UNIVERSIDADE ESTADUAL PAULISTA
"JÚLIO DE MESQUITA FILHO"
Câmpus de Ilha Solteira

FELIPE PERRONI CHAIM

DESENVOLVIMENTO DE UM MÓDULO PERIDINÂMICO BIDIMENSIONAL EM *PYTHON*

ILHA SOLTEIRA
2022

FELIPE PERRONI CHAIM

DESENVOLVIMENTO DE UM MÓDULO PERIDINÂMICO BIDIMENSIONAL EM *PYTHON*

Trabalho de Graduação apresentado à Faculdade de Engenharia de Ilha Solteira – UNESP como parte dos requisitos para obtenção do título de Bacharel em Engenharia Mecânica.

Universidade Estadual Paulista “Júlio de Mesquita Filho”

Faculdade de Engenharia

Câmpus de Ilha Solteira

Orientador: Márcio Antonio Bazani

Ilha Solteira

2022

FICHA CATALOGRÁFICA
Desenvolvido pelo Serviço Técnico de Biblioteca e Documentação

C434d

Chaim, Felipe Perroni.
Desenvolvimento de um módulo peridinâmico bidimensional em python / Felipe Perroni Chaim. -- Ilha Solteira: [s.n.], 2022

83 f. : il.

Trabalho de conclusão de curso (Graduação em Engenharia Mecânica) - Universidade Estadual Paulista. Faculdade de Engenharia de Ilha Solteira, 2022

Orientador: Márcio Antonio Bazani
Inclui bibliografia

1. Peridinâmica. 2. Código. 3. Python. 4. Algoritmo. 5. Vetorização.

Raiane da Silva Santos
Supervisora Técnica de Seção
Seção Técnica de Referência, Atendimento ao usuário e Documentação
Diretoria Técnica de Biblioteca e Documentação
CRB/8 - 9999

UNIVERSIDADE ESTADUAL PAULISTA
"JÚLIO DE MESQUITA FILHO"
Câmpus de Ilha Solteira

CURSO DE ENGENHARIA MECÂNICA
ATA DA DEFESA – TRABALHO DE GRADUAÇÃO

TÍTULO: DESENVOLVIMENTO DE UM MÓDULO PERIDINÂMICO BIDIMENSIONAL EM *PYTHON*

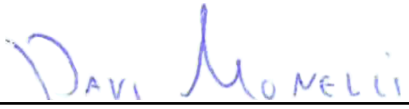
ALUNO: FELIPE PERRONI CHAIM RA: 162054921

ORIENTADOR: MÁRCIO ANTONIO BAZANI

Aprovado (X) - Reprovado ( ) pela Comissão Examinadora

Comissão Examinadora:

Márcio Antonio Bazani
Prof. Dr. e Orientador – UNESP



Davi Moraes Monelli
Engenheiro Mecânico – UNESP

Lucas Perroni Chaim
Engenheiro Mecânico Me. – UNESP

Felipe Perroni Chaim
Aluno

Ilha Solteira (SP), 15 de dezembro de 2022.

DEDICATÓRIA

Este trabalho é dedicado à minha família que sempre me apoiou incondicionalmente.

AGRADECIMENTOS

Os agradecimentos vão para a minha família, por seu apoio incondicional, ao meu professor e orientador Dr. Márcio Antonio Bazani por todo o conhecimento transmitido, à faculdade de forma geral e a todos os meus amigos que, de alguma forma, acabaram me ajudando.

“Que todos os nossos esforços estejam
sempre focados no desafio à impossibilidade.
Todas as grandes conquistas humanas vieram
daquilo que parecia impossível”
(Charles Chaplin)

RESUMO

Este trabalho busca trazer um código capaz de resolver alguns problemas peridinâmicos em duas dimensões desenvolvido originalmente por Patriota (2019) em *MATLAB* para o *Python*, com algumas melhorias, com destaque para a vetorização de alguns algoritmos de áreas parciais, mais liberdade na escolha das condições de contorno, geração de malha e visualização dos resultados. Além disso, o código fica disponível de forma gratuita e pode ser executado em diferentes sistemas operacionais.

A metodologia utilizada foi basicamente a engenharia reversa do código original, realizada linha a linha, com o objetivo de entender o código e poder replicá-lo. Posteriormente foram feitas modificações com auxílio da literatura e com alguns truques de programação.

Por fim, foi feita uma tentativa de efetivação do código, colocando condições de contorno de um problema existente, rodando o programa e realizando a comparação dos resultados.

Palavras-chave: peridinâmica, código, *python*, algoritmo, vetorização.

ABSTRACT

This work focus on bringing a peridynamic code capable of solving some two dimensional problems, focusing a MATLAB code developed by Patriota (2019) to Python, while also adding improvements, such as bringing vectorization technique for the partial area algorithms, more freedom when choosing the boundary conditions, mesh generation and results visualization. Also, the code will be free to download and use and can be used on different operational systems.

The methodology used was basically reverse engineering of the original code, done on a per line basis, trying to figure it out and replicate it. Then, modifications where made with the help from the theorical background and some programming tricks.

Finally, there was an attempt of code efectivation by placing boundary conditions of an existing problem, running the code and comparing the results.

Keywords: peridynamics, code, python, algorithm, vectorization.

LISTA DE ILUSTRAÇÕES

LISTA DE QUADROS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

DTT	*Du*, *Tao* e *Tian*
FA	*Full Area*
IPA-AC	*Improved Partial Area Analytical Calculation*
LBB	*Linearized bond-based model*
LPS	Linear LeridynamicSolid
PA	*Partial Area*
PA-AC	*Partial Area Analytical Calculation*
PA-HHB	*Partial Area Hu, Ha, Bobaru*
PMB	*Prototype Brittle Model*

LISTA DE SÍMBOLOS

$c(\xi)$	função do micro-módulo de elasticidade
d	razão de malha
$\underline{f}\langle\xi\rangle$	force scalar state
h	espaçamento da malha: distância entre os pontos x_i
δ	horizonte peridinâmico
ΔV_{ij}	volume do nó j na vizinhança δ do nó i
E	módulo de *Young*
f_{ij}	força de interação entre os nós de índice i e $j \in \mathcal{F}_i$
$\mathcal{F}_i$	família dos índices relacionados ao nó *i*
G_0	taxa de liberação de energia
m	volume pesado
μ	fator de dano
S_c	estado crítico de elongação relativa
$\underline{T}$	vetor do estado de dano
τ_i	célula quadrada de lados h centrada no ponto x_i
$\tau_j^{(i)}$	célula quadrada de lados h centrada no ponto $x_j^{(i)}$
$\theta(x)$	dilatação
u	vetor contendo o deslocamento dos pontos x_i
ν	coeficiente de *Poisson*
x_i	ponto i, localizado no centro da célula τ_i
$x_j^{(i)}$	pontos j, vizinhos do ponto x_i
$\omega(\lvert\xi\rvert)$	função influência
ξ	vetor de ligação

LISTA DE ALGORITMOS

SUMÁRIO

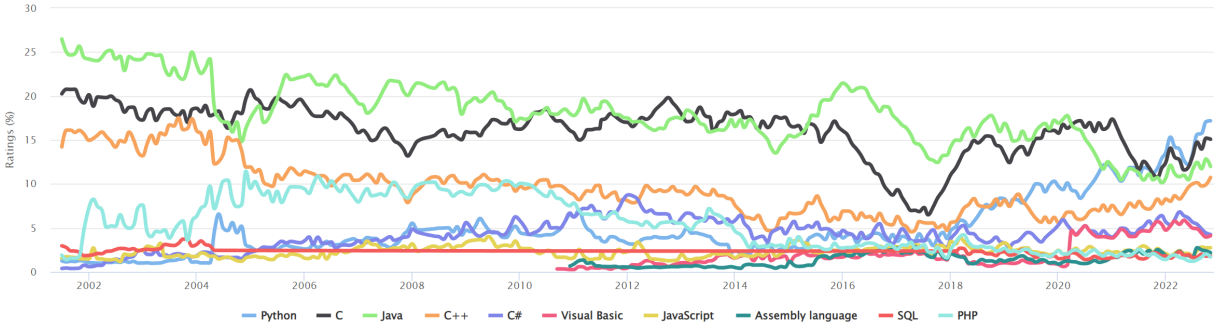
1 INTRODUÇÃO

1.1 Objetivos e motivações

O desenvolvimento de um programa capaz de resolver problemas peridinâmicos em duas dimensões em *Python* baseado em um programa desenvolvido em *MATLAB* por Patriota (2019) se deve a alguns fatores que serão abordados a seguir.

A linguagem de programação *Python* é gratuita e pode rodar em vários sistemas operacionais, como *Windows*, *macOS* e *Linux*. Além disso, o *Python* é a linguagem de programação mais relevante dos últimos anos (TIOBE, 2022).

Figura 1.1 – Índice de popularidade das linguagens de programação nos últimos anos



Fonte: (TIOBE, 2022).
Nota: *Python* cresce significativamente a partir de 2018.

Figura 1.2 – Tabela contendo 20 linguagens de programação com os melhores índices de novembro de 2022

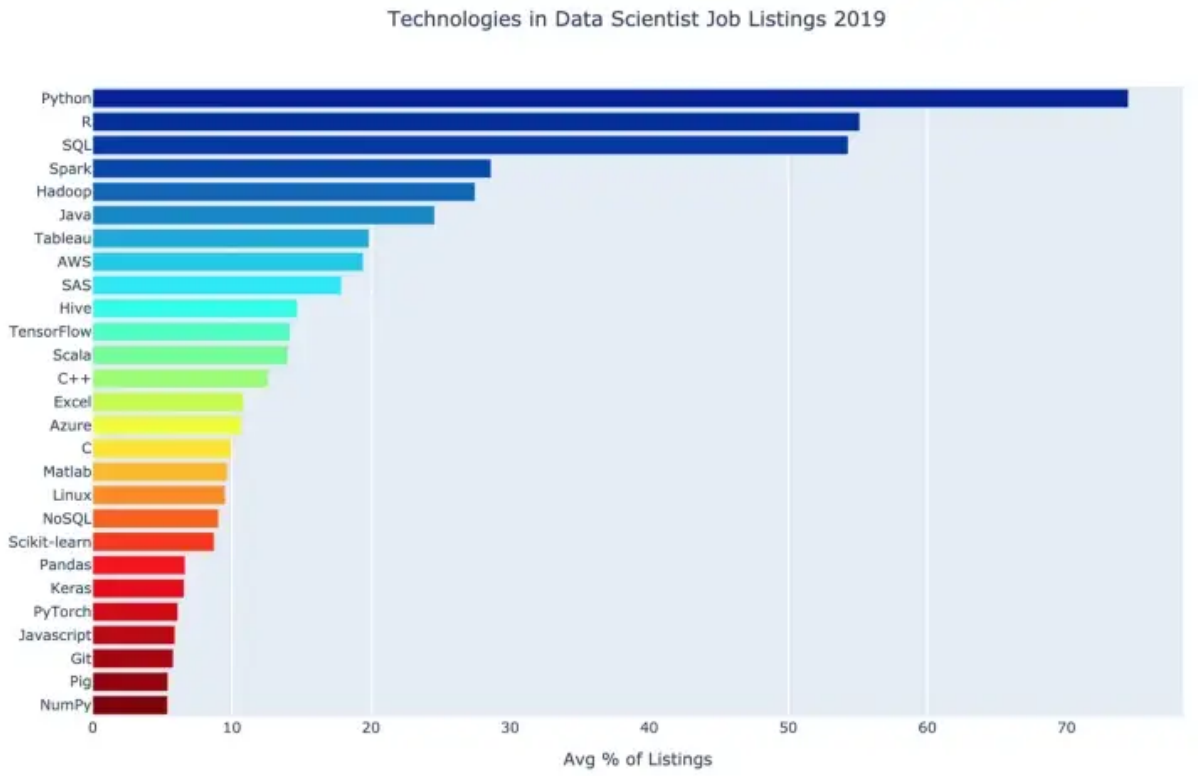
Nov 2022	Nov 2021	Change	Programming Language	Ratings	Change
1	1		Python	17.18%	+5.41%
2	2		C	15.08%	+4.35%
3	3		Java	11.98%	+1.26%
4	4		C++	10.75%	+2.46%
5	5		C#	4.25%	-1.81%
6	6		Visual Basic	4.11%	-1.61%
7	7		JavaScript	2.74%	+0.08%
8	8		Assembly language	2.18%	-0.34%
9	9		SQL	1.82%	-0.30%
10	10		PHP	1.69%	-0.12%
11	18	⇈	Go	1.15%	-0.06%
12	15	↑	R	1.14%	-0.14%
13	11	↓	Classic Visual Basic	1.10%	-0.46%
14	17	↑	Delphi/Object Pascal	1.08%	-0.14%
15	20	⇈	MATLAB	1.02%	-0.15%
16	28	⇈	Objective-C	0.93%	+0.39%
17	24	⇈	Scratch	0.88%	+0.11%
18	14	⇊	Swift	0.87%	-0.56%
19	13	⇊	Ruby	0.85%	-0.59%
20	29	⇈	Rust	0.75%	+0.21%

Fonte: (TIOBE, 2022).

O *MATLAB*, por outro lado, é um software pago e limitado a funcionar somente no sistema opecional *Windows* e não está com uma relevância significativa nestes últimos anos.

Outra motivação se dá ao fato que o *Python* é muito popular entre cientistas de dados. tds (2019) aponta em uma pesquisa realizada em sites de emprego entre 2018 e 2019 que as aplicações para empregos relacionados à área de ciência de dados utilizando *Python* esteve em alta, liderando o ranking de linguagens para esta área, conforme mostra a figura 1.3.

Figura 1.3 – Porcentagem de aplicação de linguagens de programação em empregos relacionados à ciência de dados



Fonte: (TDS, 2019).

Além disso, todo programa tem limitações. Nesta nova iteração, foram feitas algumas melhorias, trazendo algumas funcionalidade diferentes e melhorias de desempenho, quando possível.

Em termos da peridinâmica em si, podemos dizer que ela é uma excelente ferramenta de prognóstico.

É de extrema importância em ambientes industriais a manutenção de equipamentos. Uma estimativa chamada *Remaining Useful Life* (*RUL*), é de interesse particular na área de engenharia voltada para prognósticos. A estimativa *RUL* busca, basicamente, monitorar a capacidade de componentes e sistemas de resistir à carregamentos estruturais, térmicos e químicos (MORAES VINHA, 2022).

Um modelo de *Finite Elements Method* (*FEM*) é limitado por não conseguir lidar com corpos que sofrem dano ou quando há propagação de trincas, devido a sua hipótese de que o corpo sempre permanece contínuo após a deformação (MORAES VINHA, 2022).

Uma das grandes vantagens da peridinâmica é justamente permitir modelar corpos que sofrem dano ou nos quais há propagação de trincas, podendo ser utilizado em casos de prognósticos e abrindo um grande leque de aplicações diferentes.

1.2 A Peridinâmica

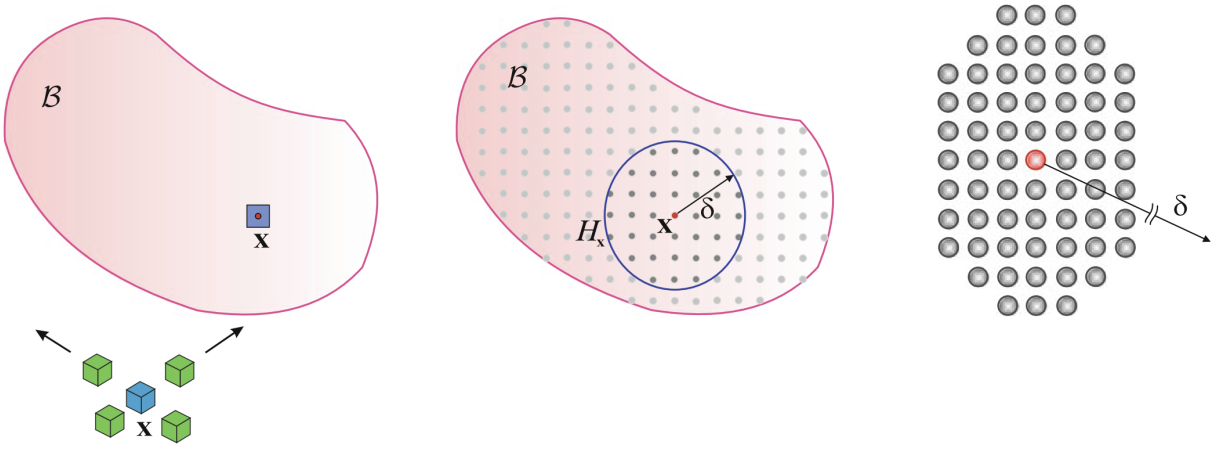
A Peridinâmica é uma teoria dentro da área da mecânica dos sólidos, sendo uma extensão da mecânica do contínuo clássica. É formulada a partir da integração das interações das forças entre partículas separadas por distâncias finitas (PATRIOTA, 2019).

Essa integração é dita espacial e é capaz de determinar a força interna total que age sobre um material infinitesimal em um certo tempo. Feita esta integração, utiliza-se um processo de integração temporal para rastrear as posições das partículas do material ao longo do tempo devido à aplicação de cargas (MADENCI; OTERKUS, 2014).

A ideia principal da Peridinâmica é oferecer a possibilidade de modelar partículas discretas como se fossem um sólido contínuo. Desta forma, tornam-se compatíveis interações entre partículas discretas e sólidos contínuos (BOBARU et al., 2016).

A mecânica molecular é quem permite a modelagem peridinâmica a partir da mecânica do contínuo. A Figura 1.4 ilustra a diferença entre um modelo de um sólido contínuo (*Local*) e um modelo peridinâmico (*Peridynamics*), bem como a dinâmica molecular (*Molecular dynamics*) (MADENCI; OTERKUS, 2014).

Figura 1.4 – Comparação entre diferentes modelos



Fonte: (MADENCI; OTERKUS, 2014).

A Figura 1.4 também mostra o conceito de família e horizonte. Todos as partículas dentro da circunferência H_x são consideradas a família de x. O raio da circunferência é dado pelo parâmetro δ, que é conhecido como horizonte ou comprimento interno. A interação peridinâmica da partícula x ocorre dentro da distância δ, tornando-se mais local conforme o horizonte diminui. Assim, o horizonte é considerado o caso limite entre a peridinâmica e a mecânica clássica do contínuo (MADENCI; OTERKUS, 2014).

As interações peridinâmicas são dadas pela equação de movimento mostrada na Equação 1.1 (PATRIOTA, 2019).

$$\rho\ddot{\mathbf{u}}(\mathbf{x},t) = \int_{H_x} \mathbf{f}\left(\mathbf{x},\mathbf{x}',\mathbf{u},t\right) dV_{\mathbf{x}'} + \mathbf{b}(\mathbf{x},t) \tag{1.1}$$

onde $\ddot{\mathbf{u}}(\mathbf{x},t)$ é a aceleração do ponto x no momento t, ρ é a densidade, f é uma função da interação de forças entre a ligação peridinâmica entre os pontos x e x', cuja unidade é dada em força por volume ao quadrado e b é a densidade de forças externas atuantes no corpo (PATRIOTA, 2019).

A interação entre dois pontos materiais x e x' é uma função de deslocamento entre estes pontos. Como estes pontos formam uma ligação (dada por um vetor que conectas os dois pontos), o modelo depende depende destas ligações (*Bond-based peridynamics*). Para esta modelagem, considera-se o material como sendo micro elástico, onde as interações de força assumem a forma mostrada na Equação 1.2 (PATRIOTA, 2019).

$$\mathbf{f}\left(\mathbf{x},\mathbf{x}',\mathbf{u}\right) \equiv \hat{\mathbf{f}}(\boldsymbol{\xi},\boldsymbol{\eta}) = \underline{f}\langle\boldsymbol{\xi}\rangle\underline{\mathbf{e}}\langle\boldsymbol{\xi}+\boldsymbol{\eta}\rangle \tag{1.2}$$

onde $\boldsymbol{\xi} = \mathbf{X}' - \mathbf{X}$ é a ligação não deformada, $\boldsymbol{\eta} = \underline{\mathbf{u}}\langle\boldsymbol{\xi}\rangle$ é o vetor de deslocamento da ligação e $\underline{\mathbf{e}}\langle\boldsymbol{\xi}+\boldsymbol{\eta}\rangle = \frac{\boldsymbol{\xi}+\boldsymbol{\eta}}{|\boldsymbol{\xi}+\boldsymbol{\eta}|}$ é o vetor unitário na direção da ligação deformada (PATRIOTA, 2019).

O termo $\hat{\mathbf{f}}(\boldsymbol{\xi},\boldsymbol{\eta})$ é derivado da energia elástica $w(\boldsymbol{\xi},\boldsymbol{\eta})$ e é dada conforme a Equação 1.3 (PATRIOTA, 2019).

$$\hat{\mathbf{f}}(\boldsymbol{\xi},\boldsymbol{\eta}) = \frac{\partial w}{\partial \boldsymbol{\eta}}(\boldsymbol{\xi},\boldsymbol{\eta}) \tag{1.3}$$

A energia elástica total é dada pela soma de todas as micro energias elásticas, de acordo com a Equação 1.4 (PATRIOTA, 2019).

$$W(\mathbf{x}) = \frac{1}{2}\int_{H_x} w(\boldsymbol{\xi},\boldsymbol{\eta}) dV_{\boldsymbol{\xi}} \tag{1.4}$$

Utilizando o modelo conhecido como *Prototype micro-brittle (PMB)*, temos que o vetor de força é dado pela Equação 1.5 (PATRIOTA, 2019).

$$\underline{f}\langle\boldsymbol{\xi}\rangle = c(\boldsymbol{\xi})\underline{s}\langle\boldsymbol{\xi}\rangle \tag{1.5}$$

onde $\underline{s}\langle\boldsymbol{\xi}\rangle$ é o escalar de alongamento relativo e $c(\boldsymbol{\xi})$ é a função de micro-módulo de elasticidade.

