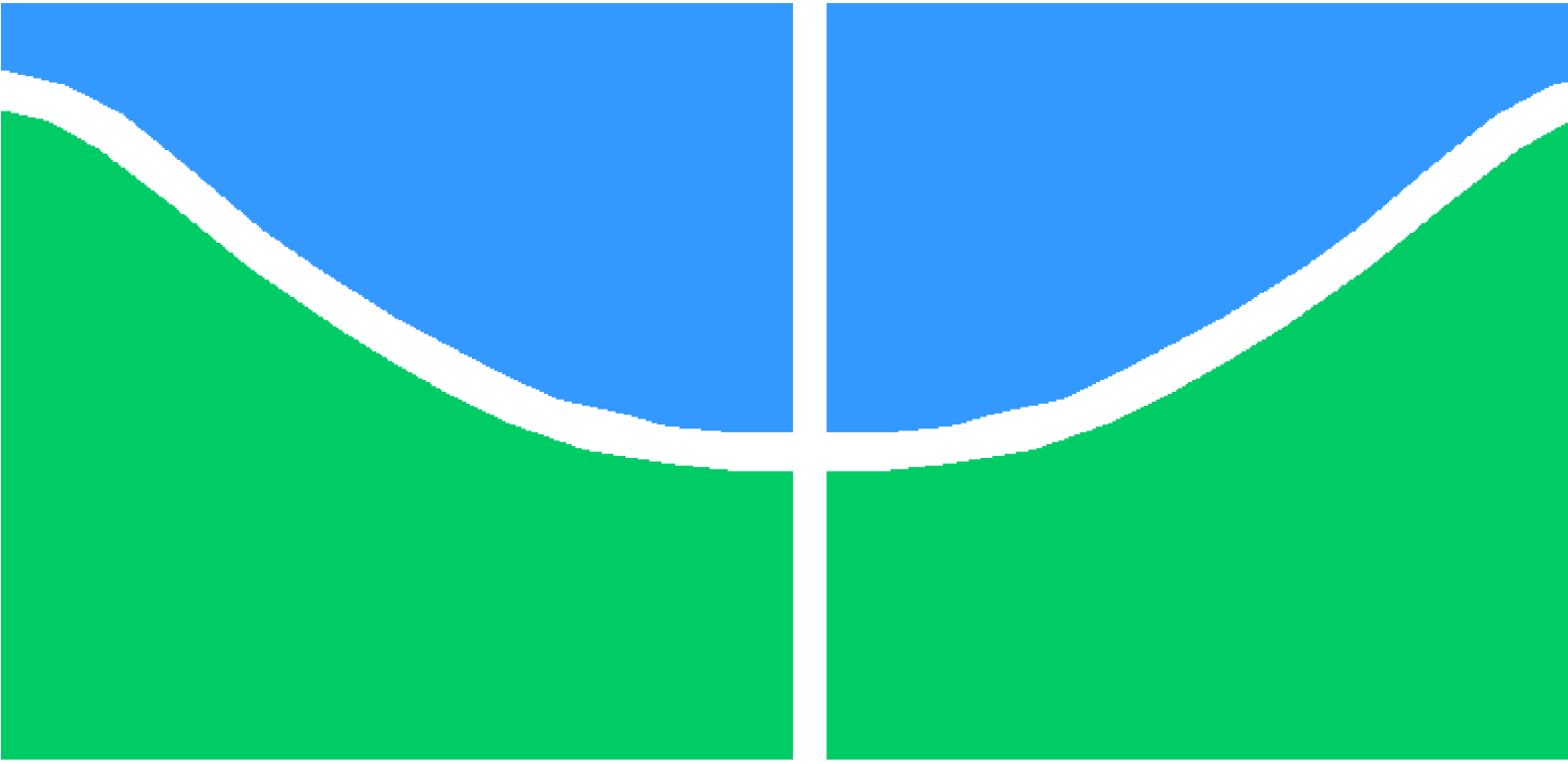


PROJETO DE GRADUAÇÃO

PROGRAMAÇÃO PARALELA DO MÉTODO DOS ELEMENTOS DE CONTORNO

Por,

Inácio Miura Junior

Brasília, 18 de dezembro de 2020

UNIVERSIDADE DE BRASÍLIA

FACULDADE DE TECNOLOGIA

DEPARTAMENTO DE ENGENHARIA MECÂNICA

UNIVERSIDADE DE BRASÍLIA
Faculdade de Tecnologia
Departamento de Engenharia Mecânica

PROJETO DE GRADUAÇÃO

PROGRAMAÇÃO PARALELA DO MÉTODO DOS ELEMENTOS DE CONTORNO

Por,

Inácio Miura Junior

Projeto submetido como requisito parcial para obtenção
do grau de Engenheiro Mecânico

Banca Examinadora

Prof. Éder Lima de Albuquerque, UnB/ENM (Orientador) _____

Prof. Taygoara Felamingo de Oliveira , UnB/ENM _____

Prof. Alvaro Campos Ferreira. , UnB/ENM _____

Brasília 18 de dezembro de 2020

Resumo

O presente trabalho explora técnicas de programação de alto desempenho para aperfeiçoar um código de método dos elementos de contorno (MEC). Serão discutidas as vantagens da implementação da programação em paralelo, passando por uma revisão teórica sobre vetorização, *multi-threading* e programação paralela em *clusters*. A implementação do MEC foi escrita em Python devido a facilidade de programação que reduz o tempo gasto na etapa de desenvolvimento. Serão avaliadas as partes que mais consomem tempo de processamento para uma otimização utilizando a biblioteca Cython, que transforma trechos do programa em extensões em C. Foi explorado o uso de Softwares livres, Open Source, para soluções de geração de CAD (FreeCad, Designspark Mechanical), geração de malha (GMSH), leitura e conversão de malha (Meshio), visualização e pós-processamento (ParaView). Por fim, foi analisado o impacto da implementação de paralelismo, com memória compartilhada, em um dos trechos críticos da rotina de cálculo e foram discutidos os aspectos práticos e dificuldades técnicas, deste método, com o intuito de fomentar o uso de novas práticas mais eficientes de programação. A paralelização da rotina de integração, existente no código, aumentou a velocidade por um fator igual ao número de processadores utilizados.

Palavras-chaves: Programação em paralelo, Elementos de contorno, SIMD, OpenMP, MPI, Python.

Abstract

The present work explore high performance programming techniques to enhance a boundary element method (BEM) code. It was discussed the advantages of implementing parallel programming, throughout the concept of vectorization, multi-threading and parallel programming on clusters. The BEM implementation has been written in Python due to the ease of programming that reduces the time spent in the development stage. The most time consuming parts will be evaluated for a optimization utilizing the Cython library, that turns program snippets into C extensions. Free Open Source Softwares will be explored for solutions of CAD generation (FreeCad, Designspark Mechanical), Mesh generation (GMSH), Mesh reading and conversion (Meshio), visualization and post processing (ParaView).Finally, it was analised the impact of a parallel implementation, with shared memory, on a critical section of the calculation routine and it was discussed the practical aspects and technical dificulties of such method, with the intention of fomenting the use of new and more efficient programming practices. The paralelization of the integration routine, that exist in the code, enhanced the speed by a factor equal to the number of processors utilized.

Key-words: Parallel programming, Boundary elements, SIMD, OpenMP, MPI, Python.

Lista de Figuras

Lista de abreviaturas e siglas

BEM	*Boundary Element Method*
CPU	*Central Processing Unit*
MPI	*Message Passing Interface*
OpenMP	*Open Multi-Processing*
SIMD	*Single Instruction Multiple Data*
SMT	*Simultaneous MultiThreading*

Lista de símbolos

δ	Função Delta de Dirac
δ_{ij}	Operador Delta de Kronecker
Ω	Domínio omega
Γ	Contorno gamma

Sumário

1 INTRODUÇÃO

1.1 Desenvolvimento dos Processadores

Em 1965 o co-fundador da Intel, Gordon Moore, fez uma previsão, revisada em 1975, de que a quantidade de transistores nos processadores dobraria a cada 2 anos sem para isso aumentar o seu custo. Esta previsão ficou conhecida como Lei de Moore e tem se mantido verdadeira desde então. O avanço tecnológico na manufatura de transistores foi responsável por grande parte do desenvolvimento dos processadores por permitir um aumento no número de ciclos capazes de serem executados por unidade de tempo. No entanto, a geração de calor em uma CPU é proporcional a frequência ao cubo (HAGER; WELLEIN, 2010) e embora a Lei de Moore continue válida os nossos processadores atingiram a barreira de calor i.e., a partir de uma certa velocidade de *clock* a quantidade de calor produzida por sua operação não pode mais ser dissipada por métodos convencionais de resfriamento.

A necessidade de aumento da performance dos processadores, a despeito das limitações introduzidas pela geração de calor, motivou os fabricantes a buscarem uma saída para este dilema através do design de processadores com múltiplos núcleos. Esta solução reduz os requisitos tecnológicos pois torna linear a relação entre capacidade de processamento e geração de calor mas, ao mesmo tempo, introduz novos desafios pois um código não otimizado continuará utilizando apenas um núcleo.

O aumento da complexidade da arquitetura dos novos processadores motivou o desenvolvimento de técnicas de Computação de Alta Performance (High Performance Computing ou HPC) que objetivam tornar a execução de códigos mais rápida e eficiente possível com a aplicação de conceitos como a Programação em Paralelo que é, em linhas gerais, um conjunto de instruções que permitem que partes distintas do código sejam executadas concomitantemente. A paralelização é em geral uma solução *Ad Hoc* até o presente momento, o que torna necessário a capacitação dos desenvolvedores para fazer um uso pleno dos processadores modernos. Existem estudos sobre paralelização automática de códigos através de compiladores mas sem resultados significativos. Este assunto é considerado o Santo Graal da programação em paralelo pois permitiria alta performance na execução de códigos escritos de forma ingênua.

