(ELEMENT,L-NODE,G-NODE

1 0 0

5 3 30000000

Constrained_Unknowns_(ELEMENT,L-NODE,G-NODE)

0 0 0

Crack_Propagattion_(Number_OF_Crack-Extension_Increments)

0 0 0