Desenvolvendo as Equações (1.3) e (1.5), chega-se que a energia de deformação em um ponto, para o modelo *PMB* é dado conforme a a Equação 1.6 (PATRIOTA, 2019).

$$W_{PMB}(\mathbf{x}) = \frac{1}{2}\int_{H_x} \frac{1}{2}\underline{f}\langle\boldsymbol{\xi}\rangle\underline{e}\langle\boldsymbol{\xi}\rangle dV_{\mathbf{x}'} = \frac{1}{4}\int_{H_x} c(\boldsymbol{\xi})\underline{s}\langle\boldsymbol{\xi}\rangle^2|\boldsymbol{\xi}|dV_{\mathbf{x}'} \tag{1.6}$$

A função de micro-módulo de elasticidade pode ser dada pela Equação 1.7 (PATRIOTA, 2019).

$$c(\boldsymbol{\xi}) = Cj(|\boldsymbol{\xi}|). \tag{1.7}$$

onde C é uma constante e $j(\cdot) : \mathbb{R} \to \mathbb{R}$ é um escalar de valor positivo no comprimento da ligação, que implica anisotropia do material. De acordo com o modelo *PMB*, o micro-módulo de elasticidade é constante, portanto, $j(|\boldsymbol{\xi}|) = 1$. Assumindo que a energia de deformação é igual a energia total de deformação peridinâmica em uma deformação homogênea. Com isso, é possível relacionar o micro-módulo de elasticidade com as contantes da elasticidade, conforme a Equação 1.8 (PATRIOTA, 2019).

$$C = \begin{cases} \frac{6E}{\pi\delta^4(1-2\nu)} & \text{(3D)} \\ \frac{12E}{\pi t\delta^3(1+\nu)} & \text{(2D)} \end{cases} \tag{1.8}$$

onde E é o módulo de *Young*, ν é o coeficiente de *Poisson* e t é a espessura da placa. É possível provar que no caso tridimensional (3D) e quando há deformação planar em um caso bidimensional (2D), ν é $\frac{1}{4}$. Em casos 2D onde há tensões planares, ν é $\frac{1}{3}$ (PATRIOTA, 2019).

Em um domínio com N_P partículas, o número de interações para cada iteração computacional é $N_F = \frac{N_P\,(N_P-1)}{2}$, sendo aproximadamente $N_F = \frac{{N_P}^2}{2}$ para um número grande de partículas. Portanto, é necessário limitar o número de cálculos de acordo com uma ideia topológica (GERSTLE, 2015).

A abordagem mais comum é decompor o domínio em uma matriz de células cúbicas cujos tamanhos são levemente superiores ao tamanho do horizonte (δ). Usando este conceito, cada partícula só precisa checar por interações entre partículas localizadas na sua própria célula e entre células adjacentes, tornando o número de interações $N_F \approx \frac{N_P N_A N_Q}{2}$, sendo N_Q o número aproximado de partículas por célula e N_A o número de células adjacentes (GERSTLE, 2015).

Segundo Patriota (2019), a função influência $\omega(|\xi|)$ garante que, ao utilizar uma função simétrica e quando toas as ligações possuem o mesmo comprimento, todas as

ligações são tratadas da mesma maneira, garantindo a isotropia do material. A função influência é mostrada na Equação 1.9.

$$\omega(|\boldsymbol{\xi}|) = \frac{\omega_\delta(|\boldsymbol{\xi}|)}{|\boldsymbol{\xi}|^\gamma} \tag{1.9}$$

A escolha da função influência pode afetar as características de dispersão, aderindo peso às interações não locais (BUTT; TIMOTHY; MESCHKE, 2017).

As funções influências consideradas neste trabalho são:

$$\omega(|\boldsymbol{\xi}|) = \begin{cases} \exp\left(\dfrac{-|\boldsymbol{\xi}|^2}{\left(\frac{\delta}{4{,}3}\right)^2}\right) & \text{(Exponencial)} \\ \dfrac{1}{|\boldsymbol{\xi}|^\gamma} & \text{(Constante)} \\ \dfrac{1-\frac{|\boldsymbol{\xi}|}{\delta}}{|\boldsymbol{\xi}|^\gamma} & \text{(Cônica)} \\ \dfrac{1-3\left(\frac{|\boldsymbol{\xi}|}{\delta}\right)^2+2\left(\frac{|\boldsymbol{\xi}|}{\delta}\right)^3}{|\boldsymbol{\xi}|^\gamma} & \text{(Polinomial cúbica)} \\ \dfrac{1-10\left(\frac{|\boldsymbol{\xi}|}{\delta}\right)^3+15\left(\frac{|\boldsymbol{\xi}|}{\delta}\right)^4-6\left(\frac{|\boldsymbol{\xi}|}{\delta}\right)^5}{|\boldsymbol{\xi}|^\gamma} & \text{(P5)} \\ \dfrac{1-35\left(\frac{|\boldsymbol{\xi}|}{\delta}\right)^4+84\left(\frac{|\boldsymbol{\xi}|}{\delta}\right)^5-70\left(\frac{|\boldsymbol{\xi}|}{\delta}\right)^6+20\left(\frac{|\boldsymbol{\xi}|}{\delta}\right)^7}{|\boldsymbol{\xi}|^\gamma} & \text{(P7)} \\ \dfrac{\frac{\delta}{|\boldsymbol{\xi}|}-1}{|\boldsymbol{\xi}|^\gamma} & \text{(Singular)} \end{cases} \tag{1.10}$$

Observando a equação (1.10), temos que a norma $|\boldsymbol{\xi}|$ representa a distância entre um ponto x_i e seus vizinhos $x_j^{(i)}$, sendo assim, este valor nunca será 0, evitando problemas numéricos.

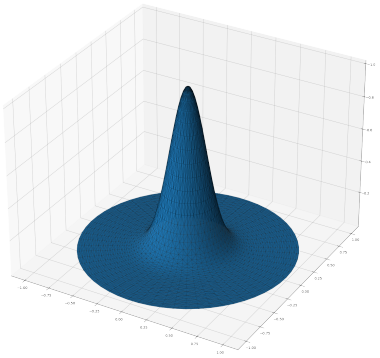
Para visualização, foi discretizado um domínio circular representando um horizonte $\delta = 1$ centrado no ponto $(0+\epsilon,0+\epsilon)$[1] e considerou-se $\gamma = 0$. Aplicando cada caso mostrado equação (1.10), chegou-se na figura 1.5.

Observando a figura 1.5, vemos que a função influência constante implica que as forças entre os pontos materiais dentro da vizinhança de um ponto da malha peridinâmica possuem a mesma importância, ou seja, o mesmo valor. Isso implica em uma alta não-localidade para o modelo. Já a função cônica, por exemplo, suaviza

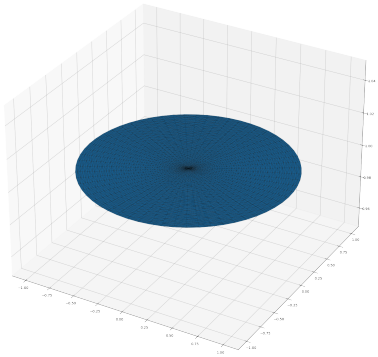
[1] Excepcionalmente, utilizou-se de um valor ϵ justamente para evitar os problemas numéricos citados anteriormente. ϵ é um valor muito pequeno na ordem de precisão da máquina. Isso não acontece durante o uso canônico do programa.

a influência dos vizinhos de acordo com a sua distância, sendo uma característica desejada para atenuar efeitos peridinâmicos de superfície. Portanto, neste trabalho será utilizado a função influência cônica (PATRIOTA, 2019).

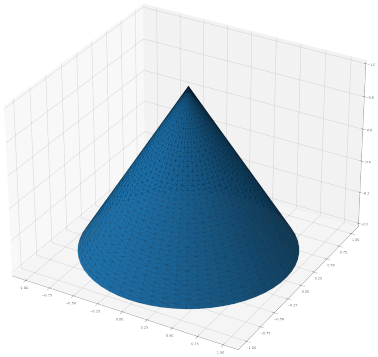
Figura 1.5 – Comparação entre diferentes funções influência



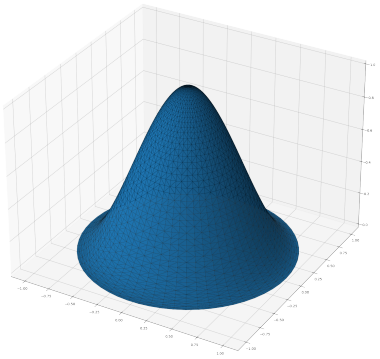
(a) Exponencial



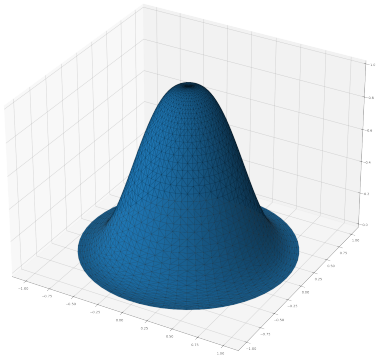
(b) Constante



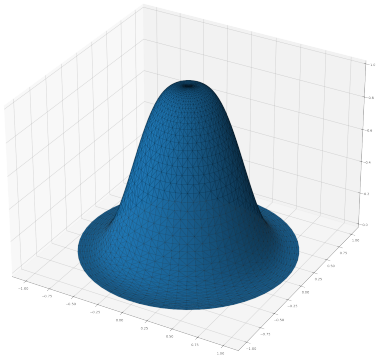
(c) Cônica



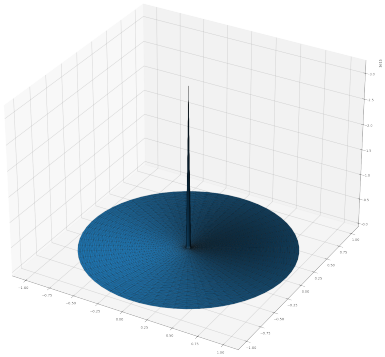
(d) Polinomial cúbica



(e) P5



(f) P7



(g) Singular

Fonte: Elaborado pelo autor.

A equação (1.11) mostra um exemplo de um vetor de estado de força para um modelo 2D de um material *LPS* linearizado (PATRIOTA, 2019).

$$\widetilde{\mathbf{T}}_{2D,PL}\langle\boldsymbol{\xi}\rangle = \left[\frac{\tilde{c}_1}{m}\omega(|\boldsymbol{\xi}|)|\boldsymbol{\xi}|\frac{1}{m}\tilde{\theta}_{2D} + \frac{\tilde{c}_2}{m}\omega(|\boldsymbol{\xi}|)\underline{\mathbf{u}}\langle\boldsymbol{\xi}\rangle\cdot\boldsymbol{\xi}\right]\underline{\mathbf{e}}\langle\boldsymbol{\xi}\rangle \tag{1.11}$$

onde $\tilde{\theta}_{2D} = \frac{2}{m}\int_{\mathcal{H}_\delta}\omega(|\boldsymbol{\zeta}|)\boldsymbol{\zeta}\cdot\underline{\mathbf{u}}\langle\boldsymbol{\zeta}\rangle dV_{\boldsymbol{\zeta}}$ é a dilatação em 2D, $\tilde{c}_1$ e $\tilde{c}_2$ são constantes do material (PATRIOTA, 2019).

Seguindo nesta linha, Patriota (2019) demonstra que a densidade de energia de deformação pode ser expressa pela equação (1.12).

$$W_{LPS}(\mathbf{x}) = \left[\frac{\hat{c}_1}{2} + \frac{\hat{c}_2}{2}\frac{m}{3^2}\left(\frac{-\nu+2}{(2\nu-1)}\right)\right]\hat{\theta}^2 + \frac{\hat{c}_2}{2}\int_{\mathcal{H}_\delta}\omega(|\boldsymbol{\xi}|)\underline{e}\langle\boldsymbol{\xi}\rangle^2 dV_{\boldsymbol{\xi}} \tag{1.12}$$

2 METODOLOGIA

Este trabalho foi baseado grande parte no código e na tese de Patriota (2019). Utilizando o extenso código desenvolvido em *MATLAB* como base, foi feita a conversão para *Python*, trazendo várias dificuldades devido à complexidade do código e particularidades entre as duas diferentes linguagens de programação.

Com o código desenvolvido, tentou-se realizar a sua validação, ao aplicar em um experimento físico, conforme mostrado no capítulo 4.

De forma geral, foram feitas algumas otimizações como:

- Algumas partes do código foram vetorizadas, tornando-o mais eficiente;
- O código está mais modular, conferindo mais liberdade na sua utilização;
- Algumas variáveis que são utilizadas mais de uma vez foram calculadas somente uma vez e armazenadas em um objeto, evitando a necessidade de recalculá-la ou de carregar várias variáveis, simplificando o processo;
- O pós-processamento foi dividido em funções menores, dando mais liberdade na hora da exibição dos resultados gráficos;
- Criação de funções auxiliares para ajudar o usuário a extrair a figura de um objeto e salvá-la no seu computador;
- As funções, de forma geral, foram simplificadas no seu uso, trazendo mais verbalidade na entrada dos argumentos e/ou exigindo menos argumentos de entrada. Neste sentido, em especial a classe *BoundaryConditions* referente às condições de contorno foi totalmente reformulada, permitindo o total controle por parte do usuário de forma mais simples, sendo necessário apenas a definição por argumentos de entrada, em contraste ao código original que exigia a edição de uma função no código fonte;
- A classe responsável pela geração de malha, *Mesh*, trouxe mais flexibilidade ao permitir que o usuário crie malhas de formato arbitrário, não sendo necessária a criação de malhas retangulares, bem como muitas facilidades embutidas como métodos e atributos, como a possibilidade de realizar plotar o seu gráfico com apenas um comando;
- É oferecida uma função para salvar os objetos, facilitando a reutilização de variáveis que levam muito tempo para calcular, como é o caso, por exemplo, da determinação de uma família de uma malha com um alto número de elementos.

3 IMPLEMENTAÇÃO NUMÉRICA

A equação de movimento da peridinâmica é integro-diferencial e, portanto, sua resolução analítica costuma ser complicada. Portanto, utilizam-se técnicas de integrações espaciais e de tempo (MADENCI; OTERKUS, 2014).

A integração espacial é feita utilizando o esquema de malha livre, devido à sua simplicidade. Primeiro, o domínio é dividido em volumes (3D) ou áreas (2D). Então, faz-se a integração espacial envolvendo a somatória de volumes ou áreas dos pontos materiais que estão dentro do horizonte (MADENCI; OTERKUS, 2014).

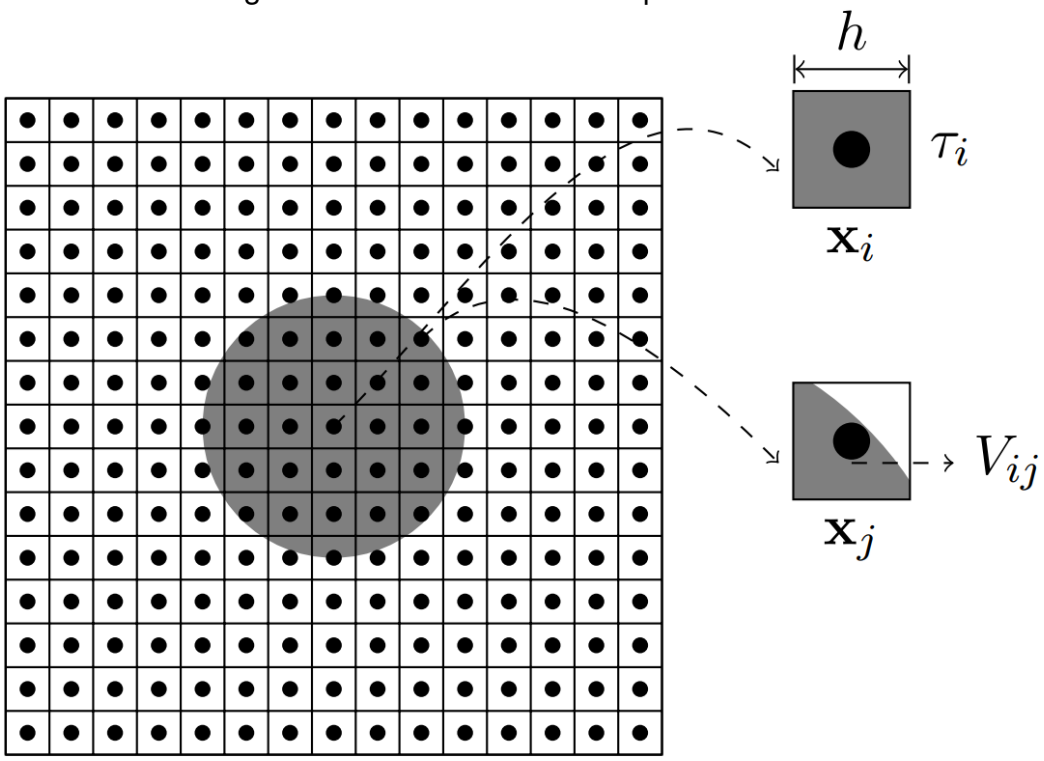
Os volumes ou as áreas podem não estar contidos em sua integridade dentro do horizonte, como por exemplo, pontos localizados na superfície podem ter seus volumes ou áreas truncadas, podendo ser necessário aplicar um fator de correção para corrigir o excesso de volume ou área que pode atrapalhar na integração. A esta técnica dá-se o nome de fator de correção de volume (MADENCI; OTERKUS, 2014).

Neste trabalho, este fator de correção de volume foi obtido e será detalhado na seção 3.3.

3.1 Equação governante

O domínio considerado neste trabalho será bidimensional. Chamamos este domínio de $\mathcal{B}$ e sua forma discretizada de $\mathcal{B}_D$. $\mathcal{B}_D$ aproxima o domínio $\mathcal{B}$ por meio de uma malha homogênea. Cada ponto tem uma posição x_i, é centralizado em uma célula quadrada de área h^2, densidade ρ_i e é espaçado entre o próximo ponto x_i de h unidades de distância (PATRIOTA, 2019).

Figura 3.1 – Domínio discreto peridinâmico



Fonte: (PATRIOTA, 2019).

Nota: O círculo em cinza escuro representa a vizinhança-δ do ponto i, onde δ é o horizonte e representa o raio deste círculo.

A equação (3.1), válida para os pontos $x_i \in \mathcal{B}$ é espacialmente integrada utilizando a regra de quadratura de meio ponto (PATRIOTA, 2019).

$$\rho_i \ddot{\mathbf{u}}_i = \sum_{j \in \mathcal{F}_i} \mathbf{f}_{ij} \Delta V_{ij} + \mathbf{b}_i \tag{3.1}$$

onde $\ddot{\mathbf{u}}_i = \ddot{\mathbf{u}}(x_i)$, $\mathbf{b}_i = \mathbf{b}(x_i)$, $\mathbf{f}_{ij} = \mathbf{f}(x_i, x_j, u)$ é força de interação entre os nós i e $j \in \mathcal{F}_i$, Δ_{ij} é o volume (3D) ou área (2D) do nó j dentro da vizinhança de i e $\mathcal{F}_{ij}$ é a família de índices relacionados ao nó i da seguinte forma:

$$\mathcal{F}_i = \{j | \ |x_j - x_i| < \delta\} \tag{3.2}$$

O horizonte δ é um parâmetro do material e não depende da sua geometria. Na peridinâmica, ele é análogo à interações atômicas em sistemas microscópicos. No entanto, valores muito pequenos podem impactar significativamente no tempo levado para realizar uma simulação. Então, utiliza-se o parâmetro razão de malha, dado conforme a equação (3.3) (PATRIOTA, 2019).

$$d = \frac{\delta}{h} \tag{3.3}$$

Quanto maior é d, maior é o nível de refinamento da malha dentro da vizinhança-δ e, portanto, deve-se escolher um valor sensato que forneça uma malha com um nível pelo menos minimalmente aceitável. Normalmente utiliza-se d entre 4 e 6 para simulação em 2D e d entre 6 e 9 para simulações em 3D (PATRIOTA, 2019).

3.2 Malha

A criação da malha em duas dimensões foi feita baseada nos pontos x_i, que ficam localizados no centro das células quadradas τ_i.

O programa desenvolvido permite a criação de dois tipos de malha: retangular ou não retangular. Vamos focar primeiro na malha retangular, que será o foco do trabalho e depois abordar sobre as dificuldades de trabalhar com uma malha não retangular.

3.2.1 Geração de uma malha retangular

Os parâmetros utilizados para a geração de uma malha retangular são a distância entre os pontos x_i, dada por h, que também representa os lados das células τ_i, a largura, o comprimento, que foram apelidados de $size_x$ e $size_y$ e o modo, que é um parâmetro opcional.

A malha criada é um objeto, permitindo que este objeto carregue com ele variáveis nos seus atributos e métodos que alteram seus atributos ou retornam novas variáveis. Desta forma, todas as variáveis e métodos relevantes à malha, ficam em uma só variável objeto.

A criação da malha funciona da seguinte forma. Primeiramente, o programa calcula o número de linhas e colunas baseadas nas dimensões e na distância h.

$$\begin{cases} cols = int(size_x/h) + 1 \\ rows = int(size_y/h) + 1 \end{cases} \tag{3.4}$$

onde int é uma função que transforma o número em inteiro, considerando apenas a sua parte inteira, semelhante a função $floor$. Por exemplo, $int(3{,}999) = 3$.

Este processo, dependendo da entrada do usuário, pode causar erros de ponto flutuante ao converter a divisão de dois pontos flutuantes para um número inteiro. Para consertar isto, faz-se quatro verificações.

$$\begin{cases} cols = cols - 1 & \text{se size\_x/h for próximo de cols-1} \\ cols = cols + 1 & \text{se size\_x/h for próximo de cols+1} \\ rows = rows - 1 & \text{se size\_y/h for próximo de rows-1} \\ rows = rows + 1 & \text{se size\_x/h for próximo de rows+1} \end{cases} \tag{3.5}$$

Então entra o modo, que determina o comportamento que a malha fará para incluir ou não pontos x_i nas suas extremidades:

- *restrictive*: garante que os pontos x_i não ultrapassem os valores estabelecidos por $size_x$ e $size_y$;
- *expansive*: permite que os pontos x_i sejam iguais aos limites pela geometria ou ultrapassem, até, no máximo uma distância h para acomodar os pontos;
- *exact*: é semelhante ao *restrictive*, porém emite um erro caso os pontos x_i das extremidades não coincidam com os limites geométricos estabelecidos.

Tendo essas informações, o programa gera, utilizando vetorização, um vetor para as coordenadas x das colunas e um vetor y para as coordenadas y das linhas.

$$\begin{cases} x_m = hm \\ y_n = hn \end{cases} \tag{3.6}$$

com m indo de 0 até o número de colunas e n indo de 0 até o número de linhas.