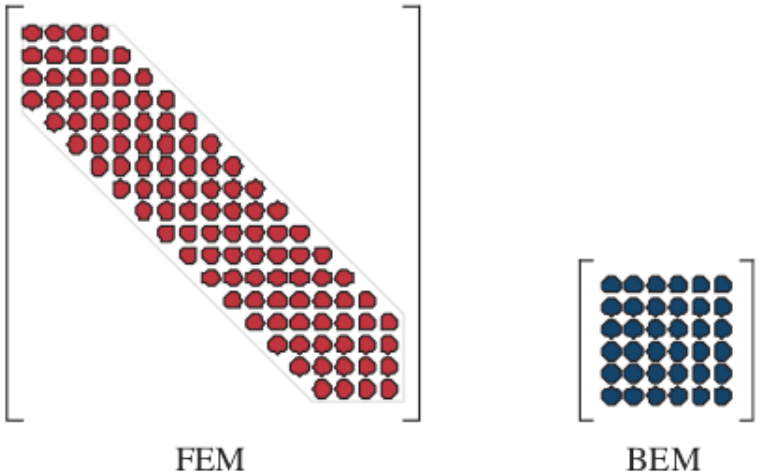
1.2 Método dos Elementos de Contorno

O método dos elementos de contorno (MET), mais amplamente conhecido como *Boundary Elements Method* (BEM), é uma técnica de análise comportamental de sistemas mecânicos sujeitos a fontes externas de perturbação que provocam uma resposta que se traduz em uma alteração no campo da propriedade em questão. Algumas das vantagens no uso do BEM, descritas por Katsikadelis (2016), são:

1. Facilidade na discretização que só precisa ser feita no contorno dos elementos.
2. O método é particularmente eficiente para problemas de condição de Neumann (e.g. fluxos, tensões, deformações, momentos) e pode lidar facilmente com problemas de condição de Dirichlet dentro ou no contorno do domínio utilizando a função delta de dirac.
3. A solução pode ser avaliada em qualquer ponto do domínio a qualquer instante de tempo, pois o método usa uma representação integral da solução por meio de uma expressão matemática contínua, que pode ser diferenciada e utilizada como uma formula matemática. Isto é impossível para métodos como o dos elementos finitos, pois a solução é obtida apenas nos nós.
4. O método é adequado para a solução de problemas em domínios com peculiaridades geométricas como trincas.

Por outro lado, a aplicação do BEM possui severas restrições pois sua aplicação requer a representação do fenômeno de forma integral. Isto só é possível se a solução fundamental puder ser estabelecida. Por último, o método gera sistemas lineares de equações algébricas com matrizes cheias e não-simétricas, mas esta desvantagem é compensada pelo tamanho das matrizes que em geral são menores conforme ilustrado na figura 1.

Figura 1 – Matrizes de Coeficientes FEM vs BEM



Fonte: Katsikadelis, 2016, p.5.

2 Paralelismo

2.1 Processos e *Threads*

Primeiramente é necessário estabelecer a diferença entre programas, processos e *threads*. Um programa é um código armazenado no computador, em algum arquivo, com o objetivo de cumprir uma determinada tarefa específica. Ao ser executado, este arquivo é compilado em uma linguagem binária (linguagem de baixo nível) em um arquivo que pode então ser interpretado pela CPU.

Um processo é um programa em sua forma binária carregado na memória juntamente com os recursos necessários para a sua execução. Os recursos essenciais são os endereços dos dados, instruções e ordem de execução. A alocação de recursos é feita pelo sistema operacional que gerencia todos os processos estruturando os dados em pilhas (*Stacks* e *Queues*) ou por alocação dinâmica (*Heap*). Um programa pode ser executado em múltiplos processos e cada um tem seu respectivo espaço na memória, o que significa que sua execução acontece de forma independente e isolada. Um processo não pode acessar diretamente os dados de outro processo.

Uma *thread* é uma unidade de execução dentro de um processo. Um processo pode ter uma única *thread*, neste caso o processo e a *thread* são a mesma coisa, ou ter múltiplas *threads*. A diferença fundamental é que as *threads* compartilham todos os recursos mas cada uma tem seu registro, *i.e.* memória temporária, e opera sobre sua respectiva pilha de tarefas.

Uma forma prática de compreender a diferença entre estes conceitos é observando o navegador de internet Google Chrome. O navegador é o programa que tem o objetivo de acessar a internet. Cada janela aberta é um processo e como dito anteriormente cada processo é independente. Cada aba em uma janela é uma *thread* que compartilha recursos com as outras, assim, logando em um site em uma aba significa logar em todas as abas mas o mesmo não se observa quando se troca de janela. O paralelismo com compartilhamento de recursos é chamado de "paralelismo de *thread*" enquanto o recíproco é chamado de "paralelismo de processo".

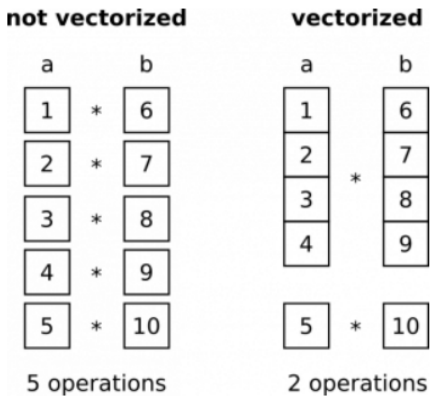
2.2 Conceito de Paralelismo

O conceito de paralelismo é amplo e diz respeito a alocação de memória e ordenamento da execução das tarefas na CPU de forma a otimizar seu desempenho, podendo se referir desde o nível mais fundamental da arquitetura dos processadores (nível de bit) até a execução de tarefas em vários nós simultaneamente, como é o caso dos *clusters*. A ideia geral é que o trânsito de dados entre a memória e a CPU é mais lento do que a capacidade de processamento destes dados, o objetivo é então manter a unidade aritmética lógica constantemente alimentada com dados pela memória e instruções da unidade de controle pra aumentar a saída de resultados. O presente trabalho se restringe a tratar das formas de paralelismo conhecidas como *Single Instruction Multiple Data* (SIMD), *Simultaneous MultiThreading* (SMT), programação paralela com memória compartilhada (OpenMP) e *Message Passing Interface* (MPI).

2.3 Single Intruction Multiple Data (SIMD)

A estratégia neste caso é maximizar a saída da CPU agrupando múltiplos dados em vetores e aplicando uma única instrução para todo o vetor. O exemplo da figura 2 demonstra como o processamento de dados agrupados pode se tornar mais eficiente; para a mesma carga de trabalho, o caso vetorizado reduz o número de operações necessárias de 5 para 2.

Figura 2 – Multiplicação Paralelizada VS Não paralelizada



Fonte: Konrad (2018), https://datascience.blog.wzb.eu/2018/02/02/vectorization-and-parallelization-in-python-with-numpy-and-pandas/

Uma simples tarefa de subtração de vetores pode se tornar, pelo menos, dezenas de vezes mais rápida com um comando vetorizado, como evidenciado no exemplo da figura 3 na qual compara-se o tempo de execução utilizando um *loop for* e um comando *slice* em Python. O comando vetorizado foi 10 vezes mais rápido conforme as observações no código.

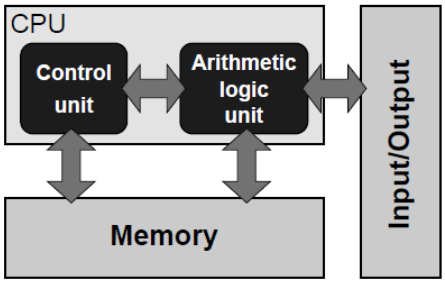
Figura 3 – Exemplo de Vetorização

```
# impacto da vetorização
import timeit

print(timeit.timeit('''import numpy as np
arr = np.arange(1000)
dif = np.zeros(999, int)

for i in range(1, len(arr)):
    dif[i-1] = arr[i] - arr[i-1]''', number=50))
#0 processador levou 0,02257s para realizar esta tarefa.

print(timeit.timeit('''import numpy as np
arr = np.arange(1000)
dif = np.zeros(999, int)
dif = arr[1:] - arr[:-1]''', number=50))
#0 processador levou 0,00214s para realizar esta tarefa.
```

2.4 Simultaneous Multithreading (SMT)

A figura 4 apresenta um esquema conceitual do funcionamento de uma CPU elementar que executa uma linha de comando (*Thread*) de cada vez.

Figura 4 – Diagrama esquemático simplificado de uma CPU



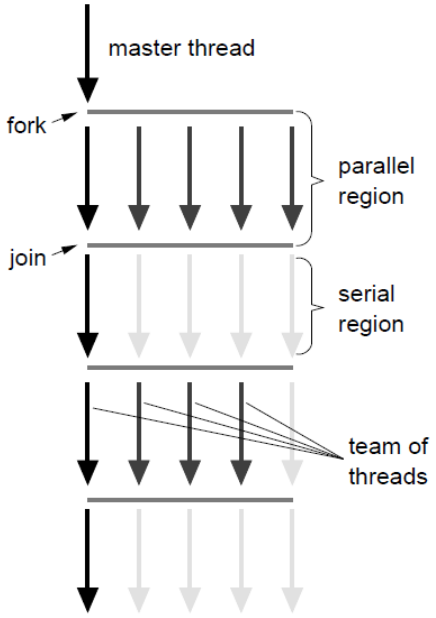
Fonte: Hager & Wellein, 2010, p.2.

Aumentando uma unidade de processamento lógico, é possível executar duas linhas de comando em uma mesma CPU com um incremento de tamanho físico desprezível. No caso dos processadores Xeon da Intel este incremento é de 5%. Esta estratégia leva a ganhos de até 30% de capacidade de processamento em uma mesma CPU, mas como todos os recursos são divididos entre as unidades lógicas, não existe ganho significativo ao introduzir mais do que dois desses componentes (RAUBER; RÜNGER, 2010). Isto dá a característica conhecida como *Multithreading* para um único núcleo que passa a atuar como múltiplos processadores. A Intel utiliza esta tecnologia sob o nome Hyper-Threading e a AMD sob o nome Cluster Multi Threading (CMT). Existem alguns casos extremos como o processador Xeon-Phi da Intel que utiliza instrução *Multi Thread* de 4 vias para cada núcleo físico.

2.5 Open Multi-Processing (OpenMP)

O compartilhamento de memória entre múltiplos processadores possibilita o acesso imediato de dados sem necessidade de comunicação explícita. Um esforço conjunto foi feito pelos vendedores de compiladores para criar um conjunto de normas para esta área que ficou conhecido como OpenMP (HAGER; WELLEIN, 2010). Nem todos os processos são paralelizáveis, algumas instruções são únicas e outras possuem interdependência entre iterações. Um programa OpenMP divide o código em regiões paralelas e seriais, i.e., regiões de linhas de comando únicas (*Single Thread*) e múltiplas (*MultiThread*).

Figura 5 – Fluxograma de uma *thread* OpenMP



Fonte: Hager & Wellein, 2010, p.144.