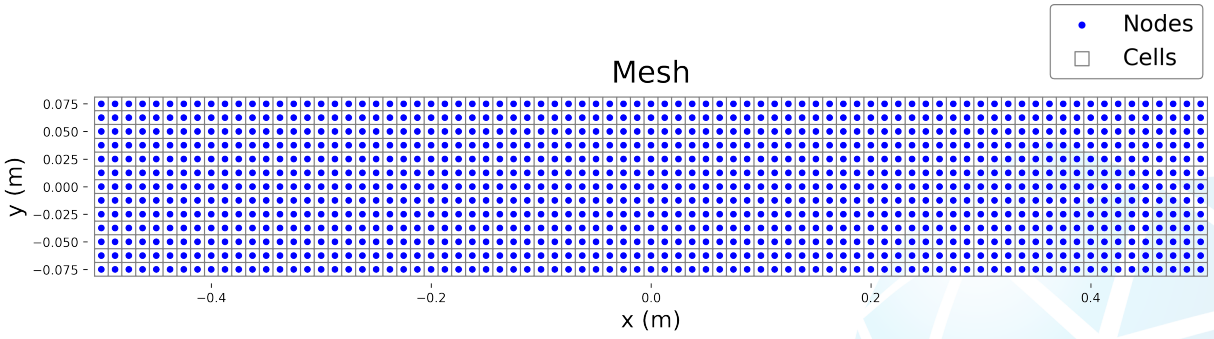
Então, duplicam-se os vetores x e y até que fiquem com tamanho *cols rows*, o vetor y é reordenado em ordem crescente para fazer pares com o vetor x e por fim junta-se os dois vetores em um vetor de duas colunas, formando o vetor dos pontos x_i, tornando a forma da matriz mostra na equação (3.7).

$$x_i = \begin{bmatrix} 0 & 0 \\ h & 0 \\ 2h & 0 \\ \vdots & \vdots \\ size_x & 0 \\ 0 & h \\ h & h \\ 2h & h \\ \vdots & \vdots \\ size_x & h \\ h & 2h \\ 2h & 2h \\ \vdots & \vdots \\ size_x & 2h \\ h & size_y \\ 2h & size_y \\ \vdots & \vdots \\ size_x & size_y \end{bmatrix} \tag{3.7}$$

Com os pontos x_i calculados, o programa automaticamente atualiza o objeto com alguns atributos que serão utilizados posteriormente, como o próprio h fornecido, a área de uma célula A, os pontos dados por $points$, as coordenadas x e y dos pontos por conveniência, os vértices dados por $vertices$, entre outros.

A figura 3.2 mostra um exemplo de uma malha retangular gerada com dimensões x = 1 m, y = 0,15 m e espaçamento nodal h = 0,0125 m, plotada com as configurações padrões.

Figura 3.2 – Exemplo de uma malha retangular centralizada



Fonte: Elaborado pelo autor.

A figura 3.3 mostra a criação da mesma malha, porém rotacionada em +30°, do eixo +x para o eixo +y e com a opção de remover as células ativada. Na plotagem, mostrou-se que é possível realizar algumas alterações, como alteração do título, cores e tamanhos.

Figura 3.3 – Exemplo de uma malha retangular centralizada e rotacionada

Fonte: Elaborado pelo autor.

3.2.2 Geração de uma malha não retangular

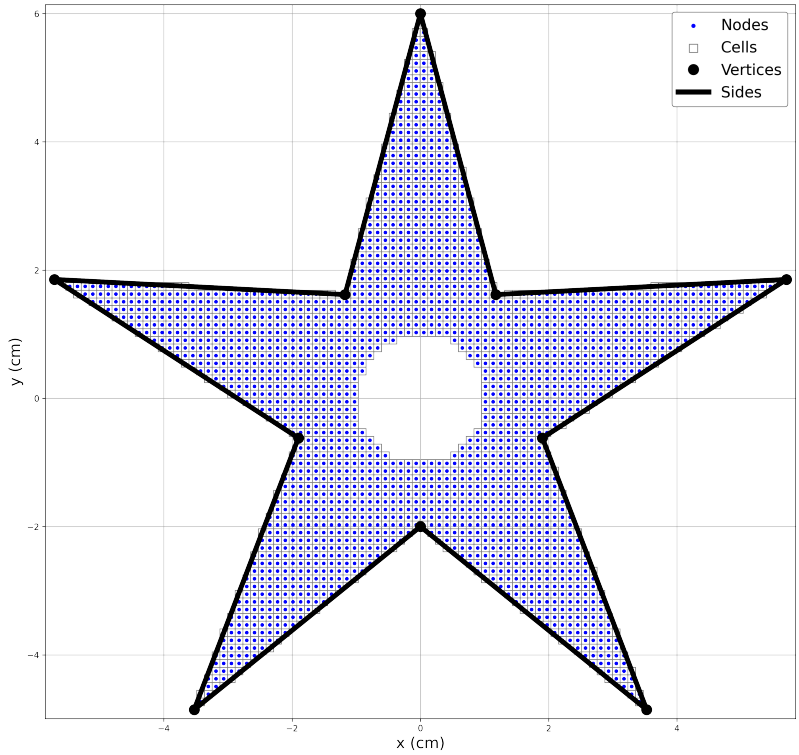
Para a geração de malhas não retangulares, utiliza-se da mesma técnica utilizada para a malha retangular, porém, depois que os pontos são definidos, faz-se uma filtragem para selecionar apenas os pontos que estão contidos dentro do polígono que deve ser utilizado na entrada da função.

A figura 3.4 mostra o exemplo de uma malha não retangular gerada com os pontos (aproximados), representando unidades de distância, no caso, cm:

$$P = \begin{bmatrix} 5{,}71 & 1{,}85 \\ 1{,}18 & 1{,}62 \\ -1{,}18 & 1{,}62 \\ -5{,}71 & 1{,}85 \\ -1{,}90 & -0{,}62 \\ -3{,}53 & -4{,}85 \\ 0{,}00 & -2{,}00 \\ 3{,}53 & -4{,}85 \\ 1{,}90 & -0{,}62 \end{bmatrix}$$

Posteriormente foi utilizado o método de remoção de círculos, sendo este com centro em (0,0) e raio 1 cm. Na plotagem, escolheu-se mostrar os vértices, os lados, a grade, o quadro e customizar o posicionamento da legenda e remover o título, tudo feito diretamente pela classe *Mesh*.

Figura 3.4 – Exemplo de uma malha não retangular com um círculo removido



Fonte: Elaborado pelo autor.

3.2.3 Uso da classe *Mesh*

Para gerar uma malha com a classe *Mesh*, precisa-se primeiramente importar o código peridinâmico e ter acesso à classe *Mesh*. Então, é necessário instanciar a classe em uma nova variável. Feito isso, aplica-se os métodos desejados nesta variável.

A maioria dos métodos atualiza os atributos da própria variável, não sendo necessário carregar um alto número de variáveis pelo código.

A figura 3.5 mostra o uso básico da classe, cuja imagem representativa do resultado final é exposto na figura 3.2.

Figura 3.5 – Exemplo da criação de uma malha retangular

```
1  import peridynamics as pd
2
3  mesh=pd.Mesh()
4  mesh.generate_rectangular_mesh(h=0.0125,size_x=1,size_y=0.15)
5  mesh.center()
6  mesh.plot()
```

Fonte: Elaborado pelo autor.

3.2.4 Métodos da classe *Mesh*

O quadro 3.1 mostra os métodos expostos ao usuário ao instanciar a classe em uma variável.

Quadro 3.1 – Métodos fornecidos pela classe *Mesh*

Método	**Descrição**
center	Centraliza a malha no ponto (0,0).
generate_mesh_constrained_within_points	Gera uma malha contida nos pontos fornecidos. Os pontos são vértices de um polígono.
generate_rectangular_mesh	Gera uma malha retangular.
get_borders	Fornece pontos correspondentes às bordas.
get_borders_bool	Retorna um *array* de booleanos que apontam verdadeiro para pontos das bordas.
get_borders_ind	Retorna um array com os índices correspondentes às bordas.
get_fig_ax	Retorna objetos de figura e eixo da biblioteca *matplotlib*.
offset	Move a malha em um *offset* (x,y).
plot	Plota o gráfico da malha.
remove_circles	Permite a remoção de pontos x_i contidos em círculos baseados em seus centros e raios.
rotate	Rotaciona a malha.
set_unit	Atribui unidade à medida de distância (que por padrão é m).

Fonte: Elaborado pelo autor.

3.2.5 Atributos da classe *Mesh*

A figura 3.6 mostra os atributos gerados automaticamente para o caso da malha gerada na figura 3.2. É importante observar que alguns destes atributos são dinâmicos, ou seja, só são gerados quando necessários, isto é, chamados pela primeira vez e posteriormente são armazenados no cache do computador, não sendo necessário recalculá-los. Todos os atributos funcionam com os métodos fornecidos pela classe e são atualizados se necessário. Uma única exceção é quanto ao atributo *grid*, que, por sua natureza, não faz sentido imediato para malhas não retangulares. No entanto, partes posteriores do código levam isso em conta e aplicam métodos de correção.

Figura 3.6 – Alguns atributos calculados automaticamente

```
4  mesh.__dict__
✓ 0.2s

Output exceeds the size limit. Open the full output data in a text editor
{'rotation_angle_rad': 0,
 'type': 'rectangular',
 'unit': 'm',
 'h': 0.0125,
 'A': 0.00015625000000000003,
 'vertices': array([[-0.5  , -0.075],
        [ 0.5  , -0.075],
        [ 0.5  ,  0.075],
        [-0.5  ,  0.075],
        [-0.5  , -0.075]]),
 'points': array([[-0.5   , -0.075 ],
        [-0.4875, -0.075 ],
        [-0.475 , -0.075 ],
        ...,
        [ 0.475 ,  0.075 ],
        [ 0.4875,  0.075 ],
        [ 0.5   ,  0.075 ]]),
 'x': array([-0.5   , -0.4875, -0.475 , ...,  0.475 ,  0.4875,  0.5   ]),
 'y': array([-0.075, -0.075, -0.075, ...,  0.075,  0.075,  0.075]),
 '_fig': <Figure size 1600x430.569 with 1 Axes>,
 '_ax': <AxesSubplot: title={'center': 'Mesh'}, xlabel='x (m)', ylabel='y (m)'>,
 'counters': array([3, 5, 5, ..., 5, 5, 3]),
 'grid': (array([[-0.5   , -0.4875, -0.475 , ...,  0.475 ,  0.4875,  0.5   ],
        [-0.5   , -0.4875, -0.475 , ...,  0.475 ,  0.4875,  0.5   ],
        [-0.5   , -0.4875, -0.475 , ...,  0.475 ,  0.4875,  0.5   ],
...
        [ 0.05  ,  0.05  ,  0.05  , ...,  0.05  ,  0.05  ,  0.05  ],
        [ 0.0625,  0.0625,  0.0625, ...,  0.0625,  0.0625,  0.0625],
        [ 0.075 ,  0.075 ,  0.075 , ...,  0.075 ,  0.075 ,  0.075 ]])),
 'ncols': 81,
 'nrows': 13}
```

Fonte: Elaborado pelo autor.

3.3 Família, Áreas Parciais e Correção de Volume/Área

A família e áreas parciais estão relacionadas, pois ambas são calculadas de uma só vez, alterando o método de cálculo conforme o algoritmo utilizado.

Neste trabalho, foram implantados os seguintes algoritmos da literatura:

- *FA*
- *PA-PDLAMMPS*
- *PA-HHB*

- *PA-AC*
- *IPA-AC*

Além disso, alguns desses algoritmos foram vetorizados, recebendo melhorias significativas no tempo de execução. Foi adicionado o prefixo “+” para separá-los dos algoritmos canônicos:

- *FA+*
- *PA-PDLAMMPS+*
- *PA-HHB+*

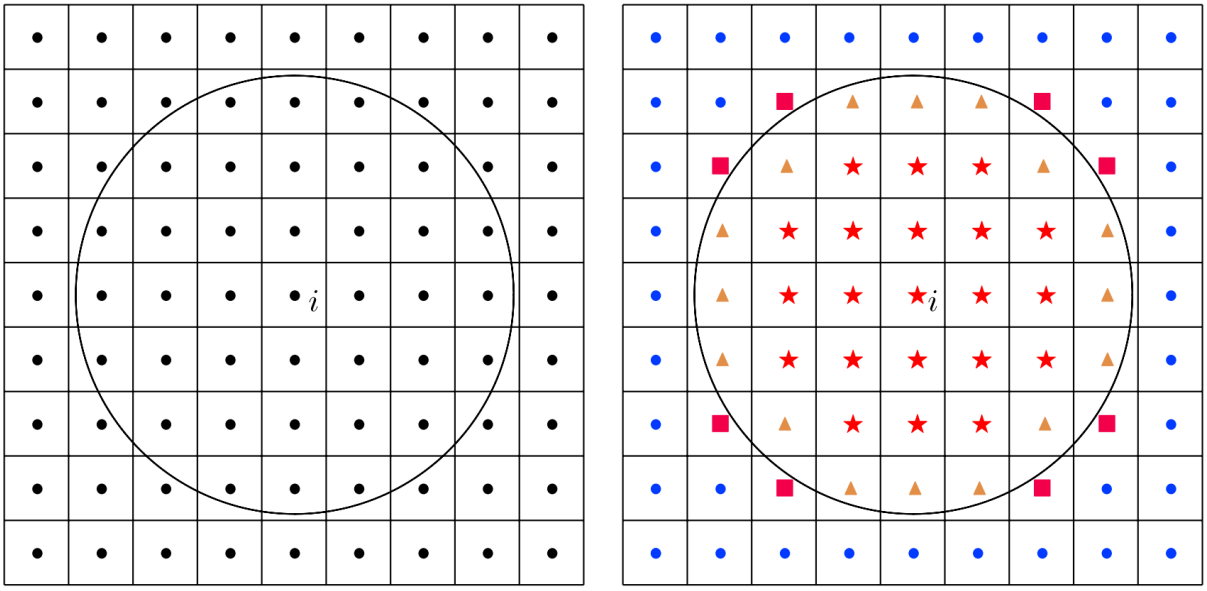
3.3.1 Considerações

O domínio discretizado $\mathcal{B}_D$ com espaçamento regular h nos dois eixos pode ter seus pontos x_i separados em 4 classes (SELESON, 2014):

- ★ – Classe 1: pontos cujas células estão completamente dentro da vizinhança de i;
- ▲ – Classe 2: pontos que estão dentro da vizinhança de i, porém suas células se sobrepõem parcialmente com a vizinhança de i;
- ■ – Classe 3: pontos que estão fora da vizinhança de i, mas suas células se sobrepõem parcialmente com a vizinhança de i;
- ● – Classe 4: pontos cujas células estão completamente fora da vizinhança de i.

A figura 3.7 mostra um domínio discretizado $\mathcal{B}_D$ contendo os 4 tipos de classes de pontos.

Figura 3.7 – Vizinhança e classe de pontos considerados



(a) Neighborhood of i on a square lattice.

(b) Classes of lattice points with respect to the neighborhood of i.

Fonte: (SELESON, 2014).

Esses pontos apresentados serão importantes nas definições dos algoritmos de área parcial utilizados, especialmente no caso do algoritmo *PA-AC* e seus derivados.

3.3.2 *FA*

O algoritmo *Full Area* (*FA*) é o mais simples dos algoritmos implementados. Foi desenvolvido originalmente por Silling e Askari (2005). Neste trabalho, utilizou-se como base o algoritmo conforme colocado por Seleson (2014).

Para cada ponto x_i, ele considera simplesmente pontos de classe 1 e classe 2, ou seja, somente os pontos x_j que satisfazem a condição $|x_j - x_i| \leq \delta$. A área de todos os pontos x_j são consideradas como h^2. Os pontos de classe 3 não são considerados, ou seja, é como se tivessem a área 0 (SELESON, 2014).

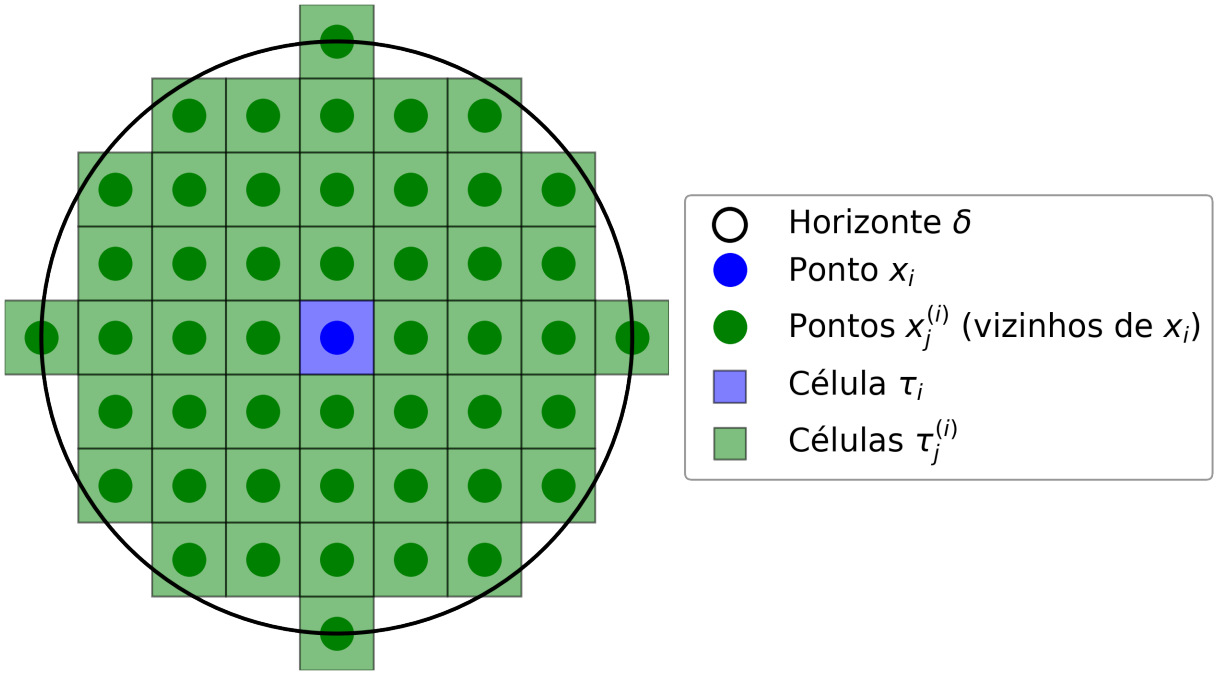
Desta forma, os pontos de classe 1 são corretamente avaliados, mas os pontos de classe 2 são considerados como tendo uma área maior do que realmente têm. E os pontos de classe 3, que deveriam contribuir com um valor pequeno de área, são desconsiderados.

Algoritmo 1 *FA*

1: $||\xi|| = ||x_j - x_i||$ ▷ Computar as distâncias entre os centros das células
2: **if** $||\xi|| \leq \delta$ **then** ▷ Verificar se o ponto j pertence à família de i
3: $\quad A_j^{(i)} = h^2$
4: **else**
5: $\quad A_j^{(i)} = 0$
6: **end if**
7: **return** $A_j^{(i)}$

Aplicando o algoritmo 1 no programa desenvolvido em *Python* e considerando uma malha com espaçamento $h = 0{,}0125\ m$ e um horizonte $\delta = 0{,}005\ m$ (razão de malha $m = \frac{\delta}{h} = 4$), obtém-se, ao redor do ponto x_i localizado no centro de uma malha de dimensões maiores ou iguais a δ em ambos os eixos, a seguinte família mostrada na figura 3.8.

Figura 3.8 – Família gerada pelo algoritmo *FA*



Fonte: Elaborado pelo autor.

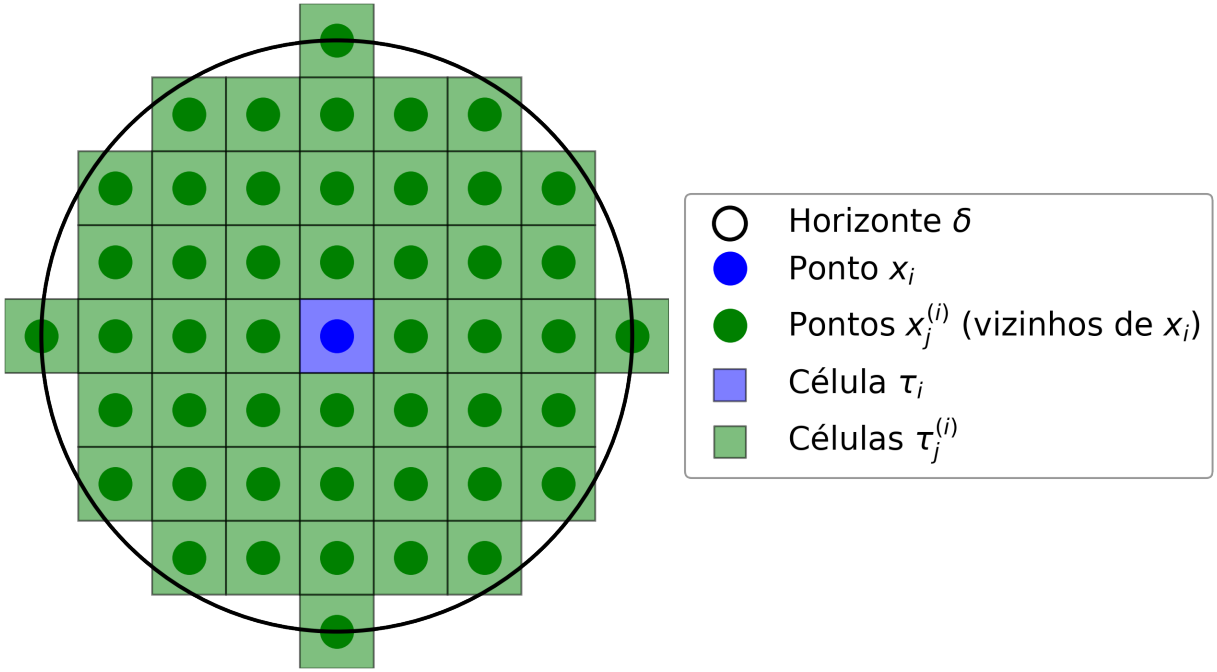
3.3.3 *FA+*

Algoritmo 2 *FA+*

1: $||\xi|| = ||x_j - x_i||$ ▷ Computar as distâncias entre os centros das células
2: $j^{(i)} =$ índices onde $||\xi|| \leq \delta$ ▷ Cria o vetor de vizinhos de i
3: $A_j^{(i)} =$ vetor do tamanho de $j^{(i)}$ com todos os valores iguais a h^2
4: **return** $j^{(i)}$,$A_j^{(i)}$

Aplicando o algoritmo 2 no programa desenvolvido em *Python* e considerando as mesmas condições utilizas na subseção 3.3.2, a figura 3.9 mostra que a família obtida pelo algoritmo *FA+* é idêntica à família obtida pelo algoritmo *FA*. Porém, por ser vetorizado, seu cálculo é realizado de forma significantemente mais rápida.

Figura 3.9 – Família gerada pelo algoritmo *FA+*



Fonte: Elaborado pelo autor.

Considerando uma malha retangular de dimensões x = y = 1 m e um espaçamento de malha h = 0,01 m , temos uma malha com aproximadamente 10.000 elementos. Utilizando uma razão de malha d = 4, temos um horizonte δ = 0,04 m. A tabela 3.1 mostra a comparação do algoritmo vetorizado contra o seu algoritmo canônico sendo aplicado nesta malha.

Tabela 3.1 – Comparação de tempo entre os algoritmos *FA* e *FA+*

Tempo levado pelo algoritmo *FA* (s)	Tempo levado pelo algoritmo *FA+* (s)
246,00±0,59	1,74±0,02

Fonte: Elaborado pelo autor.

Neste caso, o algoritmo vetorizado mostrou-se aproximadamente 140 vezes mais rápido!

Para obter este dado, o algoritmo foi executado 7 vezes em uma malha gerada pelo *software* com a ajuda da função mágica de linha *timeit* disponibilizada em um notebook. Os resultados originais se encontram na figura 3.10.

Figura 3.10 – Metodologia da obtenção de comparação de tempos dos algoritmos não vetorizados vs algoritmos vetorizado

```
1  %timeit pd.Family(mesh1,horizon1,m,'FA','None',silent=True)
2  %timeit pd.Family(mesh1,horizon1,m,'FA+','None',silent=True)
3  %timeit pd.Family(mesh1,horizon1,m,'PA-PDLAMMPS','None',silent=True)
4  %timeit pd.Family(mesh1,horizon1,m,'PA-PDLAMMPS+','None',silent=True)
5  %timeit pd.Family(mesh1,horizon1,m,'PA-HHB','None',silent=True)
6  %timeit pd.Family(mesh1,horizon1,m,'PA-HHB+','None',silent=True)
✓ 105m 10.7s

4min 6s ± 591 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
1.74 s ± 19.8 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
4min 23s ± 3.12 s per loop (mean ± std. dev. of 7 runs, 1 loop each)
2.07 s ± 37.7 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
4min 32s ± 2.01 s per loop (mean ± std. dev. of 7 runs, 1 loop each)
2.13 s ± 29.8 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Fonte: Elaborado pelo autor.

3.3.4 *PA-PDLAMMPS*

O algoritmo *PA-PDLAMMPS* tem esse nome por utilizar áreas parciais - *Partial Areas* (*PA*) - e por ter sido utilizado no *software PDLAMMPS* (SELESON, 2014).

Semelhante ao algoritmo *FA*, ele não leva em consideração pontos da classe 3, porém ele tenta melhorar o cálculo da área parcial de pontos de classe 2, conforme mostra a linha 5 do algoritmo 3 (SELESON, 2014).

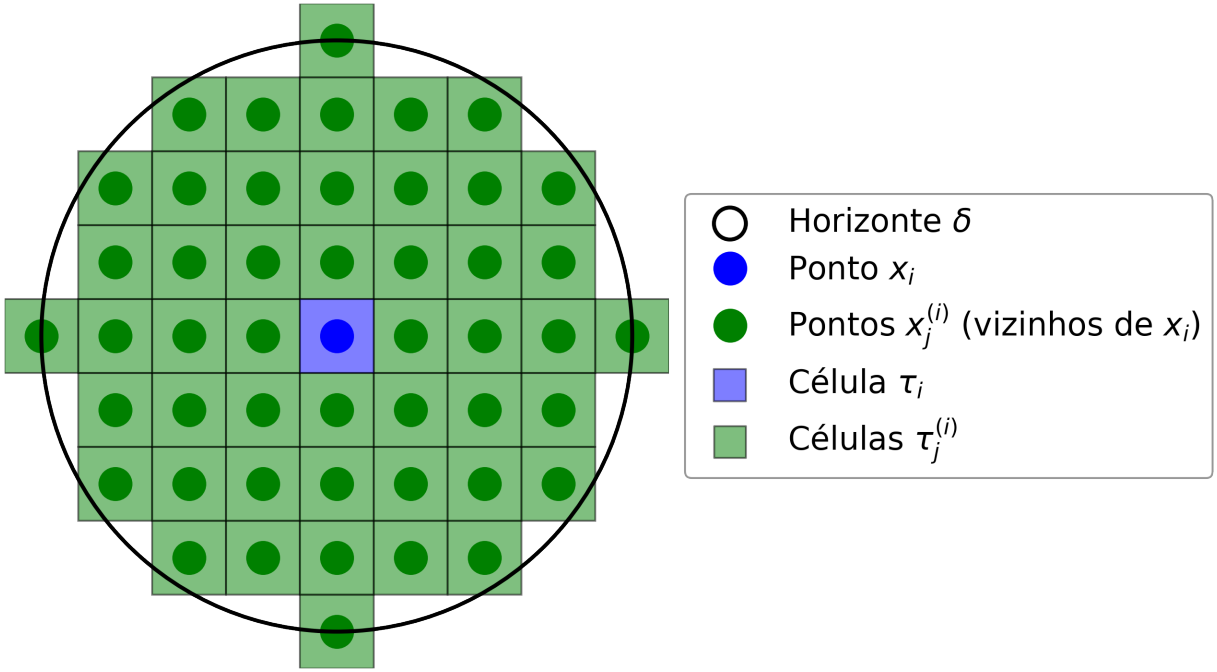
Algoritmo 3 *PA-PDLAMMPS*

1: $||\xi|| = ||x_j - x_i||$ ▷ Computar as distâncias entre os centros das células
2: **if** $||\xi|| + \frac{h}{2} \leq \delta$ **then** ▷ Verificar se a célula τ_j está contida na vizinhança de i
3: $\quad A_j^{(i)} = h^2$
4: **else if** $||\xi|| \leq \delta$ **then** ▷ Verificar se o ponto j pertence à família de i
5: $\quad A_j^{(i)} = \frac{1}{h}\left[\delta - \left(||\xi|| - \frac{h}{2}\right)\right] h^2$
6: **else**
7: $\quad A_j^{(i)} = 0$
8: **end if**
9: **return** $A_j^{(i)}$

Aplicando o algoritmo 3 no programa desenvolvido em *Python* e considerando as mesmas condições utilizas na subseção 3.3.2, a figura 3.11 mostra a família obtida utilizando o algoritmo *PA-PDLAMMPS*. É interessante observar que os pontos x_j são

iguais ao obtidos utilizando o algoritmo *FA*. A única alteração é a consideração acerca da área parcial dos pontos de classe 2.

Figura 3.11 – Família gerada pelo algoritmo *PA-PDLAMMPS*



Fonte: Elaborado pelo autor.

3.3.5 *PA-PDLAMMPS+*

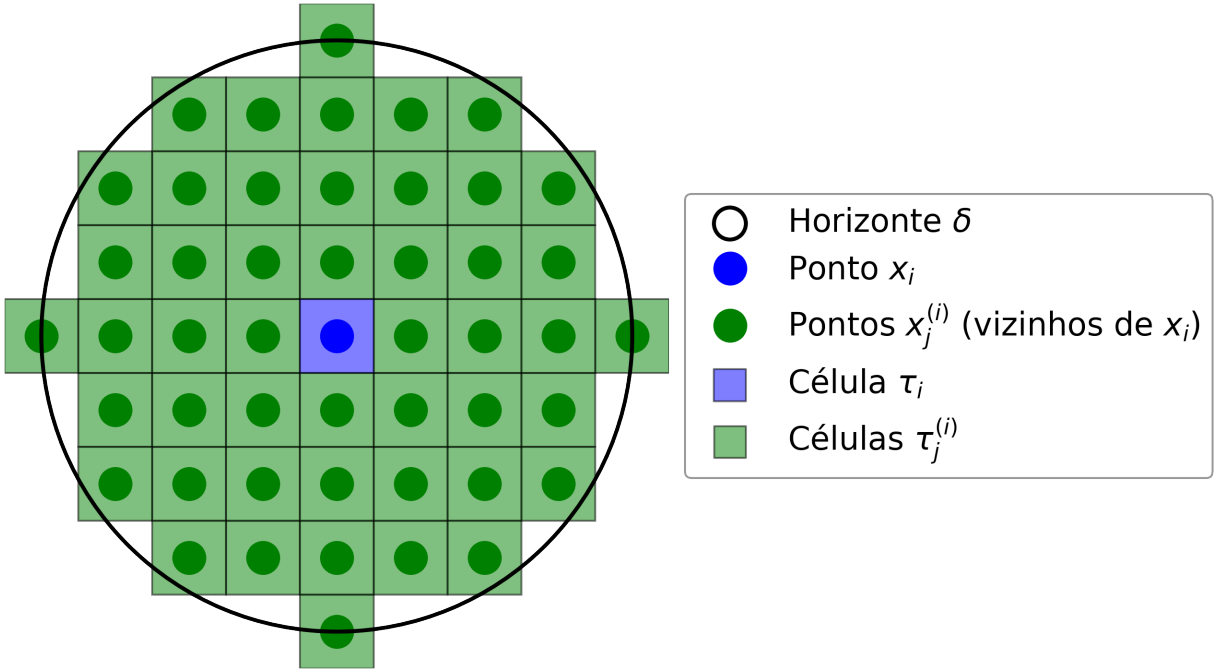
Algoritmo 4 *PA-PDLAMMPS+*

1: $||\xi|| = ||x_j - x_i||$ ▷ Computar as distâncias entre os centros das células
2: $k1 =$ índices onde $||\xi|| + \frac{h}{2} \leq \delta$ & $||\xi|| > 0$ ▷ Índices de vizinhos sob a condição 1
3: $j1 =$ comprimento de $k1$
4: $k2 =$ índices onde $||\xi|| \leq \delta$ & $||\xi|| > 0$ ▷ Índices de vizinhos sob a condição 2
5: $k2 =$ remover elementos de k2 iguais a k1
6: $j2 =$ comprimento de $k2$
7: Inicializar $j^{(i)}$ com tamanho j1+j2
8: $j^{(i)}[1 : j1] = k1$
9: $j^{(i)}[j1 + 1 : j1 + j2] = k2$
10: Inicializar $A_j^{(i)}$ com tamanho j1+j2
11: $A_j^{(i)}[1 : j1] = h^2$
12: $A_j^{(i)}[j1 + 1 : j1 + j2] = \frac{1}{h}\left[\delta - \left(||\xi|| - \frac{h}{2}\right)\right] h^2$
13: **return** $j^{(i)}, A_j^{(i)}$

Aplicando o algoritmo 4 no programa desenvolvido em *Python* e considerando as mesmas condições utilizas na subseção 3.3.2, a figura 3.12 mostra que a famí-

lia obtida pelo algoritmo *PA-PDLAMMPS+* é idêntica à família obtida pelo algoritmo *PA-PDLAMMPS*. Porém, por ser vetorizado, seu cálculo é realizado de forma significantemente mais rápida.

Figura 3.12 – Família gerada pelo algoritmo *PA-PDLAMMPS+*



Fonte: Elaborado pelo autor.

Considerando uma malha retangular de dimensões x = y = 1 m e um espaçamento de malha h = 0,01 m , temos uma malha com aproximadamente 10.000 elementos. Utilizando uma razão de malha d = 4, temos um horizonte δ = 0,04 m. A tabela 3.2 mostra a comparação do algoritmo vetorizado contra o seu algoritmo canônico sendo aplicado nesta malha.

Tabela 3.2 – Comparação de tempo entre os algoritmos *PA-PDLAMMPS* e *PA-PDLAMMPS+*

Tempo alg. *PA-PDLAMMPS* (s)	Tempo alg. *PA-PDLAMMPS+* (s)
263,00±3,12	2,07±0,04

Fonte: Elaborado pelo autor.

Neste caso, o algoritmo vetorizado mostrou-se aproximadamente 130 vezes mais rápido!

3.3.6 *PA-HHB*

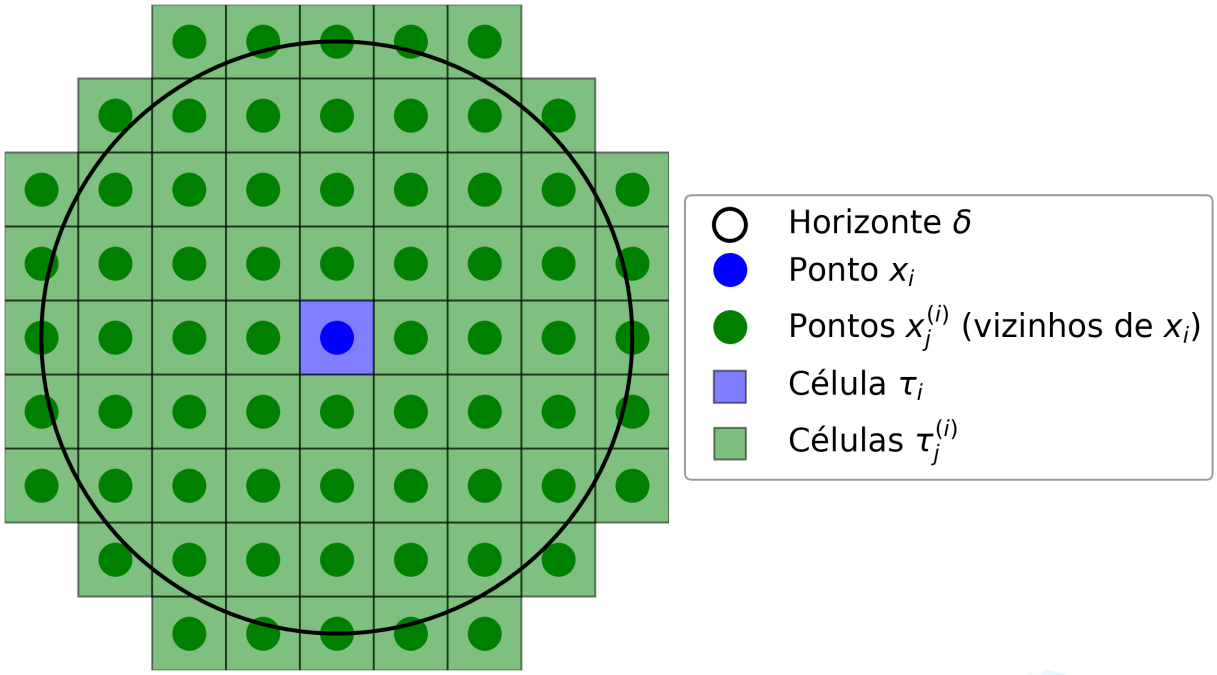
Este algoritmo utiliza áreas parciais - *Partial Areas* (*PA*) - e foi proposto por *Hu*, *Ha* e *Bobaru*, daí o seu nome. Seu propósito foi tentar melhora o algoritmo *PA-PDLAMMPS* ao incluir pontos de classe 3 (SELESON, 2014).

Algoritmo 5 *PA-HHB*

1: $||\xi|| = ||x_j - x_i||$ ▷ Computar as distâncias entre os centros das células
2: **if** $||\xi|| + \frac{h}{2} \leq \delta$ **then** ▷ Verificar se a célula τ_j está contida na vizinhança de i
3: $\quad A_j^{(i)} = h^2$
4: **else if** $||\xi|| - \frac{h}{2} \leq \delta$ **then** ▷ Verifica rse o ponto j pertence à família de i
5: $\quad A_j^{(i)} = \frac{1}{h}\left[\delta - \left(||\xi|| - \frac{h}{2}\right)\right] h^2$
6: **else**
7: $\quad A_j^{(i)} = 0$
8: **end if**
9: **return** $A_j^{(i)}$

Aplicando o algoritmo 5 no programa desenvolvido em *Python* e considerando as mesmas condições utilizas na subseção 3.3.2, a figura 3.13 mostra a família obtida utilizando o algoritmo *PA-HHB*. Este algoritmo, quando comparados às versões "anteriores", passa a incluir pontos de classe 3.

Figura 3.13 – Família gerada pelo algoritmo *PA-HHB*



Fonte: Elaborado pelo autor.

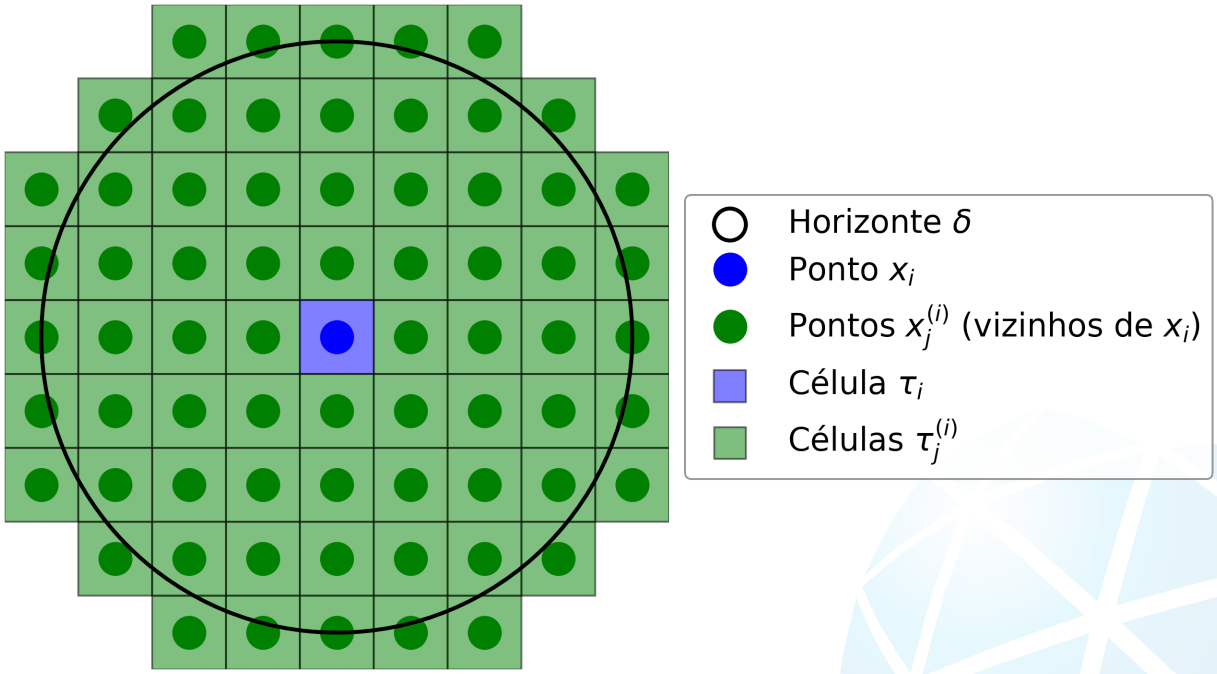
3.3.7 *PA-HHB+*

Algoritmo 6 *PA-HHB+*

1: $||\xi|| = ||x_j - x_i||$ ▷ Computar as distâncias entre os centros das células
2: $k1 =$ índices onde $||\xi|| + \frac{h}{2} \leq \delta$ & $||\xi|| > 0$ ▷ Índices de vizinhos sob a condição 1
3: $j1 =$ comprimento de $k1$
4: $k2 =$ índices onde $||\xi|| - \frac{h}{2} \leq \delta$ & $||\xi|| > 0$ ▷ Índices de vizinhos sob a condição 2
5: $k2 =$ remover elementos de k2 iguais a k1
6: $j2 =$ comprimento de $k2$
7: Inicializar $j^{(i)}$ com tamanho j1+j2
8: $j^{(i)}[1 : j1] = k1$
9: $j^{(i)}[j1 + 1 : j1 + j2] = k2$
10: Inicializar $A_j^{(i)}$ com tamanho j1+j2
11: $A_j^{(i)}[1 : j1] = h^2$
12: $A_j^{(i)}[j1 + 1 : j1 + j2] = \frac{1}{h}\left[\delta - \left(||\xi|| - \frac{h}{2}\right)\right] h^2$
13: **return** $j^{(i)}, A_j^{(i)}$

Aplicando o algoritmo 6 no programa desenvolvido em *Python* e considerando as mesmas condições utilizas na subseção 3.3.2, a figura 3.14 mostra que a família obtida pelo algoritmo *PA-HHB+* é idêntica à família obtida pelo algoritmo *PA-HHB*. Porém, por ser vetorizado, seu cálculo é realizado de forma significantemente mais rápida.

Figura 3.14 – Família gerada pelo algoritmo *PA-HHB+*



Fonte: Elaborado pelo autor.

Considerando uma malha retangular de dimensões x = y = 1 m e um espaçamento de malha h = 0,01 m , temos uma malha com aproximadamente 10.000 elementos. Utilizando uma razão de malha d = 4, temos um horizonte δ = 0,04 m. A tabela 3.3 mostra a comparação do algoritmo vetorizado contra o seu algoritmo canônico sendo aplicado nesta malha.

Tabela 3.3 – Comparação de tempo entre os algoritmos *PA-HHB* e *PA-HHB+*

Tempo levado pelo alg. *PA-HHB* (s)	Tempo levado pelo alg. *PA-HHB+* (s)
272,00±2,01	2,13±0,03

Fonte: Elaborado pelo autor.

Neste caso, o algoritmo vetorizado mostrou-se aproximadamente 130 vezes mais rápido!

3.3.8 *PA-AC*

O algoritmo *Partial Area Analytical Calcultation* (*PA-AC*), desenvolvido por Seleson (2014), busca melhorar a contribuição das áreas parciais dos pontos de classe 2 e 3 por meio de cálculos analíticos.

Além disso, este algoritmo é mais rápido por ser otimizado ao realizar comparações somente com pontos localizados nas proximidades de x_i, porém ele foi projetado para funcionar em malhas retangulares geradas de forma homogênea, podendo causar problemas em malhas não homogêneas. Este algoritmo é mais rápido que os algoritmos canônicos apresentados anteriormente, no entanto, suas versões vetorizadas são mais rápidas.

O algoritmo 7 mostra o começo do algoritmo, ou seja, realiza operações antecedentes ao algoritmo em si.

Algoritmo 7 *PA-AC*: interação com a família

$N_x = \left\lfloor \frac{\delta}{h} + \frac{1}{2} - \epsilon \right\rfloor$ ▷ Computar o máximo de vizinhos possível ao longo da direção x
for $k = i - N_x : i + N_x$ **do**
  if k=i **then**
    $N_y = N_x$
  else
    $|\xi_1| = |x_j - x_i|$
    $N_y = \left\lfloor \frac{\sqrt{\delta^2 - (|\xi_1| - \frac{h}{2})^2}}{h} + \frac{1}{2} - \epsilon \right\rfloor$
  end if
  for $l = j - N_y : j + N_y$ **do**
    Computar interações
  end for
end for

O algoritmo 8 mostra o funcionamento do algoritmo *PA-AC*. Os casos em questão estão disponibilizados no apêndice A.

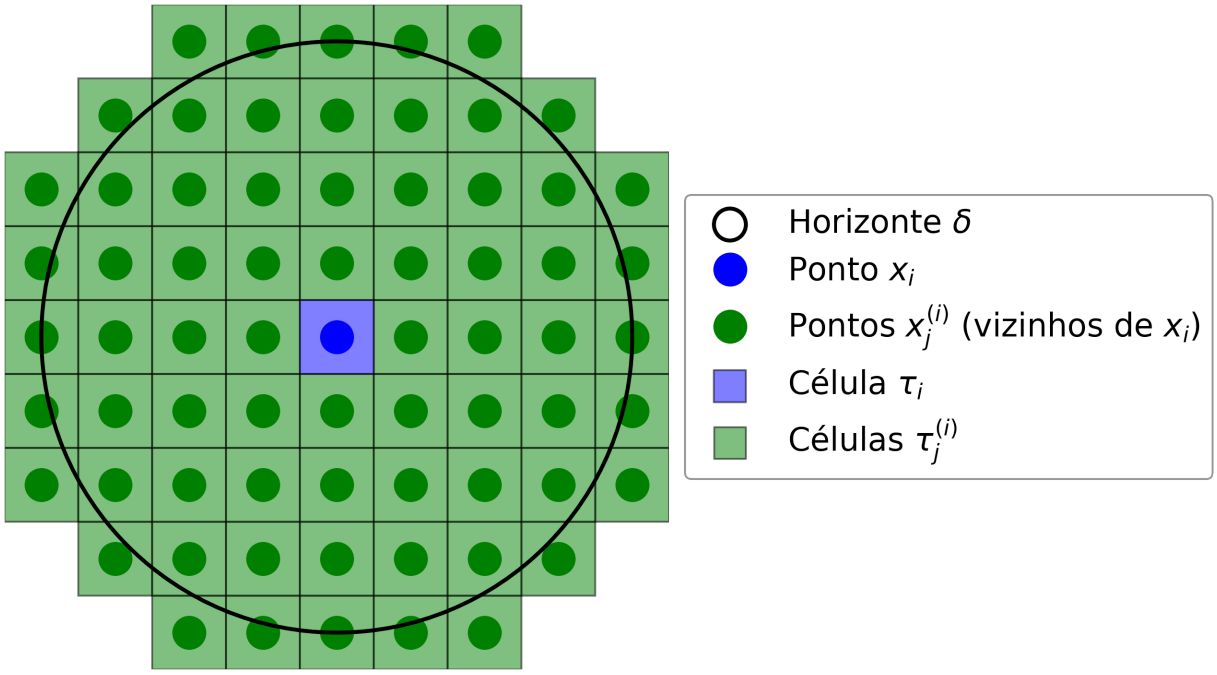
Algorithm 8 *PA-AC*

Mapear a célula para o quadrante superior direito:
if $x_j < x_i$ **then**
  $x_j = xi + (x_i - x_j)$ ▷ Transladar a célula para a direita
end if
if $y_j < y_i$ **then**
  $y_j = yi + (y_i - y_j)$ ▷ Transladar a célula para cima
end if
if $x_j = x_i$ **then** ▷ Rotacionar a célula do eixo +y para o eixo +x
  $\Delta = y_j - y_i$
  $y_j = y_i$
  $x_j = x_i + \Delta$
end if
Contar os números de cantos de τ_j dentro da vizinhança de i:
counter $= 0$
for $n = -1{,}1$ **do**
  for $m = -1{,}1$ **do**
    if $\left(x_j + n\frac{h}{2} - x_i\right)^2 + \left(y_j + m\frac{h}{2} - y_i\right)^2 < \delta^2$ **then**
      counter $=$ *counter* $+ 1$
    end if

end for
end for
Computar área parcial:
if *counter* $= 4$ **then**
    $A_j^{(i)} = h^2$ ▷ Caso I
else if *counter* $= 3$ **then**
    Computar $A_j^{(i)}$ com o Caso II
else if *counter* $= 2$ **then**
    if $y_j = y_i$ **then**
        if $x_j + \frac{h}{2} \geq x_i + \delta$ **then**
            Computar $A_j^{(i)}$ com o Caso III(a2)
        else
            Computar $A_j^{(i)}$ com o Caso III(b)
        end if
    else
        $r_{br}^2 = \left(x_j + \frac{h}{2} - x_i\right)^2 + \left(y_j + \frac{h}{2} - y_i\right)^2$ ▷ Distância ao quadrado do canto inferior direito de i até τ_j
        if $r_{br}^2 > \delta^2$ **then**
            Computar $A_j^{(i)}$ com o Caso III(a1)
        else
            Computar $A_j^{(i)}$ com o Caso III(c)
        end if
    end if
else if *counter* $= 1$ **then**
    Computar $A_j^{(i)}$ com o Caso IV
else
    if $x_j = \frac{h}{2} < x_i + \delta$ **then**
        Computar $A_j^{(i)}$ com o Caso V
    else
        $A_j^{(i)} = 0$
    end if
end if
return $A_j^{(i)}$

Aplicando o algoritmo 8 no programa desenvolvido em *Python* e considerando as mesmas condições utilizas na subseção 3.3.2, a figura 3.15 mostra a família obtida utilizando o algoritmo *PA-AC*. Este algoritmo, apresenta a mesma família que o algoritmo *PA-HHB*, porém os valores das áreas parciais são calculadas de forma analítica.

Figura 3.15 – Família gerada pelo algoritmo *PA-AC*



Fonte: Elaborado pelo autor.

3.3.9 *IPA-AC*

O algoritmo *Improved Partial Area Analytical Calcultation* (*IPA-AC*) dá um passo adiante em relação ao algoritmo *PA-AC* ao retornar os centroides das regiões de intersecção e é apresentado no algoritmo 9.

Da mesma forma que no algoritmo anterior, os casos em questão se encontram no apêndice A.

Algorithm 9 *IPA-AC*

Inicializar as bandeiras:
flag$_{\text{left–right}}$ = false
flag$_{\text{bottom–top}}$ = false
flag$_{\text{rotate}}$ = false
Mapear a célula para o quadrante superior direito:
if $x_j < x_i$ **then**
    $x_j = xi + (x_i - x_j)$ ▷ Transladar a célula para a direita
    flag$_{\text{left–right}}$ = true
end if
if $y_j < y_i$ **then**
    $y_j = yi + (y_i - y_j)$ ▷ Transladar a célula para cima
    flag$_{\text{bottom–top}}$ = true
end if
if $x_j = x_i$ **then** ▷ Rotacionar a célula do eixo +y para o eixo +x

$\Delta = y_j - y_i$
$y_j = y_i$
$x_j = x_i + \Delta$
$\text{flag}_{\text{rotate}}$ = true
end if
Inicializar as coordenadas do centroide de i:
$\bar{x}_j^{(i)} = x_j$
$\bar{y}_j^{(i)} = y_j$
Contar os números de cantos de τ_j dentro da vizinhança de i:
counter $= 0$
for $n = -1,1$ **do**
  for $m = -1,1$ **do**
    if $\left(x_j + n\frac{h}{2} - x_i\right)^2 + \left(y_j + m\frac{h}{2} - y_i\right)^2 < \delta^2$ **then**
      counter $=$ *counter* $+ 1$
    end if
  end for
end for
Computar área parcial e controide:
if *counter* $= 4$ **then**
  $A_j^{(i)} = h^2$ ▷ Caso I
else if *counter* $= 3$ **then**
  Computar $A_j^{(i)}$, $\bar{x}_j^{(i)}$ e $\bar{y}_j^{(i)}$ com o Caso II
else if *counter* $= 2$ **then**
  if $y_j = y_i$ **then**
    if $x_j + \frac{h}{2} \geq x_i + \delta$ **then**
      Computar $A_j^{(i)}$, $\bar{x}_j^{(i)}$ e $\bar{y}_j^{(i)}$ com o Caso III(a2)
    else
      Computar $A_j^{(i)}$, $\bar{x}_j^{(i)}$ e $\bar{y}_j^{(i)}$ com o Caso III(b)
    end if
  else
    $r_{br}^2 = \left(x_j + \frac{h}{2} - x_i\right)^2 + \left(y_j + \frac{h}{2} - y_i\right)^2$ ▷ Distância ao quadrado do canto inferior direito de i até τ_j
    if $r_{br}^2 > \delta^2$ **then**
      Computar $A_j^{(i)}$, $\bar{x}_j^{(i)}$ e $\bar{y}_j^{(i)}$ com o Caso III(a1)
    else
      Computar $A_j^{(i)}$, $\bar{x}_j^{(i)}$ e $\bar{y}_j^{(i)}$ com o Caso III(c)
    end if

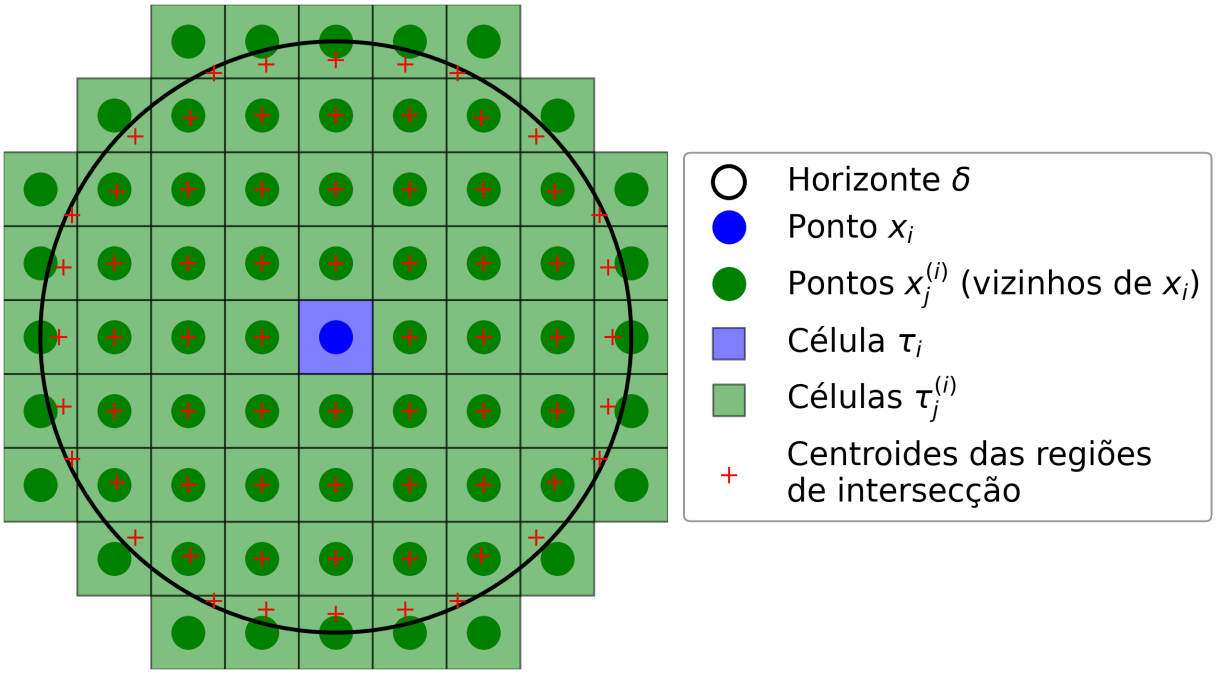
end if
else if *counter* $= 1$ **then**
    Computar $A_j^{(i)}$, $\bar{x}_j^{(i)}$ e $\bar{y}_j^{(i)}$ com o Caso IV
else
    if $x_j = \dfrac{h}{2} < x_i + \delta$ **then**
        Computar $A_j^{(i)}$, $\bar{x}_j^{(i)}$ e $\bar{y}_j^{(i)}$ com o Caso V
    else
        $A_j^{(i)} = 0$
    end if
end if

Mapear o centroide de volta para o quadrante original de τ_j:

if $\text{flag}_{\text{rotate}}$ = true **then** ▷ Rotacionar o centroide do eixo +x para o eixo +y
    $\Delta = \bar{x}_j^{(i)} - x_i$
    $\bar{x}_j^{(i)} = x_i$
    $\bar{y}_j^{(i)} = y_i + \Delta$
else if $\text{flag}_{\text{bottom–top}}$ = true **then** ▷ Transladar o centroide para baixo
    $\bar{y}_j^{(i)} = y_i - \left(\bar{y}_j^{(i)} - y_i\right)$
else if $\text{flag}_{\text{left–right}}$ = true **then** ▷ Transladar o centroide para a esquerda
    $\bar{x}_j^{(i)} = x_i - \left(\bar{x}_j^{(i)} - x_i\right)$
end if
return $A_j^{(i)}, \bar{x}_j^{(i)}, \bar{y}_j^{(i)}$

Aplicando o algoritmo 9 no programa desenvolvido em *Python* e considerando as mesmas condições utilizas na subseção 3.3.2, a figura 3.16 mostra a família obtida utilizando o algoritmo *IPA-AC*. Este algoritmo, apresenta a mesma família e as mesmas áreas parciais que o algoritmo *PA-AC*, porém introduz centroides das regiões de intersecção.

Figura 3.16 – Família gerada pelo algoritmo *IPA-AC*



Fonte: Elaborado pelo autor.

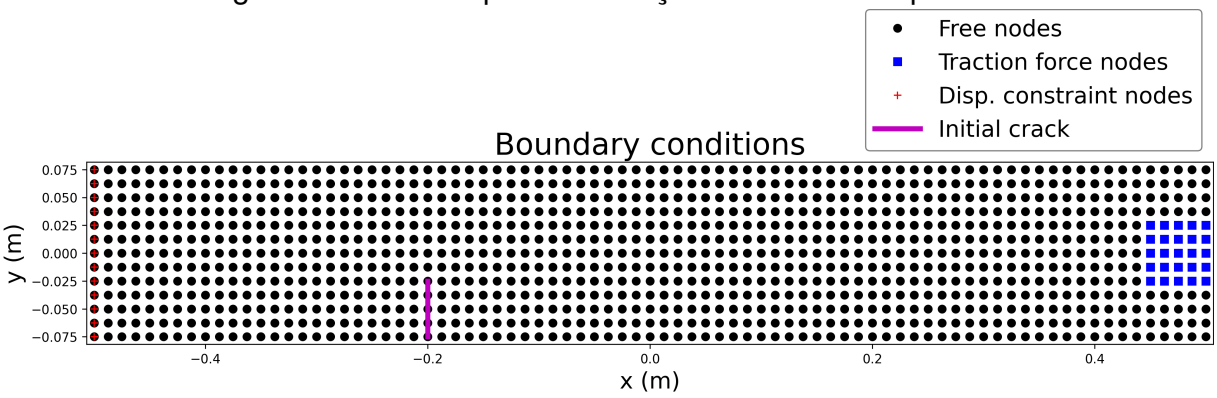
3.4 Condições de Contorno

É possível colocar condições de contorno de imposições de deslocamentos, forças externas aplicadas no corpo, velocidades nos centros dos potos x_i, segmentos iniciais da trinca e uma bandeira que indica se o modelo está sofrendo dano (importante para simulações com trinca) e se o dano é dependente do micro-módulo de elasticidade.

A figura 3.17 mostra um exemplo de uma condição de contorno aplicada em uma malha retangular, aonde são impostos que:

- Os pontos em pretos são pontos livres, isto é, não possuem condições de contorno de deslocamento, força ou velocidade;
- Os “+” da esquerda têm seus deslocamentos prescritos;
- Quadrados azuis na direita recebem forças externas;
- A linha em magenta representa os segmentos de uma trinca inicial;
- Neste caso, não há condições de contorno de velocidade;
- Embora não apareça na imagem, a condição de contorno também inclue as bandeiras mencionadas.

Figura 3.17 – Exemplo de condição de contorno aplicada



Fonte: Elaborado pelo autor.

O programa transforma estas informações em vetores armazenados em forma de atributo na variável que é instância da classe *BoundaryConditions*.

O atributo *bc_set* é um vetor de 3 colunas que representa o índice do ponto na condição N*2D na primeira coluna, os valores dos deslocamentos prescritos na segunda coluna e os valores da velocidade na terceira coluna. Assim, supondo que um ponto x_i tenha condições prescrita de deslocamento nulo, tanto em x quanto em y, suas duas linhas correspondentes no vetor *bc_set* fica conforme a equação (3.8)[1].

$$bc\_set = \begin{bmatrix} \text{índice de } x^{(i)} & 0 & nan \\ \text{índice de } y^{(i)} & 0 & nan \end{bmatrix} \tag{3.8}$$

Supondo que no exemplo mostrado na figura 3.17 todos os deslocamentos prescritos tenham sido nulos e nenhuma velocidade foi prescrita, a figura 3.18 mostra o vetor retornado pelo programa.

[1] Caso o ponto x_i tenha apenas pelo menos uma condição de velocidade ou deslocamento em somente uma dimensão, o vetor *bc_set* só terá uma linha correspondente ao ponto x_i

Figura 3.18 – Condições de contorno de deslocamento e velocidade na condição N=2D