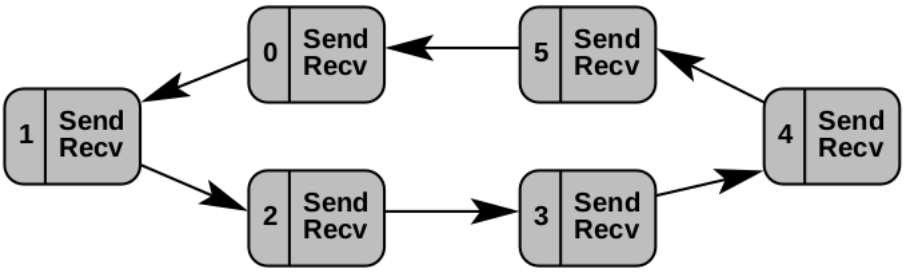
A figura 5 demonstra como um processo paralelizado pode tirar proveito de múltiplos núcleos em sua região de *multithread*. Nas regiões paralelas, cada *thread* é executada por uma CPU diferente, tornando mais rápido o processamento destes trechos.

2.6 Message Passing Interface (MPI)

A programação em paralelo para múltiplos processadores sem compartilhamento de memória torna necessário que a comunicação do acesso de dados seja feita de forma explícita. A comunicação entre processos é tida como a forma mais trabalhosa de paralelização mas também a mais flexível. Um esforço conjunto foi feito pelos vendedores de compiladores para criar um conjunto de normas para esta área, que ficou conhecido como MPI (HAGER; WELLEIN, 2010). Esta forma de paralelização, comumente utilizada em *Clusters*, se baseia no modelo *Single Program Multiple Data* (SPMD).

Existem duas formas de transferir informações entre diferentes processadores. A primeira é a estratégia do ”anel” (*Ring Shift*) onde um processador envia informações apenas para o próximo nó e recebe informações apenas do nó anterior.

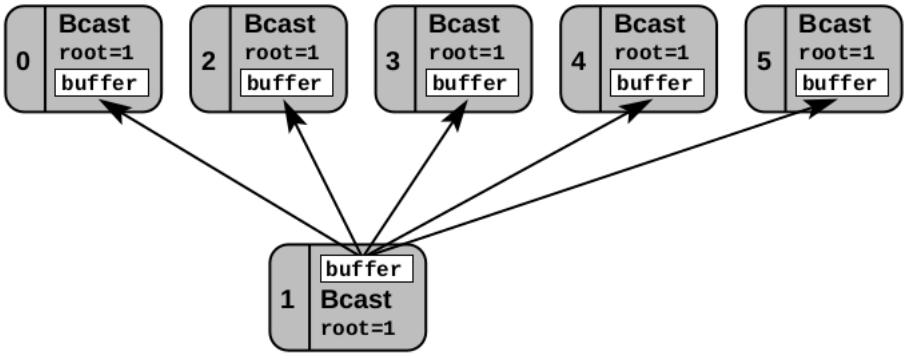
Figura 6 – MPI - Comunicação *ring shift*



Fonte: Hager & Wellein, 2010, p.211.

A segunda estratégia é utilizar um processador para controlar o processo raiz, redistribuindo os processos e informações para os outros processadores.

Figura 7 – MPI - Mestre/Trabalhador



Fonte: Hager & Wellein, 2010, p.214.

3 HPC em Python

3.1 Computação de Alta Performance

A computação científica tem se tornado o ”terceiro pilar” do método científico moderno, ao lado da teoria e experimentação, isto porque a ciência se expandiu a limites que não são práticos ou sequer possíveis de se experimentar (SKUSE, 2019). A computação tornou-se mais do que uma simples ferramenta da ciência, de modo que a modelagem computacional das formulações teóricas é um fator crítico para avanços científicos e tecnológicos.

A crescente demanda por volume de processamento de dados levou ao desenvolvimento de técnicas de Computação de Alta Performance (*High Performance Computing ou HPC*). A evolução do HPC está intimamente ligada com a evolução dos processadores, pois a estrutura dos dados é otimizada para a sua arquitetura, tornando necessário aos programadores não apenas um método numérico eficiente como também um conhecimento básico dos conceitos do Hardware e Software que formam o ”ecossistema” do código (HAGER; WELLEIN, 2010).

Um exemplo prático de aplicação de HPC é o uso otimizado dos *arrays* temporários na solução de expressões. Considerando a situação da figura 8, um vetor temporário é utilizado para armazenar o valor de 2,0*a e outro vetor temporário armazena o valor de 4,5*b.

Figura 8 – Exemplo - Vetor Temporário (Parte 1)

```
import numpy
a = numpy.random.random((1024, 1024, 50))
b = numpy.random.random((1024, 1024, 50))

c = 2.0 * a - 4.5 * b
```

Uma pequena modificação desta expressão pode ser vista na figura 9, neste caso novamente são criados vetores temporários para 2,0*a e 4,5*b, em seguida são criados vetores temporários para sen(a) e cos(b) que reaproveitam os vetores temporários anteriores.

Figura 9 – Exemplo - Vetor Temporário (Parte 2)

```
c = 2.0 * a - 4.5 * b + numpy.sin(a) + numpy.cos(b)
```

Utilizando parenteses desnecessários, algumas operações podem ficar mais fáceis de se interpretar por seres humanos mas a um alto custo computacional. Na figura 10 o parêntese extra força o computador a armazenar o vetor temporário sen(a) + cos(b) sem reaproveitar os vetores anteriores. Quando se trabalha com vetores particularmente extensos, aos desatentos, é fácil esbarrar em problemas de falta de memória, além de reduzir drasticamente a performance do processador para executar a operação.

Figura 10 – Exemplo - Vetor Temporário (Parte 3)

```
c = 2.0 * a - 4.5 * b + (numpy.sin(a) + numpy.cos(b))
```

Uma boa prática de programação neste caso é demonstrada na figura 11 em que a expressão é dividida em partes menores que reduzem a quantidade de dados armazenados na memória temporária da CPU.

Figura 11 – Exemplo - Vetor Temporário (Parte 4)

```
c = 2.0 * a
c -= 4.5 * b
c += np.sin(a)
c += np.cos(b)
```

Um algoritmo otimizado reduz o custo computacional na ordem de sua eficiência, *i.e.*, se um código roda duas vezes mais rápido é como se o usuário estivesse usando dois processadores. A diferença entre um código otimizado pode chegar na casa dos milhares. É importante ressaltar que a maior parte do tempo de processamento é gasto em partes críticas do processo conhecidas como *"HotSpots"*. Então, não é necessário a otimização do programa do início ao fim, pois isto impacta negativamente o tempo de projeto, como é dito no jargão criado por Donald Knuth: A otimização prematura é a raiz de todo o mal.

3.2 Desvantagens e Vantagens de Python

Um resultado chave apresentado pela pesquisa do ano de 2019 da comunidade Stack Overflow (Developer Survey Results 2019. StackOverflow (2019). Disponível em: https://insights.stackoverflow.com/survey/2019#methodology) coloca o Python como a principal linguagem de programação com a maior ascensão em uso, sendo também a segunda favorita dos usuários. A razão para isto é a simplicidade para os iniciantes e

flexibilidade para os mais experientes, mas estas facilidades possuem um custo de performance. Existem 3 razões principais que explicam porque um código em Python costuma ser lento:

1. A Flexibilidade:

   Um exemplo disso é o caso da criação de vetores. Quando se cria um vetor, a CPU precisa alocar um espaço na memória com elementos consecutivos para armazenar o objeto que deve ser formado por um conjunto de informações do mesmo tipo; as "listas" do Python são estruturas mais versáteis que permitem o armazenamento de objetos constituídos por dados de diferentes tipos em um espaço de memória física não contíguo. Isto acontece porque estas listas são vetores com "endereços" que apontam para o local de armazenamento dos dados reais, o que torna a alocação mais dinâmica. Esta funcionalidade de Python torna o processo de programação mais rápido e intuitivo em troca de uma redução significativa no desempenho.

2. Python é uma linguagem interpretada:

   Linguagens compiladas como C, C++ e Fortran tem seus códigos fonte transformados diretamente em linguagem de máquina tornando-as muito ágeis. Linguagens interpretadas como Java e Python são mais "amigáveis" para os programadores mas necessitam ser traduzidas para uma linguagem intermediária e só depois transformadas em linguagem de máquina, incorrendo em um custo de eficiência pela etapa extra no processamento. No caso da linguagem interpretada, a tradução desta em linguagem intermediária é considerada o código fonte.

3. Global Interpreter Lock:

   Python não é uma linguagem segura contra *Multi-threading* pois a natureza dinâmica de sua programação gera conflitos durante a checagem pré-processamento, que ocorre quando duas ou mais *threads* compartilham a mesma memória. Para contornar este problema utiliza-se o protocolo Global Interpreter Lock (GIL), que proíbe o uso de *Multi-Thread* e desta forma limita a viabilidade do uso para computação científica do Python nativo.

As razões geralmente apontadas como os pontos forte da linguagem são:

1. Bibliotecas:

   Existe um grande número de bibliotecas *Open Source* disponíveis na internet. Assim, o programador pode se focar muito mais na aplicação da lógica da programação e menos em implementação de funções.

2. Comunidade:

   Possui uma das maiores comunidades de programadores que existe, facilitando a troca de experiência entre usuários, o processo de aprendizado e o desenvolvimento

colaborativo.

3. Simplicidade:

   Esta é a principal razão pela qual os iniciantes em programação escolhem Python, por apresentar uma estrutura organizada, elegante, limpa e de fácil compreensão.

4. Versatilidade:

   Com tantas limitações de desempenho, parece controverso pensar que esta linguagem possa ser utilizada em aplicações científicas. Porém é possível trocar a flexibilidade por performance e obter resultados tão bons quanto C++ ou Fortran utilizando um compilador estático como o Cython ou até mesmo os conceitos de compilação *Just In Time* (JIT) e *Ahead Of Time* (AOT) com o Numba.

3.3 Estratégia de Programação

Analisando os prós e contras de Python, observa-se que o processo de desenvolvimento do código é mais rápido pois a linguagem é mais intuitiva e flexível. Mas ao mesmo tempo, o software é mais lento. Portanto é justo afirmar que, em sua forma nativa, Python não é uma linguagem adequada para aplicações científicas envolvendo uso intensivo de capacidade de processamento.

O Princípio de Pareto é frequentemente mencionado na área de otimização de software onde é conhecida de uma forma diferente como a regra 90/10, i.e., 90% do tempo é gasto em 10% do código fonte. Uma boa estratégia neste caso é utilizar o Python nativo para as partes não críticas e trechos migrados em C++ ou Fortran para os *HotSpots* (CAI; LANGTANGEN; MOE, 2005), esta é uma solução conveniente e eficiente para aplicações científicas pois equilibra o gasto de recursos computacionais e humanos de acordo com a tarefa.