```
   1  bc.bc_set
 ✓  0.2s
array([[0.000e+00, 0.000e+00,       nan],
       [1.000e+00, 0.000e+00,       nan],
       [1.620e+02, 0.000e+00,       nan],
       [1.630e+02, 0.000e+00,       nan],
       [3.240e+02, 0.000e+00,       nan],
       [3.250e+02, 0.000e+00,       nan],
       [4.860e+02, 0.000e+00,       nan],
       [4.870e+02, 0.000e+00,       nan],
       [6.480e+02, 0.000e+00,       nan],
       [6.490e+02, 0.000e+00,       nan],
       [8.100e+02, 0.000e+00,       nan],
       [8.110e+02, 0.000e+00,       nan],
       [9.720e+02, 0.000e+00,       nan],
       [9.730e+02, 0.000e+00,       nan],
       [1.134e+03, 0.000e+00,       nan],
       [1.135e+03, 0.000e+00,       nan],
       [1.296e+03, 0.000e+00,       nan],
       [1.297e+03, 0.000e+00,       nan],
       [1.458e+03, 0.000e+00,       nan],
       [1.459e+03, 0.000e+00,       nan],
       [1.620e+03, 0.000e+00,       nan],
       [1.621e+03, 0.000e+00,       nan],
       [1.782e+03, 0.000e+00,       nan],
       [1.783e+03, 0.000e+00,       nan],
       [1.944e+03, 0.000e+00,       nan],
       [1.945e+03, 0.000e+00,       nan]])
```

Fonte: Elaborado pelo autor.

Nota: Embora a primeira coluna esteja representada como ponto flutuante, ela representa os índices dos pontos na condição N=2D, que devem ser tratados como números inteiros.

De forma semelhante, o atributo *bb* expõe as forças aplicadas na forma N=2D, porém o vetor não retorna índices, e sim, o valor direto da força correspondente ao índice, pois as forças que não foram prescritas são consideradas como 0. A equação (3.9) mostra a estrutura deste vetor.

$$bb = \left[F_x^{(i)}, F_y^{(i)}, F_x^{(i+1)}, F_y^{(i+1)}, \ldots, F_x^{(n)}, F_y^{(n)}\right] \tag{3.9}$$

onde i representa o índice do ponto e n o número de pontos.

Por ser normalmente muito grande, será mostrado na figura 3.19 um exemplo de um *array* estruturado[2], que passa a mesma informação, porém é apenas utilizado para facilitar a visualização.

[2]Um *array* estruturado permite separar o tipo de dado por coluna, porém, isto traz alguma complicações em operações vetoriais. Sendo assim, ele não é utilizado nos com intuito de realização de cálculos.

Figura 3.19 – Forças externas aplicadas no corpo

```
with np.printoptions(linewidth=40):
    print(bc.bodyForce.dtype)
    print(bc.bodyForce)
```

✓ 0.2s

```
[('Node index', '<i4'), ('Force x value', '<f8'), ('Force y value', '<f8')]
[(400, 0., -4.8e+08)
 (401, 0., -4.8e+08)
 (402, 0., -4.8e+08)
 (403, 0., -4.8e+08)
 (404, 0., -4.8e+08)
 (481, 0., -4.8e+08)
 (482, 0., -4.8e+08)
 (483, 0., -4.8e+08)
 (484, 0., -4.8e+08)
 (485, 0., -4.8e+08)
 (562, 0., -4.8e+08)
 (563, 0., -4.8e+08)
 (564, 0., -4.8e+08)
 (565, 0., -4.8e+08)
 (566, 0., -4.8e+08)
 (643, 0., -4.8e+08)
 (644, 0., -4.8e+08)
 (645, 0., -4.8e+08)
 (646, 0., -4.8e+08)
 (647, 0., -4.8e+08)
 (724, 0., -4.8e+08)
 (725, 0., -4.8e+08)
 (726, 0., -4.8e+08)
 (727, 0., -4.8e+08)
 (728, 0., -4.8e+08)]
```

Fonte: Elaborado pelo autor.

3.5 Modelo

O programa considera que o modelo é uma variável instanciada da classe *Model*. O modelo é inicializado de acordo com o seu nome, o vetor *par_omega*[3] o módulo de *Young* E, a taxa de liberação de energia G_0 e objeto *bc*, que é uma instância da classe *BoundaryConditions* contendo as condições de contorno relevantes à simulação desejada. Os modelos baseados no modelo *LSJT* também precisam da variável *dt*, que será abordada na seção 3.6.

A variável G_0 não é muito utilizada na literatura quando comparado, por exemplo, à variável E, causando uma dificuldade na definição do material da simulação. Os outros parâmetros, como o módulo de *Young* E, a densidade ρ e o módulo de *Poisson* ν já são mais amplamente exploradas.

Tendo ciência desta limitação, o modelo é criado e contém informações sobre:

- Bandeira indicando se o modelo trabalha com dilatação;
- Bandeira indicando se o modelo é linear;

[3] O vetor *par_omega* contém informações à respeito da função influência, sendo *par_omega* $= [\delta, \omega_\delta, \gamma]$, conforme ilustrado no capítulo 1

- Bandeira indicando se o modelo possui uma matriz de rigidez analítica[4];
- A(s) constante(s) relacionada(s) ao micro-módulo de elasticidade;
- Os parâmetros α, β e γ relacionados ao dano sofrido;
- Uma função que retorna o fator de dano com as considerações do modelo;
- Uma função que retorna a força no estado vetorial;
- Uma função que retorna a densidade da energia de deformação;
- Se o modelo possui a matriz de rigidez analítica, ela é exposta;

A figura 3.20 mostra um exemplo dos métodos e atributos que definem o modelo *PMB*.

Figura 3.20 – Exemplo da definição de um modelo (*PMB*)

```
1  import peridynamics as pd
2
3  model=pd.Model('PMB',par_omega,E,G0,bc)
4  model.__dict__
✓ 0.2s
{'name': 'PMB',
 'dilatation': False,
 'dilatHt': False,
 'linearity': False,
 'stiffnessAnal': True,
 'c': array([6.6004738e+15]),
 'T': <function peridynamics.Model.Model.__init__.<locals>.T(mesh, u, ii, family, bc)>,
 'analyticalStiffnessMatrix': <function peridynamics.Model.Model.__init__.<locals>.analyticalStiffnessMatrix(mesh, u, bc, family, par_omega, penalty, mu=None)>,
 'strainEnergyDensity': <function peridynamics.Model.Model.__init__.<locals>.strainEnergyDensity(mesh, u, family, ii, bc, par_omega) -> float>}
```

Fonte: Elaborado pelo autor.

O *software* inclui os modelos: *LBB*, *LPS 2D*, *LSJ-T*, *Lipton Free Damage*, *PMB* e *PMB DTT*, tendo uma filosofia modular de tal modo que é possível modificar ou adicionar modelos conforme o necessário.

Neste ponto, o programa está pronto para prosseguir para a próxima etapa: executar um *solver* para obter os resultados desejados.

3.6 *Solvers*

Os *solver* implementados no programa são o *quasi-static* e o *dynamic-explicit*.

Em ambos os casos, é necessário definir as propriedades do material, a malha dada pela classe *Mesh*, a família dada pela classe *Family*, as condições de contorno definidas pela classe *BoundaryConditions*, o modelo definido pela classe *Model* e alguns parâmetros específicos do solver.

[4] Se o modelo possui uma matriz de rigidez analítica, normalmente o código é executado de forma significativamente mais rápida e possui maior precisão.

3.6.1 *Quase-static*

O solver *quase-static* considera que o o termo de inércia da equação (3.1) é nulo. Assim, a equação discreta governante é considerada conforme a equação (3.10) (PATRIOTA, 2019).

$$\sum_{j \in \mathcal{F}_i} \mathbf{f}_{ij} \Delta V_{ij} + \mathbf{b}(x_i) = 0 \tag{3.10}$$

O método baseia-se no método de *Newton*. No entanto, as condições de carga, ao invés de serem passos de tempo, são atualizadas a cada passo. Isso permite configurações desbalanceadas capazes de alterar o campo de deslocamento da solução (PATRIOTA, 2019).

A ideia principal do método é calcular a solução de campo de deslocamento que minimiza o escalar residual r para um determinado passo. O escalar residual é calculado da forma vetorial de r conforme a sua distância com o infinito (∞-*norm* ou L_2-*norm*), sendo definidos conforme a equação (3.12) (PATRIOTA, 2019).

Outro ponto importante é o cálculo da matriz jacobiana K que busca aproximar a solução com o passo anterior. Neste contexto, a matriz jacobiana é a matriz de rigidez tangente dada pela equação (3.11) (PATRIOTA, 2019).

$$K_{lk}(\overline{\mathbf{u}}) = \frac{\partial \bar{f}_{\text{model},l}}{\partial \bar{u}_k}(\overline{\mathbf{u}}) \tag{3.11}$$

$$\|\mathbf{r}\|_\infty := \max\{r_i\} \quad \|\mathbf{r}\|_{L_2} := \sqrt{\sum_{i=1}^{n_r} r_i^2} \tag{3.12}$$

O algoritmo 10 contém o código implementado no programa de (PATRIOTA, 2019).

Algoritmo 10 *Quasi-static solver*

procedure Procedimento: achar a solução do carregamento ($\bar{u}^n$)
  $\bar{u}^0 = 0$ ▷ Passo 1: Inicialização do vetor de graus de liberdade dos deslocamentos
  for each load step n from 1 to n_{tot} **do**
    $\bar{u}^n \leftarrow \bar{u}^{n-1}$ ▷ Passo 2: Atualizar as condições de carga e o chute da solução Passo 3: Calcular o vetor residual, $\mathbf{r}$, r residual e deternubar o critério de convergência para cada passo de carga
    $\mathbf{r} \leftarrow \bar{f}^n_{model} + \bar{b}^n \bar{V}^T$
    $r \leftarrow ||\mathbf{r}||_\infty$
    $r_{\max} \leftarrow \varepsilon \max \left\{ ||\bar{f}^n_{model}||_\infty, ||\bar{b}^n \bar{V}^T||_\infty \right\}$
    while $r > r_{\max}$ **do**
      $K^n \leftarrow$ *tangenStiffness*$(\bar{u}^n)$ ▷ Cálculo da matriz de rigidez tangente Cálculo da solução incremente e atualização do chute
      $\Delta\bar{u} = -(K^n)^{-1} r$
      $\bar{u}^n = \bar{u}^n + \Delta\bar{u}$
      $\mathbf{r} \leftarrow \bar{f}^n_{model} + \bar{b}^n \bar{V}^T$
      $r \leftarrow ||\mathbf{r}||_\infty$
    end while
  end for
end procedure

3.6.2 *Dynamic-explicit*

O *solver dynamic-explicit* realiza integração temporal baseada em um método de diferenças centrais chamado de *Velocity-Verlet scheme*, conforme dado pelo algoritmo 11 (PATRIOTA, 2019).

Algoritmo 11 *Velocity-Verlet scheme*

procedure Atualizar o próximo deslocamento no passo do tempo (u_i^{n+1})
  $v_i^{n+1/2} \leftarrow v_i^n + \dfrac{\Delta t}{a_i^n}$
  $u_i^{n+1} \leftarrow u_i^n + \Delta t v_i^{n+1/2}$
  $a_i^{n+1} \leftarrow \dfrac{1}{\rho_i} \left(\sum_{j \in \mathrm{F}_i} f_{ij}^{n+1} \Delta V_{ij} + b^{n+1} \right)$
  $v_i^{n+1} \leftarrow v_i^{n+1/2} + \dfrac{\Delta t}{2} a_i^{n+1}$
end procedure

Um fato importante que deve ser levado em consideração é que, como é uma método explícito, o incremento de tempo Δt deve ser pequeno o suficiente para garantir estabilidade (PATRIOTA, 2019).

(SILLING; ASKARI, 2005) propôs que o valor crítico (máximo) de incremento de tempo como sendo:

$$\Delta t_{crit} = \min_i \left(\sqrt{\frac{2\rho}{\sum_{p \in \mathcal{F}_i} C_{ip} \Delta V_{ip}}} \right) \tag{3.13}$$

Porém, resultados práticos indicam que basta utilizar um valor suficientemente pequeno.

3.7 Pós-processamento

O pós-processamento permite visualizar os gráficos referentes às variáveis obtidas pelo solver que estão relacionadas à malha. O *software* permite visualização de gráficos de:

- Deslocamento;
- Deformação;
- Energias;
- Índice de dano;
- Seguir trincas.

A função *plot_displacement* gera 4 figuras. A primeira possui 2 subfiguras, sendo um gráfico de superfície, onde o plano xy representa a malha discretizada e o eixo z representa os deslocamentos em x. A outra subfigura é idêntica, porém com os deslocamentos de y no eixo z.

A segunda figura é em 2 dimensões. Ela mostra a malha deformada em contraste com a malha de referência, representadas por retângulos. Um fator de escala[5] é automaticamente calculado e aplicado nas deformações.

A terceira figura é um gráfico de *quiver*, mostrando vetores que representam o módulo e deslocamento das deformações.

A quarta figura é idêntica à 2ª, porém é feita com pontos ao invés de retângulos, sendo classificado como um gráfico de dispersão.

A função *plot_strain* plota as deformações ε em um gráfico de superfície de 3 dimensões em uma figura contendo 3 subfiguras. A primeira subfigura apresenta o eixo xy como sendo a malha discretizada e o eixo z como sendo a deformação ε_x. A sengunda contém as deformações ε_y no eixo z e a terceira contém as deformações ε_{xy} no eixo z.

[5] As deformações podem ser muito pequenas, sendo necessário exagerá-las para visualização. Caso contrário, os gráficos da malha original e da malha deformada ficariam sobrepostos.

A função *plot_energy* plota duas figuras relacionadas à energia. A primeira figura é um gráfico de superfície em 3 dimensões, onde o plano xy representa a malha discretizada e o eixo y representa a densidade energética de deformação. A segunda figura é uma gráfico de 2 dimensões contendo as revoluções das energia de deformação (W), energia cinética (KE), trabalho externo (EW) e energia interna total.

A função *plot_damage* plota uma figura em duas dimensões. Ela mostra um mapa de cor no plano, com sua escala baseado em valores mínimos e máximos de ϕ.

A função *plot_crack* acompanha a trinca[6].

[6]No momento em que este trabalho foi escrito esta função não foi 100 % finalizada.

4 RESULTADOS E DISCUSSÃO

A intenção deste capítulo é tentar validar o código por meio de algum experimento físico realizado.

Será considerado um exemplo próximo do fornecido por Patriota (2019) nomeado de *experiment_template* no seu código fonte.

O experimento consiste na aplicação de tensões em um corpo de material *soda-lime glass* com as propriedades conforme a tabela 4.1.

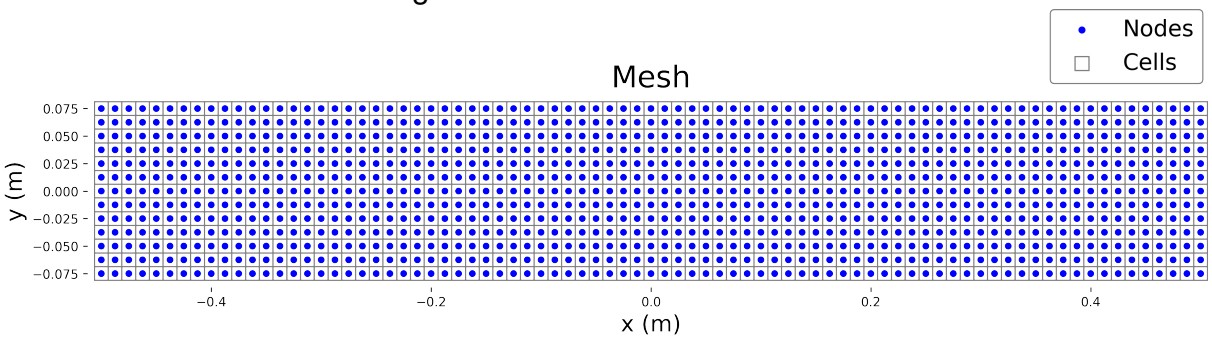
Tabela 4.1 – Propriedades do material

Propriedade	Valor
Módulo de *Young* E	72 GPa
Densidade ρ	2440 kg.m^{-3}
Coef. de *Poisson Young* ν	0,22
Taxa de liberação de energia G_0	3,8 J.m^{-2}

Fonte: (PATRIOTA, 2019).

Considerou-se uma malha retangular de dimensões comprimento = 1 m e altura = 0,15 m. Para a discretização um horizonte δ = 0,05 m e uma razão de malha d = 4. Desta forma, o espaçamento da malha foi de h = 0,0125 m. A malha discretizada se encontra na figura 4.1.

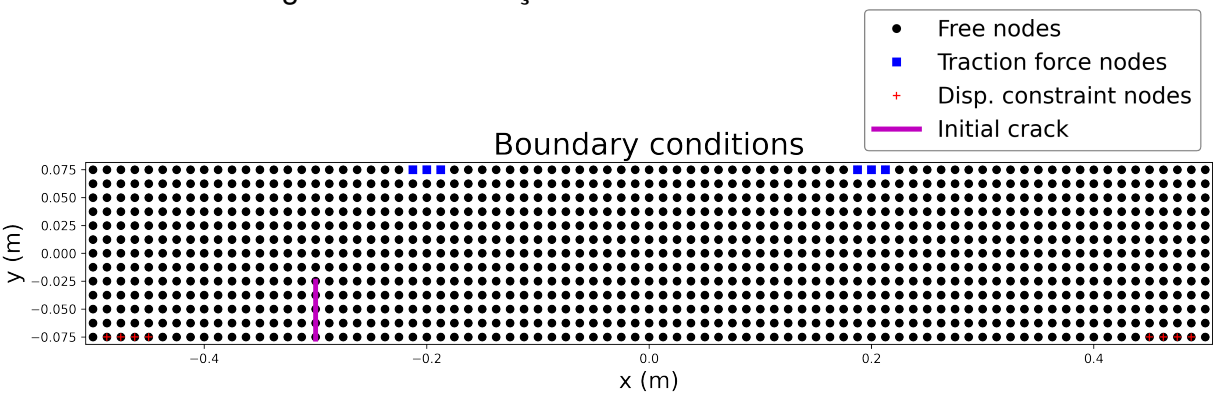
Figura 4.1 – Malha considerada



Fonte: Elaborado pelo autor.

Como condições de contorno, foram consideradas tensões de 6 MPa no sentido de -y e os deslocamentos foram considerados como 0. A figura 4.2 ilustra quais pontos receberam estas considerações.

Figura 4.2 – Condições de contorno consideradas



Fonte: Elaborado pelo autor.

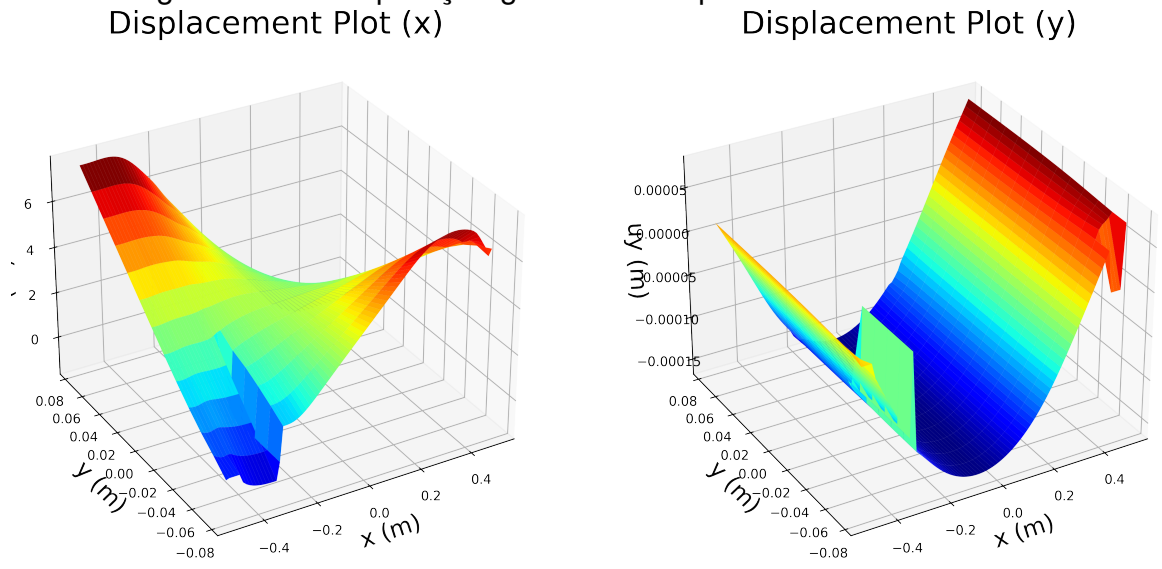
Além disso, não foi considerado aplicação de dano no modelo.

O modelo escolhido foi o *PMB*. É um modelo não linear que aceita danos externos (vide figura 4.6).

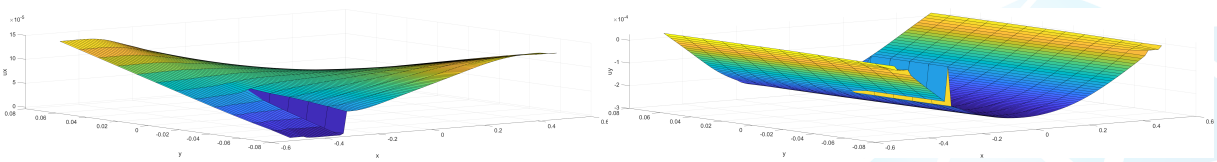
O solver escolhido foi o *quasi-static*, com 4 passos de carga.

Na figura 4.3, comparamos os resultados do gráfico de superfície de deslocamento.