3.4 Cython

Cython é um compilador que transforma um código Python em um código C. De forma prática, é uma extensão da linguagem amplamente utilizada em aplicações científicas. As vantagens da utilização de Cython são demonstradas com o uso de um material de referência de uso livre, disponibilizado pela *Partnership for Advanced Computing in Europe* (PRACE) no curso "Python in High Performance Computing". A solução numérica para a equação da condução de calor que é dada pela equação 3.1.

$$\frac{\partial u}{\partial t} = \alpha \nabla^2 u \tag{3.1}$$

Onde $u = u(x, y, t)$ representa o campo de temperaturas e α é o coeficiente de difusão térmica. Uma aproximação numérica bidimensional para esta formulação pode ser feita com o método das diferenças conforme a equação 3.2.

$$\nabla^2 u = \frac{u(i-1,j) - 2u(i,j) + u(i+1,j)}{\Delta x^2} + \frac{u(i,j-1) - 2u(i,j) + u(i,j+1)}{\Delta y^2} \tag{3.2}$$

A evolução do campo de temperatura no tempo é dada pela equação 3.3, que representa uma marcha temporal de um campo onde o termo $\Delta t \alpha \nabla^2 u_m(i,j)$ quantifica a variação da temperatura por passo temporal que é consequência de um fluxo de calor conhecido. O passo temporal é descrito na equação 3.4.

$$u_{m+1}(i,j) = u_m(i,j) + \Delta t \alpha \nabla^2 u_m(i,j) \tag{3.3}$$

$$\Delta t < \frac{1}{4\alpha} \frac{(\Delta x)^2 (\Delta y)^2}{(\Delta x)^2 + (\Delta y)^2} \tag{3.4}$$

Obs.: O algoritmo só é estável para:

$$\Delta t < \frac{1}{4\alpha} \frac{(\Delta x \Delta y)^2}{(\Delta x)^2 (\Delta y)^2} \tag{3.5}$$

Utilizando a formulação supracitada e um campo de temperatura inicial, com uma malha de 200x200 elementos, de uma garrafa fria que tem suas paredes aquecidas abruptamente no instante $t = t_0$, observa-se graficamente a evolução do campo de temperatura por 200s conforme a figura 12 em um código escrito em Python que rodou com velocidade média de 19s em um processador quad-core i5-8250U 1.6GHz com 8GB de memória ram.

Figura 12 – Evolução do Campo de Temperaturas 200x200

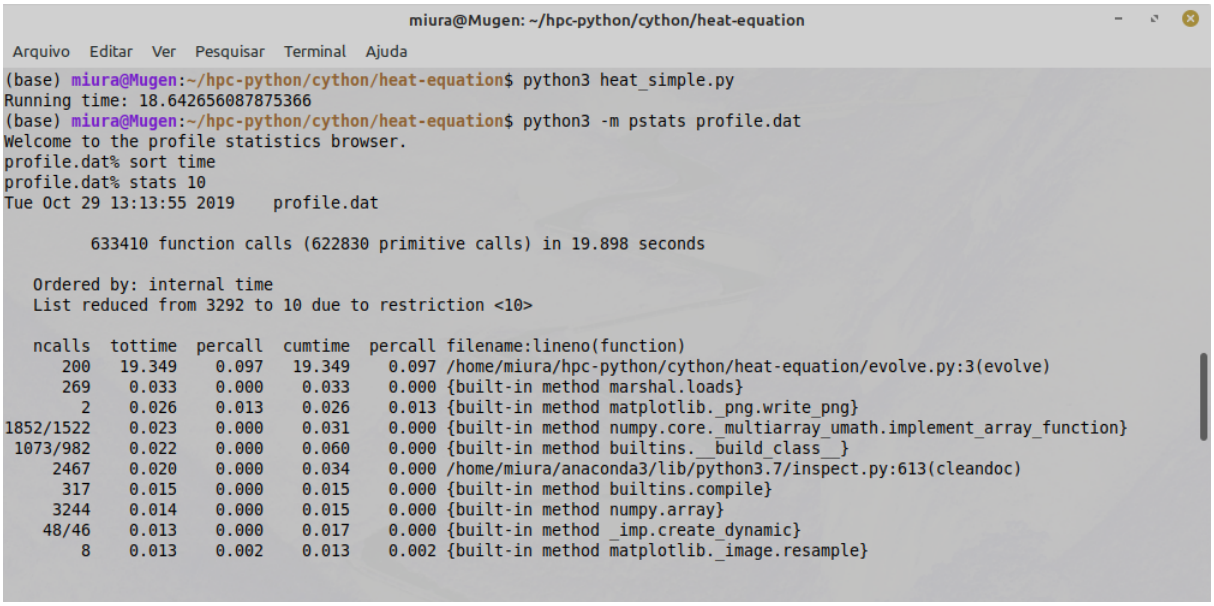
(a) Campo inicial

(b) Campo final

O Python disponibiliza um módulo em C chamado **cProfile** que executa um código e mede o tempo gasto em cada etapa do processo e posteriormente escreve estes dados em um arquivo de saída. Esta operação é realizada utilizando o comando `python3 -m cProfile -o arquivo_de_saida.dat arquivo_de_entrada.py`. O cProfile escreve, no arquivo de saída, os tempos de cada etapa em ordem de execução. Um código pode facilmente ter centenas de etapas tornando necessário um tratamento dos dados deste arquivo que pode ser feita pelo módulo **pstats** que faz um ordenamento e apresentação dos dados de acordo com a necessidade do usuário.

Uma caracterização deste código foi feita utilizando o programa cProfile para identificar os possíveis *HotSpots* visando uma otimização. O resultado é apresentado na figura 13 e de acordo com o que se verifica, uma única tarefa demanda quase a totalidade do tempo de execução do processo.

Figura 13 – Caracterização do Código de Transferência de Calor



Em Python, a definição do tipo de dado das variáveis é feita dinamicamente, o que torna mais prática a programação, mas esta opção faz com que seja necessário a verificação do tipo das variáveis para afirmar sua compatibilidade sempre que forem feitas operações entre elas. Na figura 14, analisando o trecho do código responsável pelo maior impacto no tempo de execução com uma ferramenta da biblioteca Cython, percebe-se que a formulação de diferenças finitas não é eficiente pois possui muitas interações do tipo Python. Esta tarefa foi feita utilizando o comando `cython -a arquivo.extensao` que gera um arquivo html.

Figura 14 – Tipos de Interação do Código

```
Generated by Cython 0.29.13

Yellow lines hint at Python interaction.
Click on a line that starts with a "+" to see the C code that Cython generated for it.

Raw output: evolve.c

+01: import numpy as np
 02:
+03: def evolve(u, u_previous, a, dt, dx2, dy2):
 04:     """Explicit time evolution.
 05:        u:            new temperature field
 06:        u_previous:   previous field
 07:        a:            diffusion constant
 08:        dt:           time step. """
 09:
+10:     n, m = u.shape
 11:
+12:     for i in range(1, n-1):
+13:         for j in range(1, m-1):
+14:             u[i, j] = u_previous[i, j] + a * dt * ( \
+15:               (u_previous[i+1, j] - 2*u_previous[i, j] + \
+16:                u_previous[i-1, j]) / dx2 + \
+17:               (u_previous[i, j+1] - 2*u_previous[i, j] + \
+18:                   u_previous[i, j-1]) / dy2 )
+19:     u_previous[:] = u[:]
 20:
```

Alguns ajustes foram feitos neste trecho do código para torná-lo mais rápido, como transformar esta função em uma extensão em C, definir o tipo das variáveis e trocar as divisões por multiplicações. Uma outra análise do código otimizado foi feita e pode ser vista na figura 15. Utilizando a nova extensão otimizada, o código foi executado novamente e desta vez obteve-se o tempo médio de 0,02s, um aumento de desempenho com um fator de aproximadamente 10^3. No trecho otimizado foram removidos as verificações de tipo de variáveis, a checagem de divisão por 0 e a opção de índices negativos do Python.

Figura 15 – Tipos de Interação Código Otimizado

```
Generated by Cython 0.29.13

Yellow lines hint at Python interaction.
Click on a line that starts with a "+" to see the C code that Cython generated for it.

Raw output: evolve.c

+01: # cython: profile=True
 02:
+03: import numpy as np
 04: cimport numpy as cnp
 05:
 06: import cython
 07:
 08: @cython.boundscheck(False)
 09: @cython.wraparound(False)
 10: @cython.cdivision(True)
+11: def evolve(cnp.ndarray[cnp.double_t, ndim=2]u,
 12:            cnp.ndarray[cnp.double_t, ndim=2]u_previous,
 13:            double a, double dt, double dx2, double dy2):
 14:     """Explicit time evolution.
 15:        u:            new temperature field
 16:        u_previous:   previous field
 17:        a:            diffusion constant
 18:        dt:           time step. """
 19:
+20:     cdef int n = u.shape[0]
+21:     cdef int m = u.shape[1]
 22:
 23:     cdef int i,j
 24:
 25:     # Multiplication is more efficient than division
+26:     cdef double dx2inv = 1. / dx2
+27:     cdef double dy2inv = 1. / dy2
 28:
+29:     for i in range(1, n-1):
+30:         for j in range(1, m-1):
+31:             u[i, j] = u_previous[i, j] + a * dt * ( \
+32:               (u_previous[i+1, j] - 2*u_previous[i, j] + \
+33:                u_previous[i-1, j]) * dx2inv + \
+34:               (u_previous[i, j+1] - 2*u_previous[i, j] + \
+35:                   u_previous[i, j-1]) * dy2inv )
+36:     u_previous[:] = u[:]
 37:
```

Após a otimização, o mesmo problema foi re-visitado utilizando uma malha de 1056x1056 elementos e gerou o resultado apresentado na figura 16 em 2,77s.

Figura 16 – Evolução do Campo de Temperaturas 1056x1056

3.5 Computação em Nuvem

Computação em nuvem é uma forma de prestação de serviço pela internet que se baseia em compartilhamento de recursos como capacidade de armazenamento e processamento de dados. As vantagens em seu emprego são inúmeras, dentre elas destaca-se: a disponibilidade do serviço 24/7; escalabilidade, pois o usuário só paga pelo que consumir quando consumir; economia de escala e uso de equipamentos de alta performance por assinatura mensal, sem investimento inicial.

Apesar de não ser diretamente relacionado com a computação de alta performance, a computação em nuvem ou *Cloud Computing* é um tema interessante a ser abordado nesta seção pois é uma alternativa inteligente para a formação de clusters de HPC. A solução é vantajosa tanto economicamente quanto operacionalmente por capitalizar sobre a economia de escala dos serviços de manutenção, mantenimento, atualização de hardware, não ter investimento inicial e não ter prejuízos por capacidade ociosa.