Figura 4.3 – Comparação gráficos de superfície de deslocamento



(a) Programa desenvolvido em *Python*



(b) Programa de Patriota (2019)

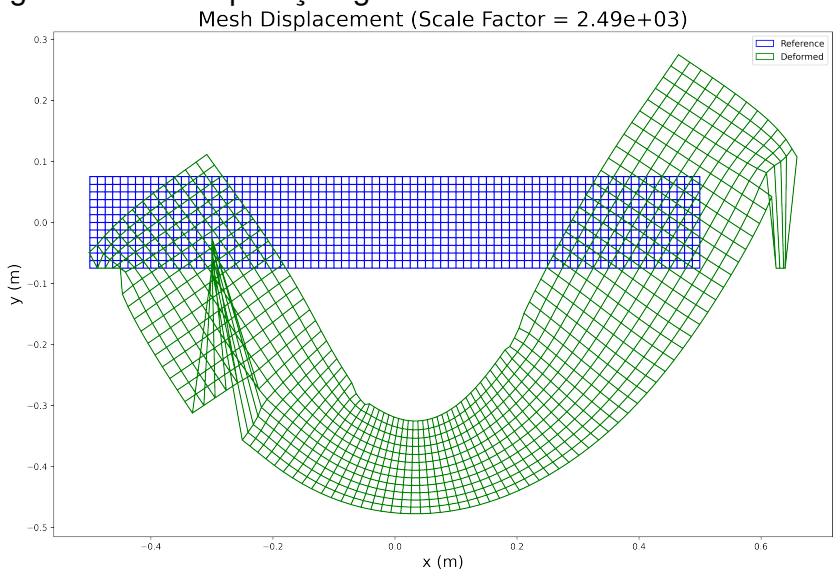
(c) Programa de Patriota (2019)

Fonte: Elaborado pelo autor.

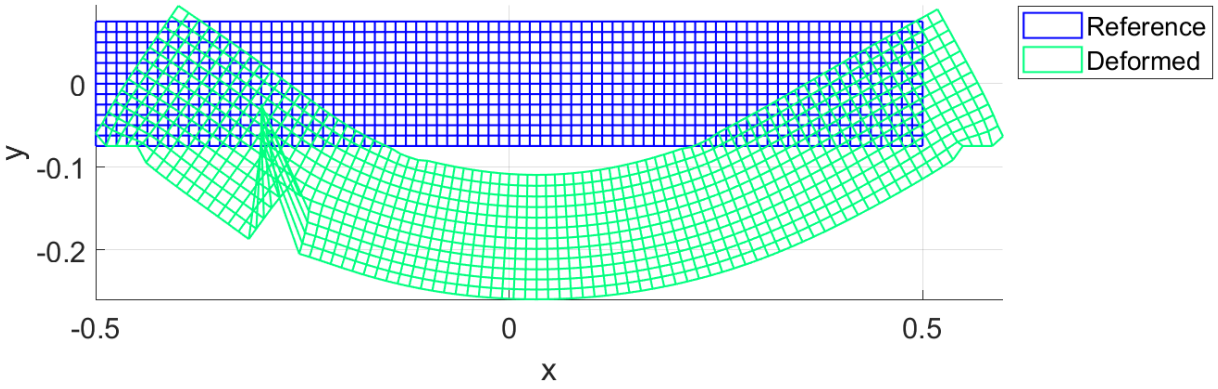
A figura 4.3 mostra que há alto grau de deslocamento nos pontos em vermelho, especialmente no canto inferior direito, ande apresenta o mair pico de energia. Vemos que há uma diferença abrupta de deslocamento nos pontos aonde foi explicitado uma condição de contorno de deslocamento.

A figura 4.4 mostra outra visualização acerca do deslocamento. Desta vez, com foco na malha em duas dimensões.

Figura 4.4 – Comparação gráficos de deslocamento da malha



(a) Programa desenvolvido em *Python*

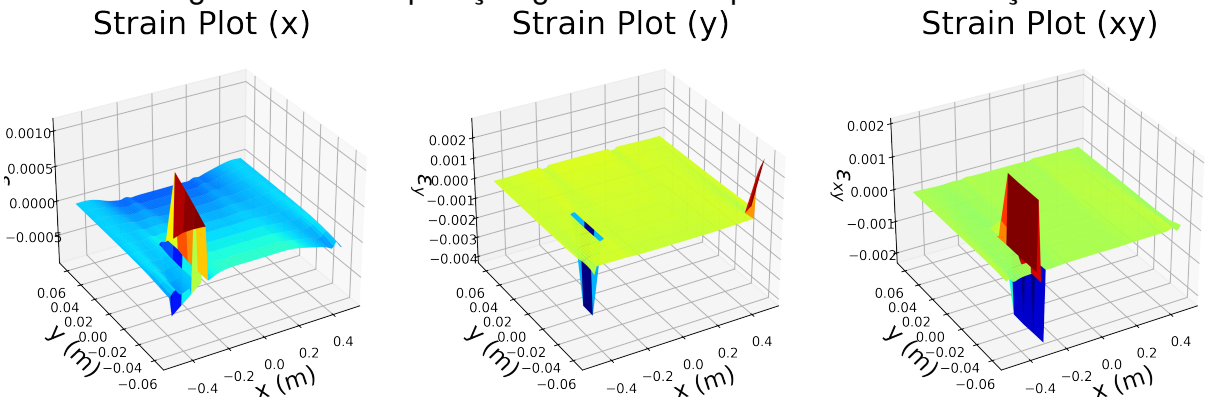


(b) Programa de Patriota (2019)

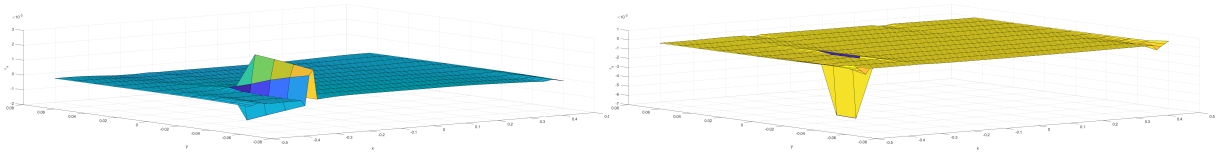
Fonte: Elaborado pelo autor.

A figura 4.5 mostra a comparação de gráfico de deformação.

Figura 4.5 – Comparação gráficos de superfície de deformação



(a) Programa desenvolvido em *Python*



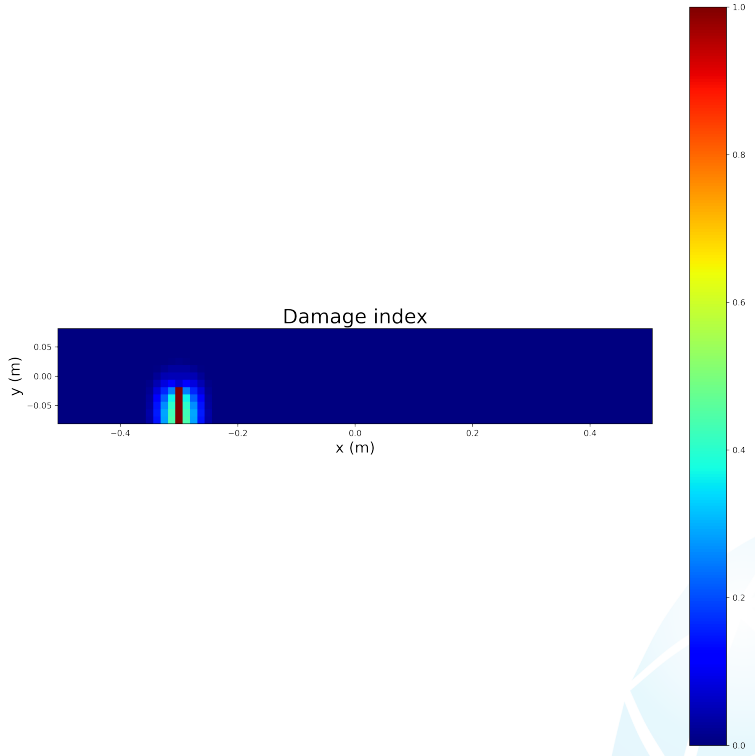
(b) Programa de Patriota (2019)

(c) Programa de Patriota (2019)

Fonte: Elaborado pelo autor.

Temos também o gráfico de índice de dano dado pela figura 4.6. É interessante observar que o dano se dá aonde a trinca foi colocada. Porém, como não foi considerado dano no modelo, a trinca não avançou.

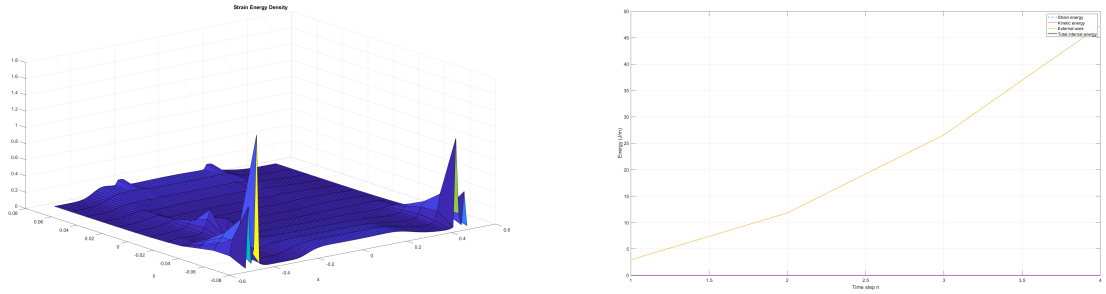
Figura 4.6 – Índice de dano



Fonte: Elaborado pelo autor.

A figura 4.7 mostra no item (a) os picos de energias em locais esperados, sendo basicamente nos locais indicados em vermelho aonde ocorreram grandes deformações e no item (b) a revolução das energias de deformação, cinética, cinemática, trabalho externo e energia interna total. Observamos que o a energia interna é nula devido ao solver *quasi-static*, que considera a equação de movimento como sendo equação (3.10). Há apenas a energia do trabalho externo realizado pelas forças aplicadas no corpo.

Figura 4.7 – Energia

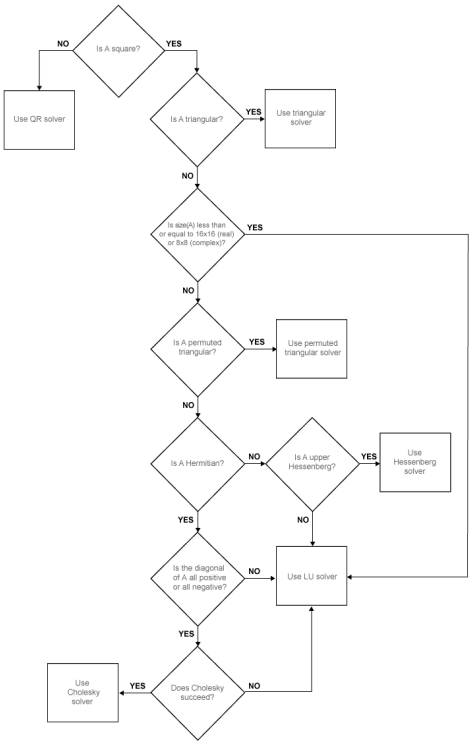


(a) Densidade de energia de deformação (b) Revolução da energia

Fonte: Elaborado pelo autor.

Vemos que o programa consegue replicar com sucesso esta simulação. No entanto, há algumas particularidades quanto à convergência e erros numéricos. Por exemplo, o jeito que o *MATLAB* trata a resolução de uma matriz inversa é bem particular. Segue o seguinte fluxo, conforme a figura 4.8.

Figura 4.8 – Fluxo da função *mldivide*



Fonte: (MLDIVIDE. . . , 2022).

Quanto a erros numéricos, ao que tudo indica, como a biblioteca *numpy* é externa e trabalha com muitas funções em C, muitas vezes é necessário converter o ponto flutuante nativo do *Python*, fazer as operações por fora trazendo o novo número, o que pode causar pequenos erros. Esses erros, se forem muito carregados podem influenciar negativamente nos resultados do código, introduzindo, por exemplos, dificuldades de realizar calcular o inverso de uma matriz pois esta pode tornar-se mal condicionada. Em algumas operações, percebeu-se que havia diferenças significativas nos valores quando comparando o *Python* com o *MATLAB*, realizando operações idênticas. Isto traz algumas dificuldades que devem ser estudadas.

5 CONCLUSÃO E SUGESTÕES PARA FUTUROS TRABALHOS

Trabalhar com a peridinâmica, de forma geral, mostrou-se como sendo um grande desafio, pois ela é uma ciência relativamente nova. Sendo assim, os cientistas espalhados pelo mundo não possuem muita informação a respeito. Um exemplo desta questão seria a dificuldade de encontrar o valor da taxa de liberação de energia G_0.

Outra dificuldade encontrada foi a conversão do código da linguagem *MATLAB* para *Python*. Algumas partes podem se tornar bem complicadas, uma vez que há diferenças fundamentais entre as linguagens, como por exemplo a indexação de vetores se dá de forma diferente, começando com 1 no *MATLAB* e começando com 0 no *Python*. A sintaxe das duas linguagens é muito diferente. Coisas que são muito simples em uma linguagem pode se tornar muito trabalhosa ou difícil em outra. Diferença de algoritmos fundamentais, como é o caso do operador "\" no *MATLAB* (similar a função *mldivide*), quando comparada à suas versões do *Python* as funções *solve* ou *lstsq*, atuam de forma diferente, o que dificultou o cálculo da matriz de rigidez tange K no *solver quasi-static* , entre outros.

Outra dificuldade encontrada foi entender o *software* desenvolvido por *Patriota (2019)*. O *software* peridinâmico, pela própria natureza da peridinâmica, é muito complicado, exigindo uma série de parâmetros e definições, laços avançados, cálculo com vetores de tamanhos que variam a cada interação, concatenações e operações diversas com vetores de múltiplas dimensões.

Tendo isso em vista, o *software* desenvolvido tenta trazer a peridinâmica para as pessoas de forma um pouco mais simples, mais flexível e desenvolvido pensando em compatibilidade entre diferentes sistemas operacionais. Mas também tem as suas desvantagens. O *Python* é conhecido por ser uma linguagem lenta em certas operações, devido ao fato de ser uma linguagem interpretativa. Embora isso venha mudando em compilações recentes, aonde se busca implementar otimizações diversas, como por exemplo, basear mais partes do código na linguagem C.

Fica como sugestão para futuros trabalhos:

- Expandir o código para suportar simulações em 3D;
- Aplicar vetorização em partes do código que utilizam laços, se possível;
- Melhorar modelos existentes e/ou trazer novos modelos;

- Implementar os demais algoritmos de áreas parciais propostos por Seleson (2014);
- Flexibilizar o algoritmo *Mesh*, responsável pela criação das malhas peridinâmicas;
- Simplificar o uso da classe *BoundaryConditions*, para que usuários com dificuldade em programação tenham mais facilidade;
- Adicionar mais verbosidades ao longo do código, simplificando o entendimento de todo o processo de simulação peridinâmica;
- Investigar inconsistências no algoritmo de cálculo da matriz de rigidez tangente analítica do modelo *PMB* quanto às diferenças entre as funções *solve*, *lstsq* do *numpy* em relação ao operador "\" (referente à função *mldivide*) do *MATLAB*;
- Implementar uma interface com habilidades de salvar e carregar configurações e arquivos para facilitar o uso do programa para pessoas com dificuldade em programação. Deve se ter um cuidado para não perder o poder e liberdade que o método tradicional fornece;
- Resolver problemas que podem estar presentes no código ou surgir devido à modificações futuras.

REFERÊNCIAS

BOBARU, Florin et al. **Advanced in Applied Mathematics**: Handbook of Peridynamic Modeling. 1. ed. New York: CRC Press, 2016. ISBN 978-1-4822-3043-7. DOI: 10.1201/9781315373331.

BUTT, Sahir N.; TIMOTHY, Jithender J.; MESCHKE, Günther. Wave dispersion and propagation in state-based peridynamics. **Computational Mechanics**, Springer, Bochum, 2017. DOI: 10.1007/s00466-017-1439-7.

GERSTLE, Walter Herbert. INTRODUCTION TO PRACTICAL PERIDYNAMICS. **Computational Solid Mechanics Without Stress and Strain**, World Scientific Publishing Co. Pte. Ltd., USA, v. 1, 2015. ISSN 2315-4713.

HALE, Jeff. **The Most In-Demand Tech Skills for Data Scientists**. Towards Data Science. 2019. Disponível em: <https://towardsdatascience.com/the-most-in-demand-tech-skills-for-data-scientists-d716d10c191d>. Acesso em: 12 dez. 2022.

MADENCI, Erdogan; OTERKUS, Erkan. **Peridynamic Theory and Its Applications**. London: Springer, 2014. ISBN 978-1-4614-8464-6,978-1-4614-8465-3. DOI: 10.1007/978-1-4614-8465-3.

MLDIVIDE. MathWorks. Disponível em: <https://www.mathworks.com/help/matlab/ref/mldivide.html>. Acesso em: 14 dez. 2022.

MORAES VINHA, André de. **ANÁLISE DO MÉTODO PERIDINÂMICO COMO POTENCIAL FERRAMENTA DE MANUTENÇÃO PREDITIVA**. 2022. Ilha Solteira.

PATRIOTA, Túlio Vinícius Berbert. **Numerical investigation of damage models in two-dimensional peridynamics using MATLAB**. 2019.

SELESON, Pablo. Improved one-point quadrature algorithms for two-dimensional peridynamic models based on analytical calculations. **Computer Methods in Applied Mechanics and Engineering**, Elsevier, Austin, v. 282, 2014. DOI: 10.1016/j.cma.2014.06.016.

SILLING, S.A.; ASKARI, E. A meshfree method based on the peridynamic model of solid mechanics. **Computers & Structures**, Elsevier, v. 83, p. 1526–1535, 2005. DOI: 10.1016/j.compstruc.2004.11.026.

TIOBE Index for November 2022. TIOBE the software quality company. 2022. Disponível em: <https://www.tiobe.com/tiobe-index/>. Acesso em: 2 dez. 2022.

APÊNDICE A – CÓDIGO FONTE DO PROGRAMA DESENVOLVIDO EM *PYTHON*

O código se encontra em `https://github.com/FPChaim/peridynamics`.

Para utilizá-lo, basta clonar o repositório com qualquer ferramenta capaz de clonar repositórios. Por exemplo, pode-se utilizar o aplicativo *GitHub Desktop*, disponível em `https://desktop.github.com/`.

Então, é necessário adicionar a pasta em que o repositório foi baixado ao caminho de busca do seu *IDE* ou editor, de tal forma que a pasta *peridynamics* seja vista por ele. Isso permite a importação do código fonte. Exemplo: *import peridynamics as pd*.