Como exemplo, um provedor deste serviço é a *Amazon Web Services* (AWS). É possível contratar o uso de máquinas virtuais por meio de uma plataforma na *web* em que o usuário seleciona o sistema operacional e as especificações que julgar necessárias. Uma pesquisa rápida no *market place* da AWS, realizada em 15 de setembro de 2019 às 18h, mostrou ser possível alugar uma máquina com 384GB de memória ram, 96 núcleos virtuais e uma taxa de transferência de dados de 20Gbps por US$ 6,624 por hora.

4 Ferramentas OpenSource

4.1 Desenvolvimento de Soluções OpenSource

O uso de soluções de softwares *Open Source* (OSS - *Open Source Software*) tem se tornado cada vez mais popular, possibilitando avanços importantes na área de pesquisa científica que se favorece com o compartilhamento de informações para desenvolvimento. De acordo com West e Gallagher (2006), as seguintes vantagens podem ser observadas para o meio acadêmico:

i Custo reduzido: Os OSS costumam ser de uso livre, tornando-se imediatamente atrativos para uso acadêmico. É importante ressaltar que nem todo OSS é gratuito. Em alguns casos é cobrada uma taxa por serviços adicionais e algumas vezes existe cobrança para seu uso empresarial.

ii Disponibilidade do código fonte: A disponibilidade do código fonte permite aos pesquisadores entenderem a solução proposta pelo autor e melhorá-lo ou criar uma versão própria da solução a partir das ideias obtidas, resultando em softwares de melhor qualidade.

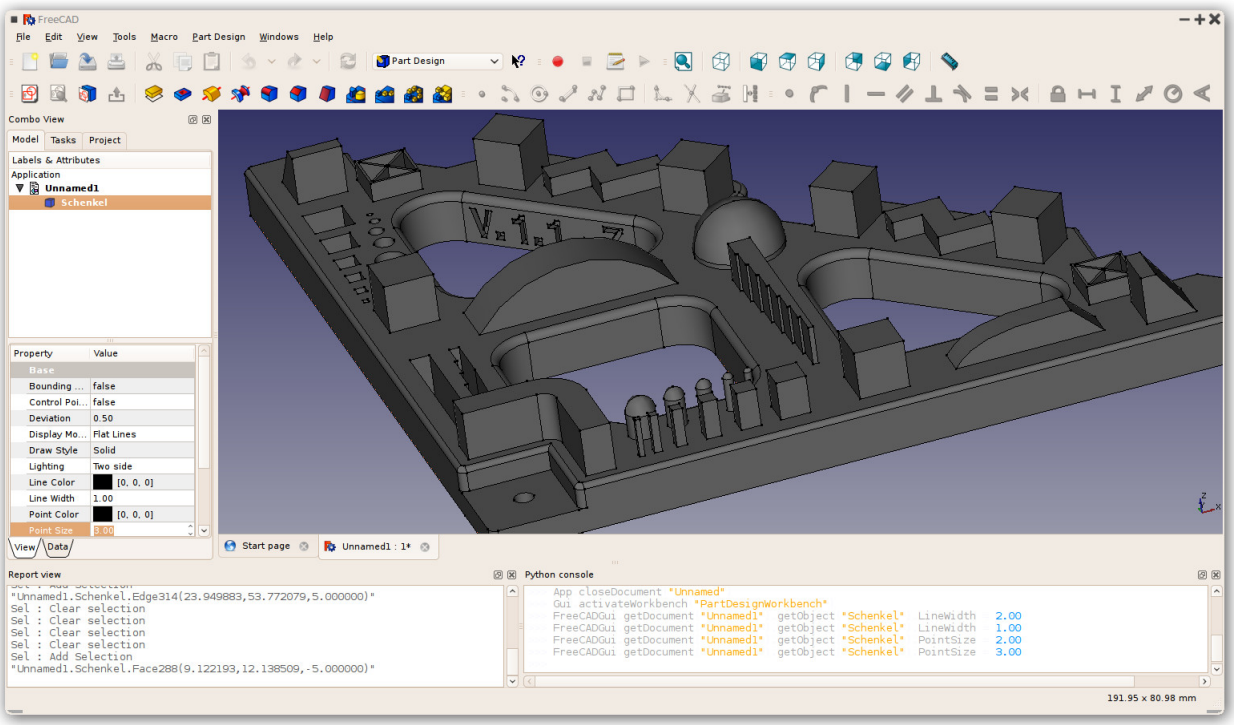
iii Maturidade: Os OSS estão atingindo níveis de confiança cada vez mais altos com versões estáveis e confiáveis.

A curva de aprendizado de uso de um OSS costuma ser mais severa e o suporte técnico é mais limitado, por estas e outras razões, seu uso deve ser adequadamente justificado para aplicações comerciais. Para o projeto em questão, serão utilizados os softwares: FreeCad para criação de modelos sólidos; GMSH para geração de malhas nos modelos sólidos; Meshio para estruturar e converter o formato das malhas; Paraview para o pós-tratamento dos dados; Cython para criar extensões em C. Uma característica fundamental que todos estes programas têm em comum é a extensibilidade, *i.e.*, a facilidade de modificação pelo usuário dos recursos disponíveis, bem como a integração com outros softwares.

4.2 FreeCad

O FreeCad é uma ferramenta de modelagem originalmente desenvolvida para uso de engenharia mecânica com ênfase na criação de projetos parametrizados. Um dos fatores que influenciaram fortemente a popularização de seu uso foi a possibilidade de utilização de linguagem de programação Python para acelerar e flexibilizar o processo de desenvolvimento de um produto (FALCK; FALCK; COLLETTE, 2012). Automatizando tarefas simples e a manipulação direta na criação de geometrias tornam-se inúmeras as possibilidades. Praticamente toda a aplicação pode ser acessada por um interpretador Python. As versões mais atuais também contam com suporte para simulações pelo método dos elementos finitos (FEM).

Figura 17 – Demonstração de Funcionalidade do FreeCad



Fonte: https://www.freecadweb.org/wiki/About_FreeCAD

4.3 GMSH

GMSH é um OSS cujo principal propósito é gerar malhas 2D e 3D, o projeto data de 1996. Na época, não existiam soluções *Open Source* e as ferramentas pagas eram muito caras, então os autores decidiram criar um OSS rápido, leve, com interface gráfica amigável ao usuário (GEUZAINE; REMACLE, 2009). O escopo era fazer um programa que combinasse um gerador de CAD, gerador de malha e um pós-processador de dados. Os pontos fortes da aplicação são:

i Velocidade e leveza: Para cumprir este propósito, o programa foi inteiramente escrito em C++.

ii Amigável ao usuário: Apesar de poder ser empregado como uma biblioteca, o programa conta com uma interface gráfica que traz uma solução completa para usuários iniciantes.

iii Robustez e portabilidade: É possível utilizar uma ampla gama de dados de entrada com uma certa tolerância a erros de *input* do usuário.

iv *Scriptability*: É possível utilizar o GMSH em rotinas de forma que todas as entradas são parametrizadas. Assim, o GMSH pode facilmente ser inserido em uma cadeia computacional.

O programa é dividido em 4 módulos: 1)geometria, 2)malha, 3)*solver* e 4)pós-processamento. Apenas o módulo de geração de malha será de interesse para este projeto.

4.4 Meshio

Meshio é uma biblioteca de código aberto capaz de estruturar malhas e converter seu formato de forma prática. É utilizado como uma sub-rotina. A biblioteca foi publicada na comunidade GitHub por Nico Schlömer sob uma licença do tipo *CopyLeft*, *i. e.*, um OSS de uso irrestrito com cláusula de reciprocidade que exige que os mesmos direitos sejam preservados para qualquer derivado deste trabalho.

4.5 ParaView

O ParaView é uma ferramenta científica de visualização e análise de grandes conjuntos de dados. Apesar de ter sido originalmente desenvolvida para aplicações extremas, houve por parte dos desenvolvedores um esforço para tornar o programa amigável e adequado para visualização de conjuntos menores de dados (AYACHIT, 2015). Possui interface gráfica interativa com técnicas de visualização que incluem paralelismo de dados. O processo de desenvolvimento e as ferramentas de visualização disponíveis, bem como exemplos de aplicação, são descritos por Ahrens, Geveci e Law (2005) de Los Alamos National Laboratory.

4.6 Cython

Como dito anteriormente, Python é uma linguagem interpretada. A implementação original de Python é o CPython desenvolvido por Guido Van Rossum. O CPython é um

programa em C que funciona como uma interface que compila um código Python em código Byte que depois será transformado em código binário para ser executado pela CPU. Python é uma linguagem inerentemente lenta mas uma forma imprática de tornar o código mais rápido seria, por exemplo, declarar as variáveis diretamente em CPython (a alocação não seria mais dinâmica) e evitar conversões. A ideia é imprática porque necessita que o usuário tenha um profundo conhecimento de como o Python funciona internamente.

O Cython é uma solução desenvolvida por Robert Bradshaw, Stefan Behnel, *et al.* Trata-se de um *superconjunto* de Python (SMITH, 2015) que escreve extensões em C e, segundo (HERRON, 2016), torna este processo tão simples quanto escrever em Python nativamente.

5 Elementos de Contorno

5.1 Condução de Calor Bidimensional e Tridimensional

Como forma de demonstrar o método dos elementos de contorno, nesta seção serão desenvolvidas as formulações para condução de calor bidimensional e tridimensional com elementos constantes. Serão apresentados os resultados principais de cada etapa. Estas deduções foram retiradas dos textos de Albuquerque (2012), Katsikadelis (2016) e Loyola (2017).

5.1.1 Conhecimento Preliminar

A equação governante da teoria potencial utilizada pelo BEM é dada pela equação 5.1. Para $f \neq 0$, esta é conhecida como a equação de Poisson enquanto para $f = 0$ é conhecida como a equação de Laplace.

$$\nabla^2 u = f(x, y) \qquad (x, y \in \Omega) \tag{5.1}$$

A equação 5.1 descreve muitos fenômenos físicos como fluxo em regime permanente de fluido, calor, eletricidade, tensão em elementos sólidos, entre outros. A solução para o campo é buscada em um domínio Ω fechado de forma que satisfaça as condições no contorno Γ que podem ser dadas por *i*, *ii* ou uma combinação linear entre eles.

i Condição de Dirichlet

$\nabla^2 = f$ em Ω
$u = \bar{u}$ em Γ

ii Condição de Neumann

$\nabla^2 = f$ em Ω
$\frac{\partial u}{\partial n} = \bar{u}_n$ em Γ

5.1.2 Solução Fundamental Para Temperatura e Fluxo de Calor

A equação de Fourier para condução de calor é descrita como:

$$\nabla^2 T = -\frac{\dot{q}_g}{k} \tag{5.2}$$

Uma solução fundamental para a equação de Laplace da condução de calor, para problemas bidimensionais, deduz que:

$$T^* = \frac{-1}{2\pi k} ln(r) \tag{5.3}$$

Onde r é a distância entre o ponto fonte e o ponto campo.