Se necessário, exemplos estão disponíveis em formatos de *notebooks* na pasta *notebooks*.

ANEXO A – CÁLCULOS DE AREAS PARCIAIS E CENTROIDES DE REGIÕES DE INTERSECÇÃO (SELESON, 2014)

A.1 Caso I: quatro cantos de τ_i dentro da vizinhança de i

Área parcial:

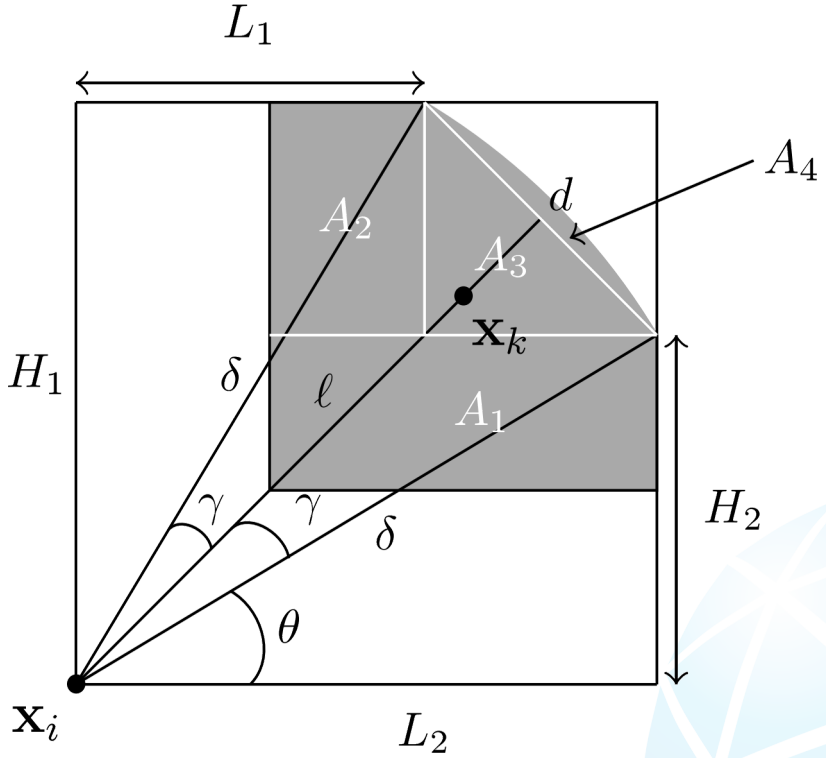
$$A_j^{(i)} = h^2 \tag{A.1}$$

Centroide:

$$\begin{cases} \bar{x}_j^{(i)} = x_j \\ \bar{y}_j^{(i)} = y_j \end{cases} \tag{A.2}$$

A.2 Caso II: três cantos de τ_i dentro da vizinhança de i

Figura A.1 – Geometria do caso II



Fonte: (SELESON, 2014).
Nota: $x_k = x_j$

$$H_1 = y_j + \frac{h}{2} - y_i$$
$$L_2 = x_j + \frac{h}{2} - x_i$$
$$L_1 = \sqrt{\delta^2 - H^2}$$
$$H_2 = \sqrt{\delta^2 - L^2}$$
$$d = \sqrt{(H_1 - H_2)^2 + (L_2 - L_1)^2}$$
$$l = \sqrt{\delta^2 - \left(\frac{d}{2}\right)^2}$$
$$\gamma = \arcsin\left(\frac{d/2}{\delta}\right)$$

Área parcial:

$$A_1 = h(h - (H_1 - H_2))$$
$$A_2 = (h - (L_2 - L_1))(H_1 - H_2)$$
$$A_3 = \frac{1}{2}(L_2 - L_1)(H_1 - H_2)$$
$$A_4 = \gamma\delta^2 - \frac{1}{2}dl$$
$$A_j^{(i)} = \frac{1}{2}(L_2 - L_1)(H_1 - H_2) + \gamma\delta^2 - \frac{1}{2}dl \tag{A.3}$$

Centroide:

$$\bar{x}_1 = x_i + L_2 - \frac{h}{2}$$
$$\bar{y}_1 = y_i + H_2 - \frac{1}{2}(h - (H_1 - H_2))$$
$$\bar{x}_2 = x_i + L_1 - \frac{1}{2}(h - (L_2 - L_1))$$
$$\bar{y}_2 = y_i + H_1 - \frac{1}{2}(H_1 - H_2)$$
$$\bar{x}_3 = x_i + L_2 - \frac{2}{3}(L_2 - L_1)$$
$$\bar{y}_3 = y_i + H_1 - \frac{2}{3}(H_1 - H_2)$$
$$\theta = \arctan\left(\frac{H_2}{L_2}\right)$$

$$\hat{l} = \frac{4\delta \sin^3 \gamma}{3(2\gamma - \sin(2\gamma))}$$

$$\bar{x}_4 = x_i + \hat{l}\cos(\theta + \gamma)$$

$$\bar{y}_4 = y_i + \hat{l}\sin(\theta + \gamma)$$

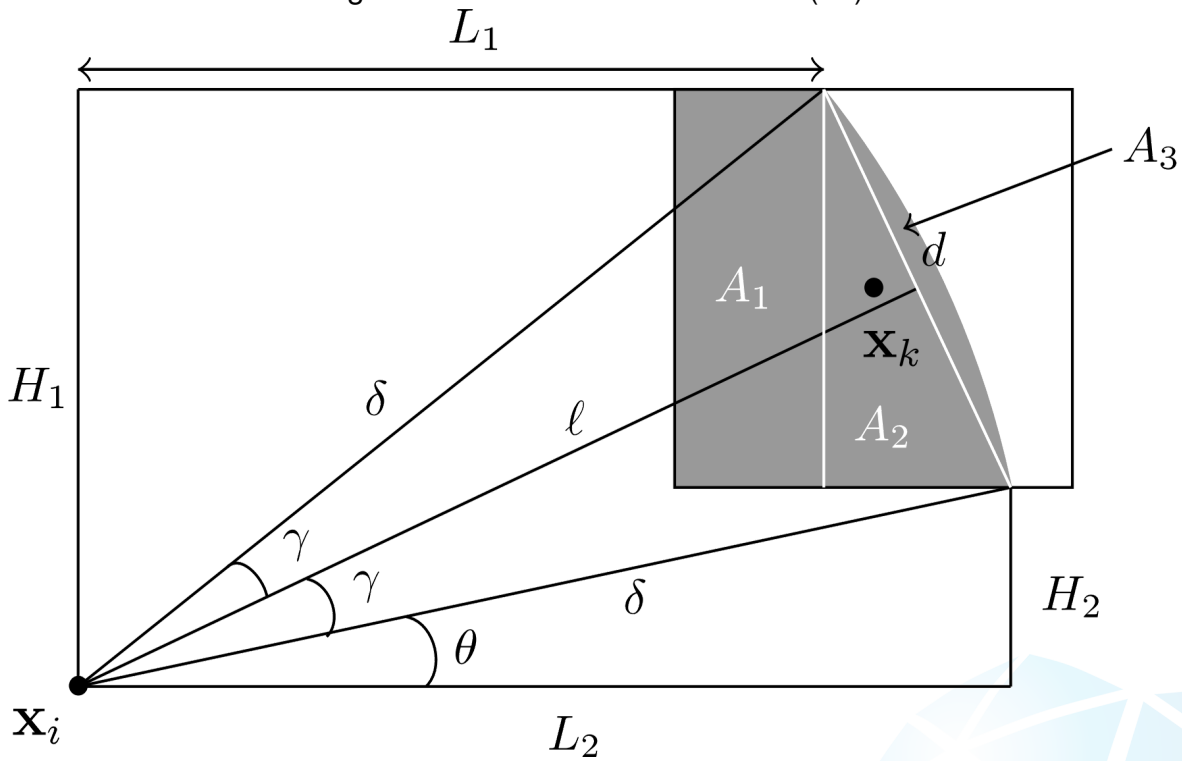
$$\begin{cases} \bar{x}_j^{(i)} = \dfrac{A_1\bar{x}_1 + A_2\bar{x}_2 + A_3\bar{x}_3 + A_4\bar{x}_4}{A_1 + A_2 + A_3 + A_4} \\ \bar{y}_j^{(i)} = \dfrac{A_1\bar{y}_1 + A_2\bar{y}_2 + A_3\bar{y}_3 + A_4\bar{y}_4}{A_1 + A_2 + A_3 + A_4} \end{cases} \tag{A.4}$$

A.3 Caso III(a1): dois cantos de τ_i dentro da vizinhança de i

Condições:

1. $y_j > y_i$;
2. canto inferior direito de τ_j deve estar fora da vizinhança de i.

Figura A.2 – Geometria do caso III(a1)



Fonte: (SELESON, 2014).
Nota: $x_k = x_j$

$$H_1 = y_j + \frac{h}{2} - y_i$$

$$H_2 = y_j = \frac{h}{2} - y_i$$

$$L_1 = \sqrt{\delta^2 - H_1^2}$$

$$L_2 = \sqrt{\delta^2 - H_2^2}$$

$$d = \sqrt{(L_2 - L_1)^2 + h^2}$$

$$l = \sqrt{\delta^2 - \left(\frac{d}{2}\right)^2}$$

$$\gamma = \arcsin\left(\frac{d/2}{\delta}\right)$$

Área parcial:

$$A_1 = \left[L_1 - \left(x_j - \frac{h}{2} - x_i\right)\right] h$$

$$A_2 = \frac{1}{2}(L_2 - L_1)h$$

$$A_3 = \gamma\delta^2 - \frac{1}{2}dl$$

$$A_j^{(i)} = h\left[\frac{L_1 + L_2}{2} - \left(x_j - \frac{h}{2} - x_i\right)\right] + \gamma\delta^2 - \frac{1}{2}dl \tag{A.5}$$

Centroide:

$$\bar{x}_1 = x_i + L_1 - \frac{1}{2}(L_1 - (x_j - \frac{h}{2} - x_i))$$

$$\bar{y}_1 = y_i + H_1 - \frac{H}{2}$$

$$\bar{x}_2 = x_i + L_2 - \frac{2}{3}(L_2 - L_1)$$

$$\bar{y}_2 = y_i + H_1 - \frac{2}{3}h$$

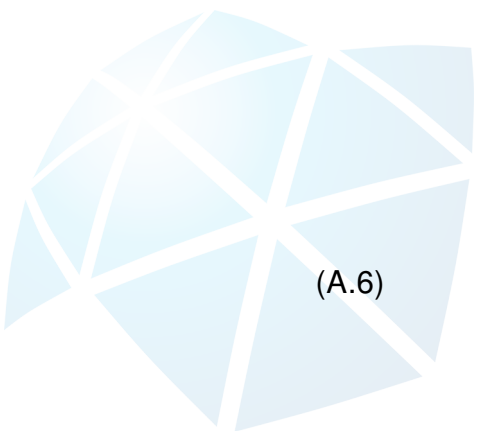
$$\theta = \arctan\left(\frac{H_2}{L_2}\right)$$

$$\bar{l} = \frac{4\delta\sin^3\gamma}{3(2\gamma - \sin(2\gamma))}$$

$$\bar{x}_3 = x_i + \bar{l}\cos(\theta + \gamma)$$

$$\bar{y}_3 = y_i + \bar{l}\sin(\theta + \gamma)$$

$$\begin{cases} \bar{x}_j^{(i)} = \dfrac{A_1\bar{x}_1 + A_2\bar{x}_2 + A_3\bar{x}_3}{A_1 + A_2 + A_3} \\ \bar{y}_j^{(i)} = \dfrac{A_1\bar{y}_1 + A_2\bar{y}_2 + A_3\bar{y}_3}{A_1 + A_2 + A_3} \end{cases} \tag{A.6}$$

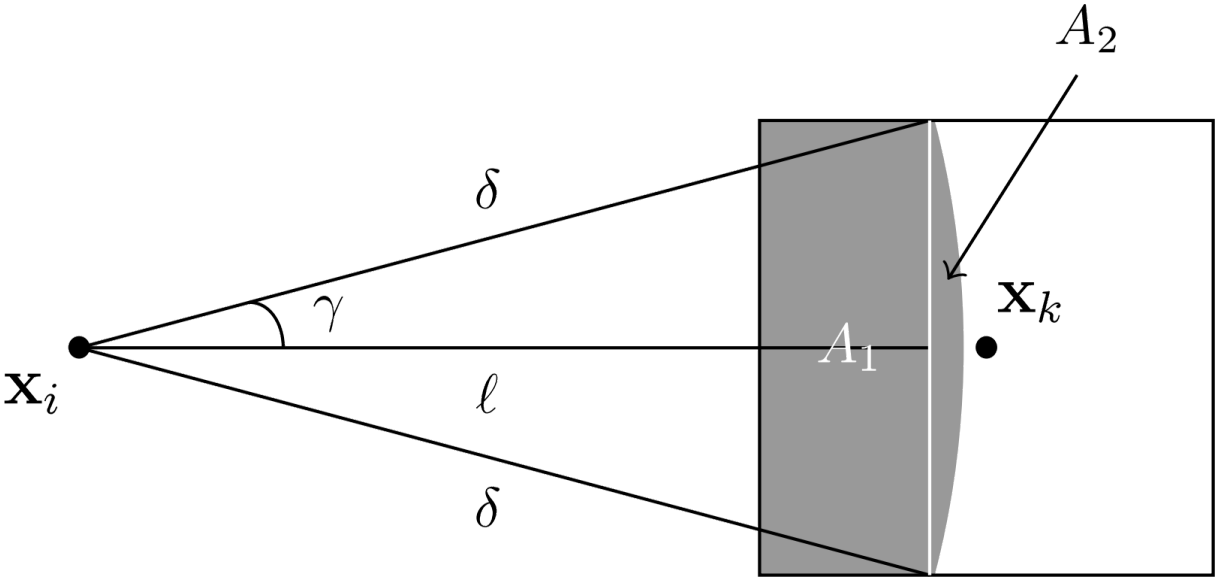


A.4 Caso III(a2): dois cantos de τ_i dentro da vizinhança de i

Condições:

1. $y_j = y_i$;
2. $x_j + \frac{h}{2} \geq x_i + \delta$.

Figura A.3 – Geometria do caso III(a2)



Fonte: (SELESON, 2014).
Nota: $x_k = x_j$

$$l = \sqrt{\delta^2 - \left(\frac{h}{2}\right)}$$

$$\gamma = \arcsin\left(\frac{d/2}{\delta}\right)$$

Área parcial:

$$A_1 = \left[l - \left(x_j - \frac{h}{2} - x_i\right)\right] h$$

$$A_2 = \gamma\delta^2 - \frac{1}{2}hl$$

$$A_j^{(i)} = \left[l - \left(x_j - \frac{h}{2} - x_i\right)\right] h + \left(\gamma\delta^2 - \frac{1}{2}hl\right) \tag{A.7}$$

Centroide:

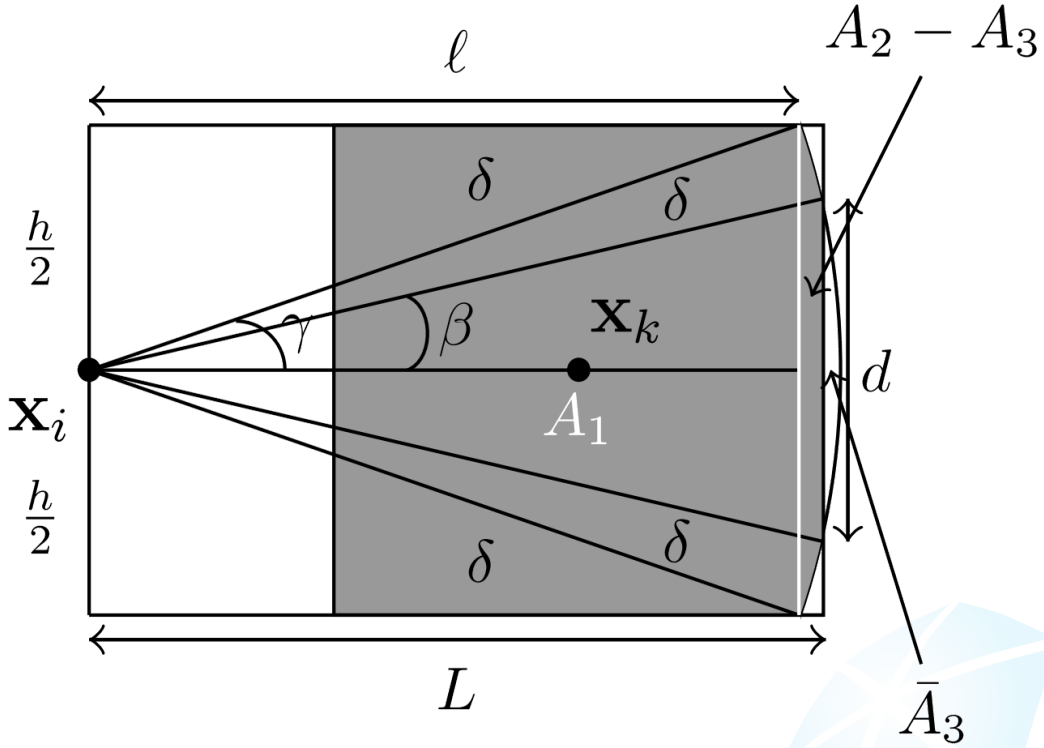
$$\bar{x}_1 = x_i + l - \frac{1}{2}\left(l - \left(x_j - \frac{h}{2} - x_i\right)\right)$$
$$\bar{y}_1 = y_i$$
$$\bar{l} = \frac{4\delta \sin^3 \gamma}{3(2\gamma - \sin(2\gamma))}$$
$$\bar{y}_2 = y_1$$
$$\begin{cases} \bar{x}_j^{(i)} = \dfrac{A_1\bar{x}_1 + A_2\bar{x}_2}{A_1 + A_2} \\ \bar{y}_j^{(i)} = y_i \end{cases} \tag{A.8}$$

A.5 Caso III(b): dois cantos de τ_i dentro da vizinhança de i

Condições:

1. $y_j = y_i$;
2. $x_j + \frac{h}{2} < x_i + \delta$.

Figura A.4 – Geometria do caso III(b)



Fonte: (SELESON, 2014).
Nota: $x_k = x_j$

$$l = \sqrt{\delta^2 - \left(\frac{h}{2}\right)^2}$$

$$L = x_j + \frac{h}{2} - x_i$$

$$\gamma = \arccos\left(\frac{l}{\gamma}\right)$$

$$\beta = \arccos\left(\frac{L}{\delta}\right)$$

$$d = 2\sqrt{\delta^2 - L^2}$$

Área parcial:

$$A_1 = (h - (L - l))h$$

$$A_2 = \gamma\delta^2 - \frac{1}{2}hl$$

$$\bar{A}_3 = \beta\delta^2 - \frac{1}{2}dL$$

$$A_j^{(i)} = h^2 - (L - l)h + \left[\left(\gamma\delta^2 - \frac{1}{2}hl\right) - \left(\beta\delta^2 - \frac{1}{2}dL\right)\right] \tag{A.9}$$

Centroide:

$$\bar{x}_1 = x_i + l - \frac{1}{2}(h - (L - l))$$

$$\bar{y}_1 = y_i$$

$$\hat{l} = \frac{4\delta \sin^3\gamma}{3(2\gamma - \sin(2\gamma))}$$

$$\bar{x}_2 = x_i + \hat{l}$$

$$\bar{y}_2 = y_i$$

$$\hat{l}' = \frac{4\delta \sin^3\beta}{3(2\beta - \sin(2\beta))}$$

$$\bar{x}_3 = x_i + \hat{l}'$$

$$\bar{y}_3 = y_i$$

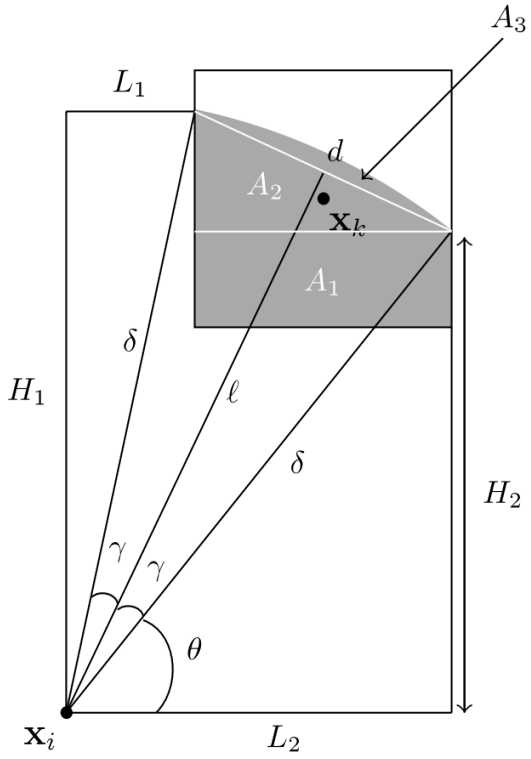
$$\begin{cases} \bar{x}_j^{(i)} = \dfrac{A_1\bar{x}_1 + A_2\bar{x}_2 - \bar{A}_3\bar{x}_3}{A_1 + A_2 - \bar{A}_3} \\ \bar{y}_j^{(i)} = y_i \end{cases} \tag{A.10}$$

A.6 Caso III(c): dois cantos de τ_i dentro da vizinhança de i

Condições:

1. $y_j > y_i$;
2. canto inferior direito de τ_j deve estar dentro da vizinhança de i.

Figura A.5 – Geometria do caso III(c)



Fonte: (SELESON, 2014).
Nota: $x_k = x_j$

$$L_1 = x_j - \frac{h}{2} - x_i$$

$$L_2 = x_j + \frac{h}{2} - x_i$$

$$H_1 = \sqrt{\delta^2 - L_1^2}$$

$$H_2 = \sqrt{\delta^2 - L_2^2}$$

$$d = \sqrt{h^2 + (H_1 - H_2)^2}$$

$$l = \sqrt{\delta^2 - \left(\frac{d}{2}\right)^2}$$

$$\gamma = \arcsin\left(\frac{d/2}{\delta}\right)$$

Área parcial:

$$A_1 = \left(H_2 - \left(y_j - \frac{h}{2} - y_i\right)\right) h$$

$$A_2 = \frac{1}{2}h(H_1 - H_2)$$

$$A_3 = \gamma\delta^2 - \frac{1}{2}dl$$

$$A_j^{(i)} = h\left[\frac{H_1 + H_2}{2} - \left(y_j - \frac{h}{2} - y_i\right)\right] + \gamma\delta^2 - \frac{1}{2}dl \tag{A.11}$$

Centroide:

$$\bar{x}_1 = x_i + L_2 - \frac{h}{2}$$

$$\bar{y}_1 = y_i + H_2 - \frac{1}{2}\left(H_2 - \left(y_j - \frac{h}{2} - y_i\right)\right)$$

$$\bar{x}_2 = x_i + L_1 + \frac{1}{3}h$$

$$\bar{y}_2 = y_i + H_2 + \frac{1}{3}(H_1 - H_2)$$

$$\hat{l} = \frac{4\delta\sin^3\gamma}{3(3\gamma - \sin(2\gamma))}$$

$$\theta = \arctan\left(\frac{H_2}{L_2}\right)$$

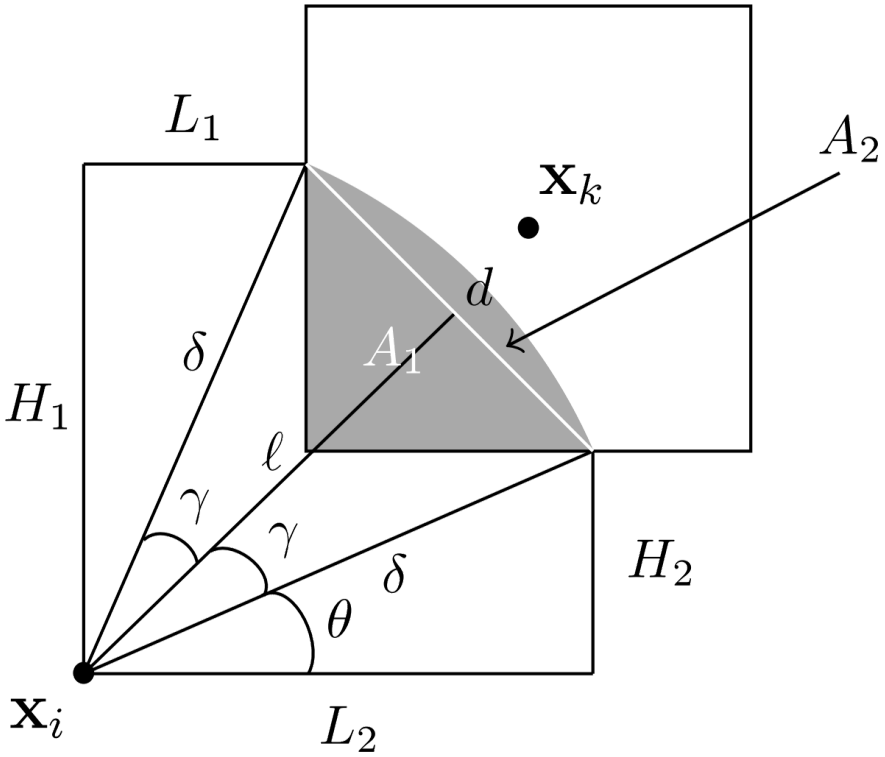
$$\bar{x}_3 = x_i + \hat{l}\cos(\theta + \gamma)$$

$$\bar{y}_3 = y_i + \hat{l}\sin(\theta + \gamma)$$

$$\begin{cases} \bar{x}_j^{(i)} = \dfrac{A_1\bar{x}_1 + A_2\bar{x}_2 + A_3\bar{x}_3}{A_1 + A_2 + A_3} \\ \bar{y}_j^{(i)} = \dfrac{A_1\bar{y}_1 + A_2\bar{y}_2 + A_3\bar{y}_3}{A_1 + A_2 + A_3} \end{cases} \tag{A.12}$$

A.7 Caso IV: um canto de τ_i dentro da vizinhança de i

Figura A.6 – Geometria do caso IV



Fonte: (SELESON, 2014).
Nota: $x_k = x_j$

$$L_1 = x_j - \frac{h}{2} - x_i$$

$$H_2 = y_j - \frac{h}{2} - y_i$$

$$H_1 = \sqrt{\delta^2 - L_1^2}$$

$$L_2 = \sqrt{\delta^2 - H_2^2}$$

$$d = \sqrt{(L_2 - L_2)^2 + (H_1 - H_2)^2}$$

$$l = \sqrt{\delta^2 - \left(\frac{d}{2}\right)^2}$$

$$\gamma = \arcsin\left(\frac{d/2}{\delta}\right)$$

Área parcial:

$$A_1 = \frac{1}{2}(L_1 - L_2)(H_1 - H_2)$$

$$A_2 = \gamma\delta^2 - \frac{1}{2}dl$$

$$A_j^{(i)} = \frac{1}{2}(L_2 - L_2)(H_2 - H_2) + \gamma\delta^2 - \frac{1}{2}dl \tag{A.13}$$

Centroide:

$$\bar{x}_1 = x_i + L_1 + \frac{1}{3}(L_2 - L_1)$$

$$\bar{y}_1 = y_i + H_2 + \frac{1}{3}(H_1 - H_2)$$

$$\hat{l} = \frac{4\delta \sin^3\gamma}{3(2\gamma - \sin(2\gamma))}$$

$$\theta = \arctan\left(\frac{H_2}{L_2}\right)$$

$$\bar{x}_2 = x_i + \hat{l}\cos(\theta + \gamma)$$

$$\bar{y}_2 = y_i + \hat{l}\sin(\theta + \gamma)$$

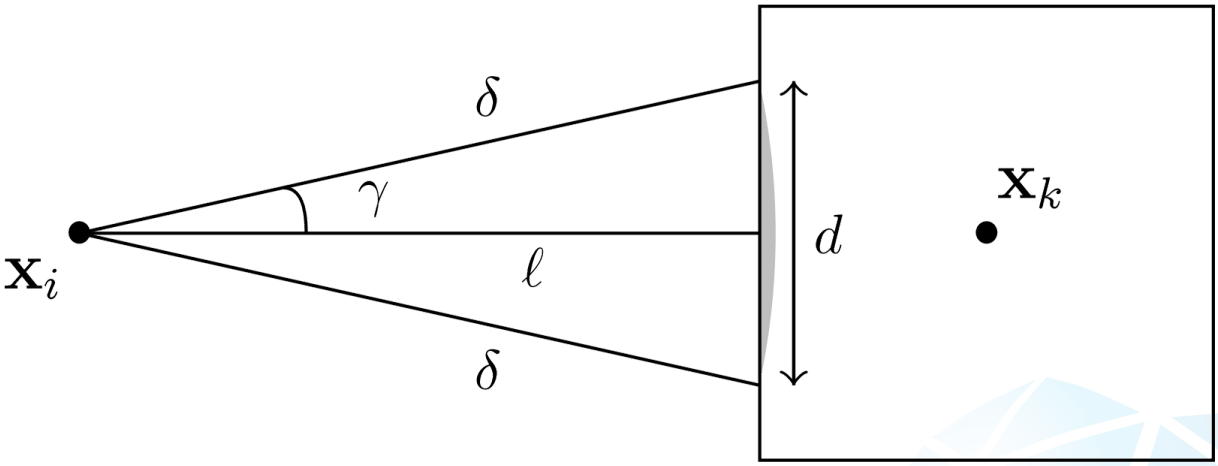
$$\begin{cases} \bar{x}_j^{(i)} = \dfrac{A_1\bar{x}_1 + A_2\bar{x}_2}{A_1 + A_2} \\ \bar{y}_j^{(i)} = \dfrac{A_1\bar{y}_1 + A_2\bar{y}_2}{A_1 + A_2} \end{cases} \tag{A.14}$$

A.8 Caso V: nenhum canto de τ_i dentro da vizinhança de i

Condição:

1. $x_j - \dfrac{h}{2} < x_i + \delta$.

Figura A.7 – Geometria do caso V



Fonte: (SELESON, 2014).
Nota: $x_k = x_j$

$$l = x_j - \frac{h}{2} - x_i$$

$$d = 2\sqrt{\delta^2 - l^2}$$

$$\gamma = \arccos\left(\frac{l}{\gamma}\right)$$

Área parcial:

$$A_j^{(i)} = \gamma\delta^2 - \frac{1}{2}dl \tag{A.15}$$

Centroide:

$$\bar{l} = \frac{4\delta \sin^3 \gamma}{3(2\gamma - \sin(2\gamma))}$$

$$\begin{cases} \bar{x}_j^{(i)} = x_i + \hat{l} \\ \bar{y}_j^{(i)} = y_i \end{cases} \tag{A.16}$$