Define-se o fluxo de calor através do contorno pela expressão:

$$q = -k\frac{\partial T}{\partial n} \tag{5.4}$$

A solução fundamental para o fluxo de calor bidimensional é dadas por:

$$q^* = -k\frac{\partial T^*}{\partial n} \tag{5.5}$$

Substituindo 5.3 em 5.5 e simplificando, a solução fundamental do fluxo, para o caso bidimensional, se apresenta na forma:

$$q^* = \frac{1}{2\pi r^2}[(x - x_d)n_x + (y - y_d)n_y] \tag{5.6}$$

Em uma segunda abordagem, a solução fundamental para a equação de Laplace, para problemas tridimensionais, é apresentada na equação 5.7.

$$T^* = \frac{1}{4\pi r} \tag{5.7}$$

A solução fundamental para o fluxo de calor tridimensional é dadas por:

$$q^* = -k\frac{\partial T^*}{\partial n} = -\frac{\frac{\partial r}{\partial n}}{4\pi r} \tag{5.8}$$

5.1.3 Formulação dos Elementos de Contorno

A partir da equação de Laplace para o campo de temperatura, deduz-se a equação da integral de contorno. Uma formulação generalizada desta integral é descrita como:

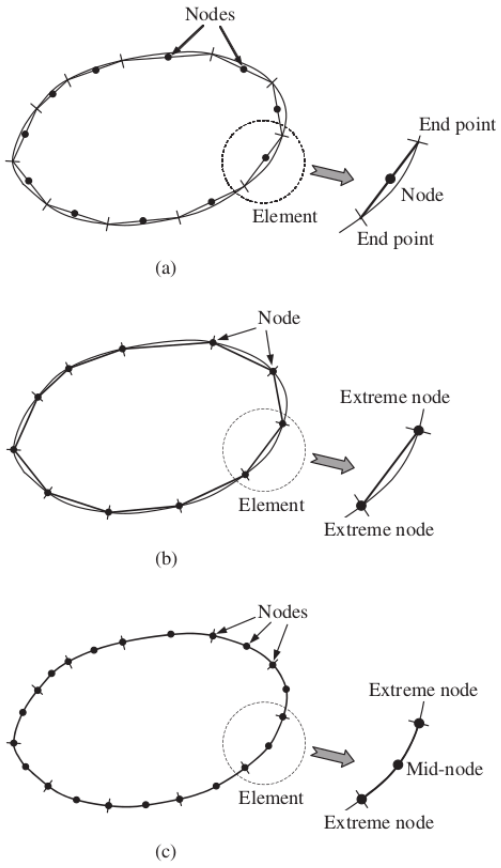
$$cT(x_d, y_d) = \int_s Tq^* ds \; - \; \int_s T^* q ds \tag{5.9}$$

$$c = \begin{cases} 1, & \text{se } x_d, y_d \in \text{ao domínio} \\ \frac{\theta_{int}}{2\pi}, & \text{se } x_d, y_d \in \text{ao contorno} \\ 0, & \text{se } x_d, y_d \notin \text{ao domínio} \end{cases} \tag{5.10}$$

Obs.: θ_{int} é o ângulo interno do contorno de forma que, se o ponto fonte encontra-se em um ponto suave do contorno, *i.e.*, não se encontra em um canto, $c = \frac{1}{2}$.

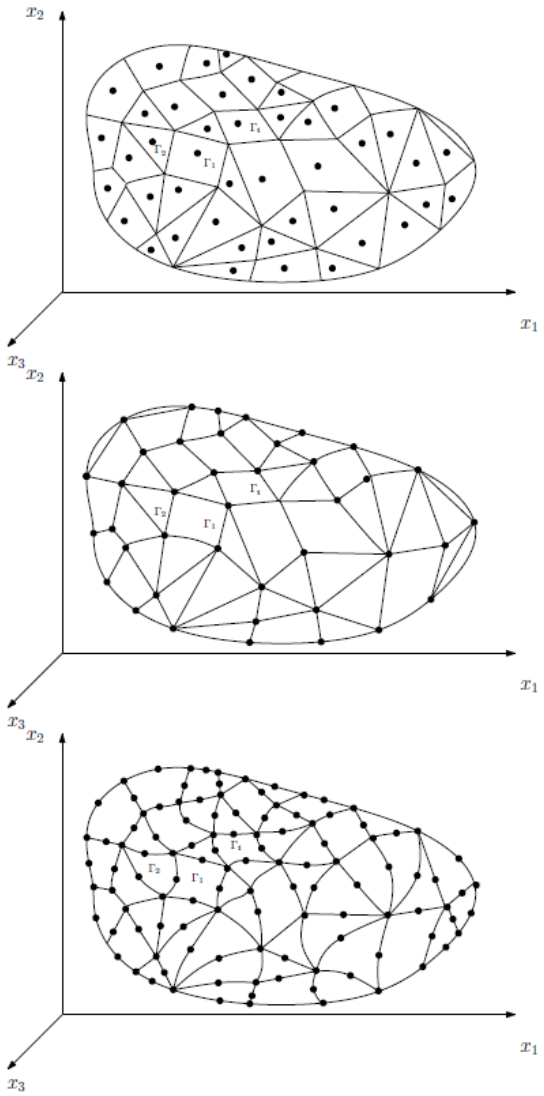
Para fins práticos de engenharia, a solução analítica da equação 5.9 está fora de questão. Com os avanços computacionais é possível obter uma solução numérica do problema. Para um domínio arbitrário Ω com contorno Γ, o BEM propõe segmentar o contorno em N elementos, não necessariamente iguais, chamados de "elementos de contorno". Desta forma, transforma-se a integral de contorno em uma soma das integrais ao longo de cada elemento. Duas aproximações são feitas por conta disso, a aproximação da geometria imposta pelo formato dos elementos e a aproximação da forma como a propriedade varia ao longo dos elementos. Os tipos de elemento normalmente utilizados, classificados de acordo com a ordem da aproximação, são: elementos constantes, lineares e parabólicos ou quadráticos (KATSIKADELIS, 2016). As figuras 18 e 19 ilustram estes elementos para os casos bidimensionais e tridimensionais.

Figura 18 – Alguns tipos de elemento de contorno bidimensionais.



(a)Constantes; (b)Lineares; (c)Parabólicos
Fonte: Katsikadelis, 2016, p.61.

Figura 19 – Alguns tipos de elemento de contorno.



(a)Constantes; (b)Lineares; (c)Parabólicos
Fonte: Loyola, 2017, p.22.

5.1.4 Elementos Constantes

Na formulação constante bidimensional, conforme demonstrado na figura 18 (a), o nó estará sempre no meio do elemento, desta forma $c = \frac{1}{2}$. Outra consideração é que a temperatura T e o fluxo q são assumidos constantes ao longo do elemento, portanto, podem sair da integral e a equação 5.9 pode ser reescrita da forma:

$$-\frac{1}{2}T^i(x_d, y_d) + \sum_{j=1}^{n}\left[T_j \int_{\Gamma_j} q^* d\Gamma\right] = \sum_{j=1}^{n}\left[q_j \int_{\Gamma_j} T^* d\Gamma\right] \tag{5.11}$$

A equação 5.11 também vale para casos tridimensionais. Uma forma de discretizar a superfície da geometria tridimensional é com elementos triangulares, conforme pode ser visto na figura 19. Os nós se encontram no centroide do elemento e as propriedades são

assumidas constantes ao longo de toda sua superfície. Como o *j-ésimo* nó estará sempre no centroide do *j-ésimo* elemento, ele estará localizado em uma região suave do contorno (LOYOLA, 2017). Da mesma forma que o caso bidimensional, $c = \frac{1}{2}$.

As integrais da equação 5.11 relacionam o i-ésimo, onde a solução fundamental é aplicada, com o j-ésimo nó. Trata-se de uma quantificação da contribuição dos valores T_j e q_j na formação de T^i e são chamados de "coeficiente de influência" (KATSIKADELIS, 2016). Estes coeficientes denotados por H'_{ij} e G_{ij} são definidos como:

$$H'_{ij} = \int_{\Gamma_j} q^* d\Gamma \tag{5.12}$$

$$G_{ij} = \int_{\Gamma_j} T^* d\Gamma \tag{5.13}$$

Substituindo 5.12 e 5.13 em 5.11, obtém-se a forma discreta da integração de contorno:

$$-\frac{1}{2}T^i + \sum_{j=1}^{n}[H'_{ij}T_j] = \sum_{j=1}^{n}[G_{ij}q_j] \tag{5.14}$$

Ainda:

$$H_{ij} = H'_{ij} - \frac{1}{2}\delta_{ij} \tag{5.15}$$

Com 5.15, a equação 5.14 se reescreve como:

$$\sum_{j=1}^{n}[H_{ij}T_j] = \sum_{j=1}^{n}[G_{ij}q_j] \tag{5.16}$$

Escrevendo a equação 5.16 em forma linear algébrica:

$$[H]\{T\} = [G]\{q\} \tag{5.17}$$

Onde [H] e [G] são matrizes quadradas $N \times N$, T e q são vetores $N \times 1$. Para cada elemento, sempre que se souber a temperatura é possível obter o fluxo e vice-versa. O sistema algébrico $N \times N$ precisa de N condições de contorno, sendo uma condição de contorno por elemento. Para um sistema algébrico linear isto é equivalente a dizer que o problema é "bem colocado", *i.e.*, existe uma solução que é única. Uma maneira mais fácil de manipular este sistema é reescrevendo todas quantidades desconhecidas para uma matriz do lado esquerdo da equação e as quantidades conhecidas à direita.

$$[A]\{x\} = \{b\} \tag{5.18}$$

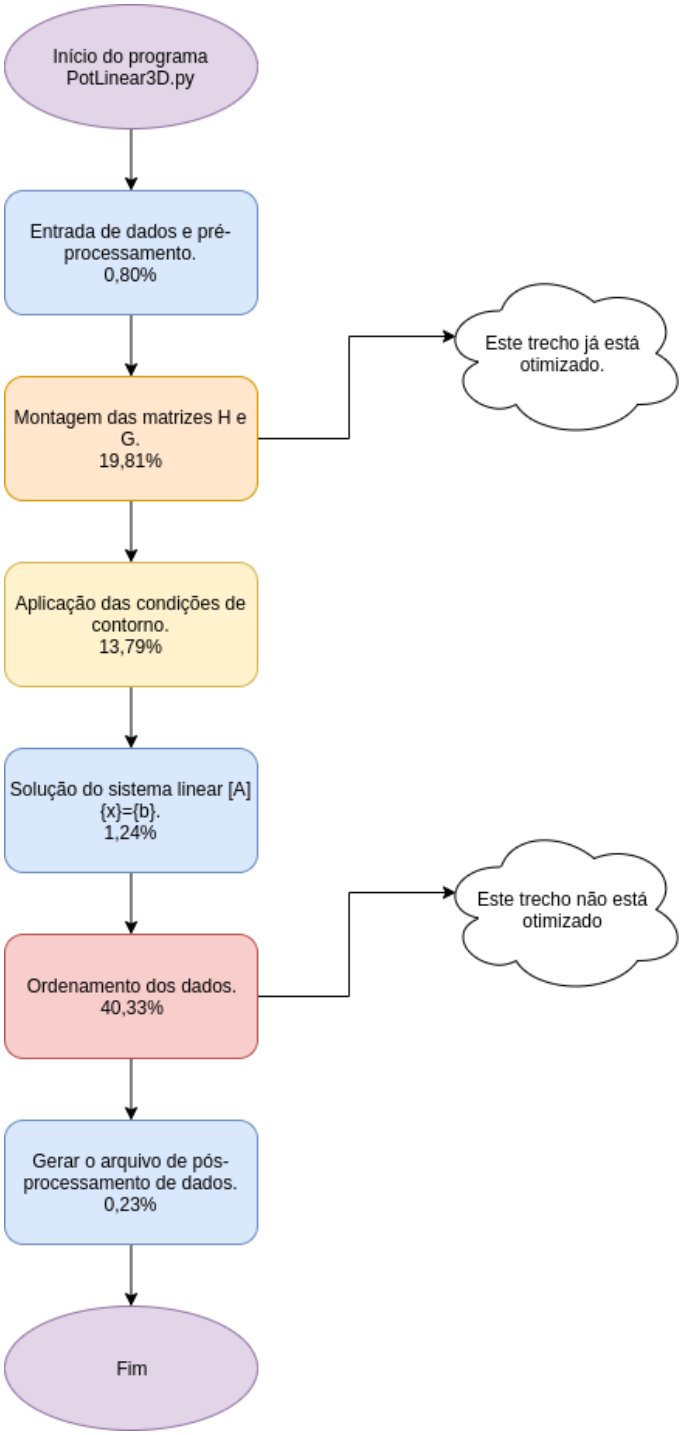
Finalmente, a solução do BEM é dada pela solução do sistema linear:

$$\{x\} = [A]^{-1}\{b\} \tag{5.19}$$

5.2 Análise Preliminar do Código PotLinear3D

A figura 20 mostra o fluxograma da implementação em análise, o programa "PoTLinear3D.py". Inicialmente é feita a caracterização do código, que identifica o tempo gasto em cada etapa, o que se pode chamar de perfilagem. Com base nos temporizadores próprios do código, montou-se o fluxograma de tarefa com o respectivo impacto de cada etapa no tempo de execução. Obs.: O fluxograma contém apenas as etapas macro de interesse no processo e o somatório de todas as etapas não será equivalente ao tempo total de execução.

Figura 20 – Fluxograma PotLinear3D



É interessante observar que apesar de estar escrito em Python, a maior parte dos trechos do código não possuem impacto significativo e uma otimização não implicaria em um ganho de performance perceptível. Os dois trechos mais críticos são da montagem das matrizes [H] e [G] e o ordenamento dos dados. A montagem das matrizes já está previamente otimizada, neste caso, baseando-se no fluxograma da figura 20 a escolha natural para otimização seria o trecho de ordenamento dos dados, no entanto, este trabalho não irá focar neste trecho. Caso o objetivo fosse otimizar a sub-rotina "monta_Teq", uma avaliação preliminar do trecho, demonstrada na figura 21, indica que podem haver ganhos de performance pela definição das variáveis e pela vetorização dos *loops for*, nativa da linguagem C. A figura 22 apresenta o trecho reescrito após torná-lo uma extensão em C, com as variáveis declaradas.

Figura 21 – Avaliação da sub-rotina monta_Teq

```
#%%  SEPARAÇÃO DOS VALORES DE T e Q
def monta_Teq(NOS, ELEM, CDC, x, T_pr):
    """
    Programa:  monta_Teq.m
    Descrição: Separa o vetor de valores calculados {x} e os valores
               prescritos [CDC] em valores de potencial {T} e valores de
               fluxo {q}.
    Autor:     Gustavo Gontijo

    Última modificação: 12/03/2014 - 18h10min
    """

    nnos = NOS.shape[0]      # Nro de nós
    nelem = ELEM.shape[0]    # Nro de elementos
    T = np.zeros(nnos)       # Inicia o vetor T (potenciais)
    q = np.zeros(3*nelem)    # Inicia o vetor q (fluxos)

    # Coloca os valores prescritos nos vetores T e q
    for i in range(0,nelem):
        tipo_cdc = CDC[i,0]
        valor_cdc = CDC[i,1]
        if tipo_cdc == 0:    # O potencial é prescrito
            indT = ELEM[i]
            for k in range(0,3):
                T[indT[k]] = valor_cdc
        else:                # O fluxo é prescrito
            indq = (i*3) + np.array([0, 1, 2])
            for k in range(0,3):
                q[indq[k]] = valor_cdc

    # Adiciona nos vetores T e q os valores calculados (estão em {x})
    for i in range(0,nnos):
       sw = 0
       for j in range(0,nelem):
          if T_pr[i,j] == 1:  #  O nó teve potencial prescrito no elemento j,
                              # portanto, o valor calculado é de fluxo.
             sw = 1
             for k in range(0,3):
                if ELEM[j,k] == i:
                   indq = (j*3-3) + k - 1

             q[indq] = x[i]

       if sw==0:  # O nó não teve potencial prescrito em nenhum elemento,
                  # portanto, o valor calculado é de potencial.
          T[i] = x[i]

    return T, q
```

Figura 22 – sub-rotina monta_Teq reescrita

```
import numpy as np
cimport numpy as cnp

import cython

cimport cython
cimport openmp
from cython.parallel cimport prange
from cython.parallel cimport parallel

@cython.boundscheck(False)
@cython.wraparound(False)
@cython.cdivision(True) # to avoid the exception checking
@cython.nonecheck(False)

@cython.initializedcheck(False)

def monta_Teq(
 cnp.ndarray[cnp.float64_t, ndim=2]NOS,
 cnp.ndarray[cnp.int64_t, ndim=2]ELEM,
 cnp.ndarray[cnp.float64_t, ndim=2]CDC,
 cnp.ndarray[cnp.float64_t, ndim=1]x,
 T_pr
 ):
    """
    Programa:  monta_Teq.m
    Descrição: Separa o vetor de valores calculados {x} e os valores
               prescritos [CDC] em valores de potencial {T} e valores de
               fluxo {q}.
    Autor:     Gustavo Gontijo

    Última modificação: 12/03/2014 - 18h10min
    """

    cpdef int nnos = NOS.shape[0]     # Nro de nós
    cpdef int nelem = ELEM.shape[0]   # Nro de elementos
    cpdef cnp.ndarray[cnp.float64_t, ndim=1]T = np.zeros(nnos)      # Inicia o vetor T (potenciais)
    cpdef cnp.ndarray[cnp.float64_t, ndim=1]q = np.zeros(3*nelem)   # Inicia o vetor q (fluxos)

    cdef int i, j, k


    # Coloca os valores prescritos nos vetores T e q

    cdef float tipo_cdc, valor_cdc
    cpdef cnp.ndarray[cnp.long_t, ndim=1]indT
    cpdef cnp.ndarray[cnp.long_t, ndim=1]indq

    for i in range(0,nelem):
        tipo_cdc = CDC[i,0]
        valor_cdc = CDC[i,1]
        if tipo_cdc == 0:    # O potencial é prescrito
            indT = ELEM[i]
            for k in range(0,3):
                T[indT[k]] = valor_cdc
        else:                # O fluxo é prescrito
            indq = (i*3) + np.array([0, 1, 2])
            for k in range(0,3):
                q[indq[k]] = valor_cdc

    # Adiciona nos vetores T e q os valores calculados (estão em {x})

    cdef int indr
    for i in range(0,nnos):
       sw = 0
       for j in range(0,nelem):
          if T_pr[i,j] == 1:  #  O nó teve potencial prescrito no elemento j,
                              # portanto, o valor calculado é de fluxo.
             sw = 1
             for k in range(0,3):
                if ELEM[j,k] == i:
                   indr = (j*3-3) + k - 1

             q[indr] = x[i]

       if sw==0:  # O nó não teve potencial prescrito em nenhum elemento,
                  # portanto, o valor calculado é de potencial.
          T[i] = x[i]

    return T, q
```

Mesmo com a implementação do trecho em Cython, as expectativas iniciais não se concretizaram e o código não apresentou diferença de performance mensurável. Estudos adicionais serão necessários para compreender o motivo.

6 Otimização do Código PotLinear3D

6.1 Paralelização da montagem das matrizes [H] e [G]

A parte do código referente a montagem das matrizes [H] e [G] apresenta uma carga de trabalho, dada por uma integração numérica, facilmente divisível por não haver interdependência entre os resultados. Este tipo de carga de trabalho é conhecida como *Embarrasingly Parallel*, que em tradução livre seria uma região "fácilmente paralelizável". O completo oposto desse caso é o da carga de trabalho denominada *Intrinsically Sequential*; esta expressão é utilizada para os casos onde a paralelização é impossível.

De fato, o problema da montagem das matrizes [H] e [G] é um bom candidato para se observar os benefícios da paralelização, portanto, este capitulo trará a análise comparativa deste trecho sendo executado de forma sequencial e de forma paralela utilizando as diretivas OpenMP que tratam do paralelismo com compartilhamento de memória. Para tanto, é necessário entender o que se espera obter como resultado desta mudança. Todas as rotinas em análise neste capítulo estão disponíveis nos anexos para consulta.

6.1.1 Aceleração do Código (*SpeedUp*)

De forma simplificada, vamos considerar que a aceleração de um processo é dada por:

$$speedup = \frac{T_s}{T_p} \tag{6.1}$$

T_p é o tempo total gasto pelo código paralelizado por n núcleos, T_s é o tempo da versão sequencial deste código e *speedup* é a aceleração.

Não existem códigos sem partes sequenciais, por menores que sejam. Em algum momento todo algoritmo vai ter que iniciar a partir de um processo único que pode se separar em regiões paralelas e após isso retornar à região sequencial. Dito isto, o senso comum é que só é possível se beneficiar de uma aceleração, dada por paralelização, em

regiões paralelas. Sempre haverá a carga de trabalho sequencial (S) e a paralela (P); desta forma:

$$T_p = S + P \tag{6.2}$$

Os possíveis casos de aceleração, para uma carga de trabalho dividida entre n núcleos, são:

speedup $\geq n$ - Este caso é raríssimo e só ocorre quando existem problemas no código sequencial como uma quantidade de informações que não cabe na memória cache e força o processador a acessar a memória principal consumindo mais tempo. Em tese seria possível obter tal aceleração caso a paralelização reduzisse o tamanho do problema em partes menores que não extrapolam um valor limite de memória mas, em geral, não se espera que aconteça.

$n >$ *speedup* ≥ 1 - Aqui costumam cair a maior parte dos casos. Quanto mais próximo de n, maior a eficiência da paralelização. Não se espera atingir o n pois sempre existe a carga de trabalho sequencial e também o tempo gasto para alocar recursos todas as vezes que uma *Thread* recebe uma tarefa, conhecido como ***Overhead Call*** ou em tradução livre uma "Chamada de Cabeçote" que corresponde a um tempo de comunicação. Quanto maior o número de Threads, abertas na região paralela, maior o tempo gasto com *Overhead Call*.

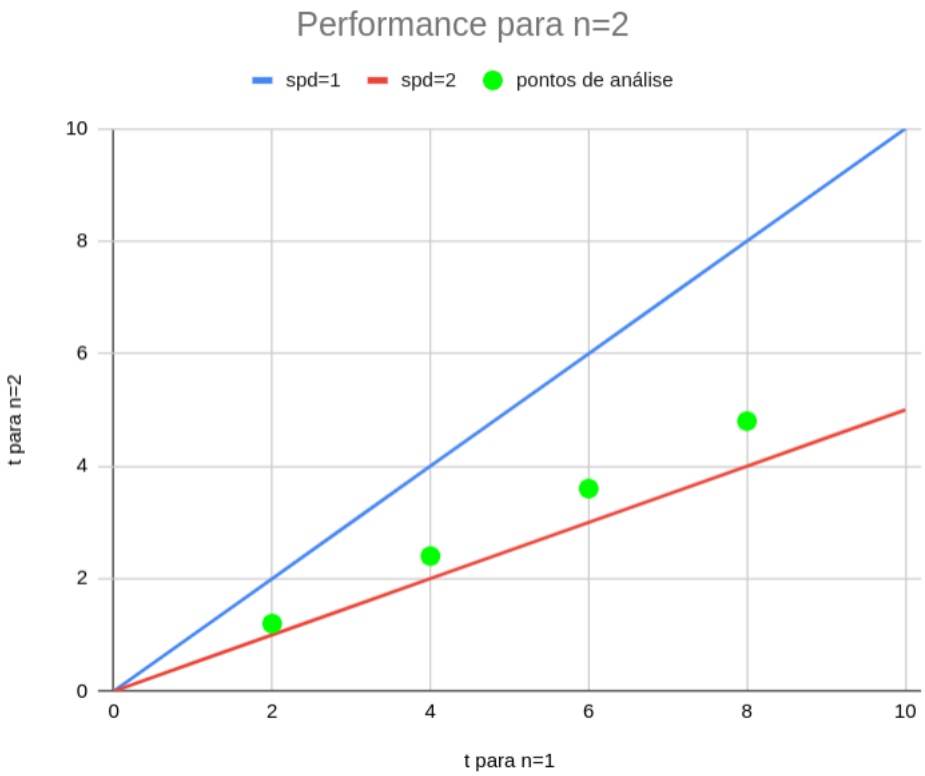
$1 >$ *speedup* - Esse caso ocorre somente quando existem *bugs* na região paralela do código. Pode ser causado, por exemplo, pelo GIL.

É possível abrir um número de *Threads* maior do que o número de núcleos. Mas nesse caso, não haverá ganho de aceleração pois as threads excedentes não funcionarão de forma paralela e sim concorrente, adicionando apenas o tempo de *Overhead Call*.

A figura 23 traz uma ilustração de como será feita a análise do desempenho do PotLinear3D. O eixo x representa o tempo gasto por um único processador para realizar uma tarefa enquanto o eixo y é o tempo gasto por n processadores com a mesma tarefa, no caso do exemplo, $n = 2$. A linha azul representa o limite no qual $speedup = 1$ enquanto a linha vermelha representa $speedup = n$. Os pontos verdes são a intercessão dos dados temporais obtidos rodando o programa; cada ponto verde equivale a uma malha diferente. A metodologia proposta é de analisar 3 malhas diferente utilizando, em cada ponto, uma média de tempo de 10 execuções. Serão estudados os casos para 2, 4, 6 e 8 *Threads*. Lembrando que qualquer ponto compreendido entre as duas linhas de limite representa um ganho de desempenho que é maior quanto mais próximo da linha vermelha. O segundo ponto a se observar é que a aceleração obtida é o inverso da inclinação da curva.

Para tornar esta análise mais prática, foi criado um algoritmo que irá executar apenas a parte do código referente à criação das matrizes [H] e [G], alternando as malhas, o número de núcleos e a rotina de calculo utilizada. O código "PotLinear3D_tempos.py" está disponível nos anexos.

Figura 23 – Gráfico para análise de desempenho



6.1.2 Consistência do Código

Na programação em paralelo com compartilhamento de memória existe uma grande facilidade de comunicação entre as unidades de processamento (CPUs) pois a memória é de domínio comum. Por outro lado, isto introduz novas dificuldades pois a ordem de execução de uma tarefa pode alterar o resultado. Para isso existem os Modelos de Consistência de dados.

Conforme definido por Lamport (2016), um sistema de multiprocessadores é dito consistente se o resultado de qualquer execução for como se as operações fossem executadas de forma sequencial, e a operação individual dos processadores aparece na ordem especificada pelo programa. A ideia funciona mas o custo de definir ordens especificas de atuação dos processadores é grande e perde-se o potencial de aceleração. Outros modelos de consistência flexíveis também foram propostos e, quanto mais flexível, maior o ganho de performance possível porém a um custo alto por parte da programação pois exige que o programador entenda a arquitetura do *Hardware* utilizado para prever em qual momento quais informações estarão disponíveis.

Segundo Chapman, Jost e Pas (2008), o OpenMP serve justamente para equilibrar esses extremos, tornando automáticas algumas dessas operações de sincronização de dados e fazendo com que o *Software* seja mais consistente sem prejudicar excessivamente o desempenho e facilitando a vida do programador. O OpenMP exige que o programador

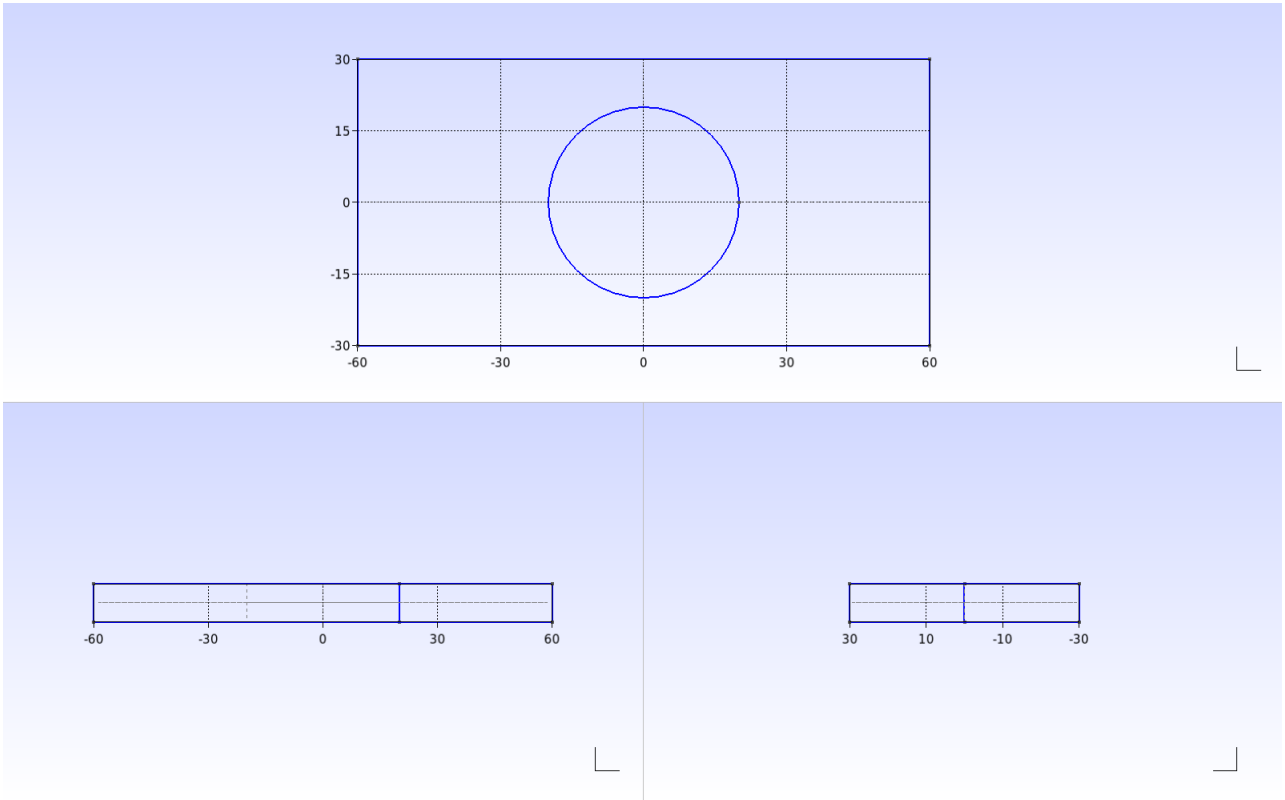
considere que não existe ordem específica de execução de tarefas entre as CPUs e, contanto que essa regra seja seguida, o código sempre dará o mesmo resultado que um código sequencial para a mesma tarefa. Como as variáveis operadas pelas *threads* podem ser globais, se duas ou mais *threads* concorrem o uso de uma mesma variável de iteração, é provável que em algum momento uma *thread* utilize um valor que foi reescrito por outra imediatamente antes de seu uso, este fenômeno é conhecido como ***Race Condition*** e provoca erros grosseiros na saída de dados.

A exatidão dos dados fornecidos pelo "PotLinear3D.py" será checada a partir de uma rotina de verificação pela qual é feita uma simulação de um caso simples, a transferência de calor por um cubo, e comparando os dados esperados de fluxo de calor nas superfícies com os obtidos pela simulação. A rotina de teste "PotLinear3D_teste.ipynb" está disponível em anexo. O formato "arquivo.ipynb" se refere ao documento ser do tipo Jupyter Notebook, usado para executar códigos em computação na nuvem; neste caso utilizou-se o servidor do Google Colab, que permite criar documentos e executar códigos diretamente do Google Drive.

6.1.3 Simulação em Análise

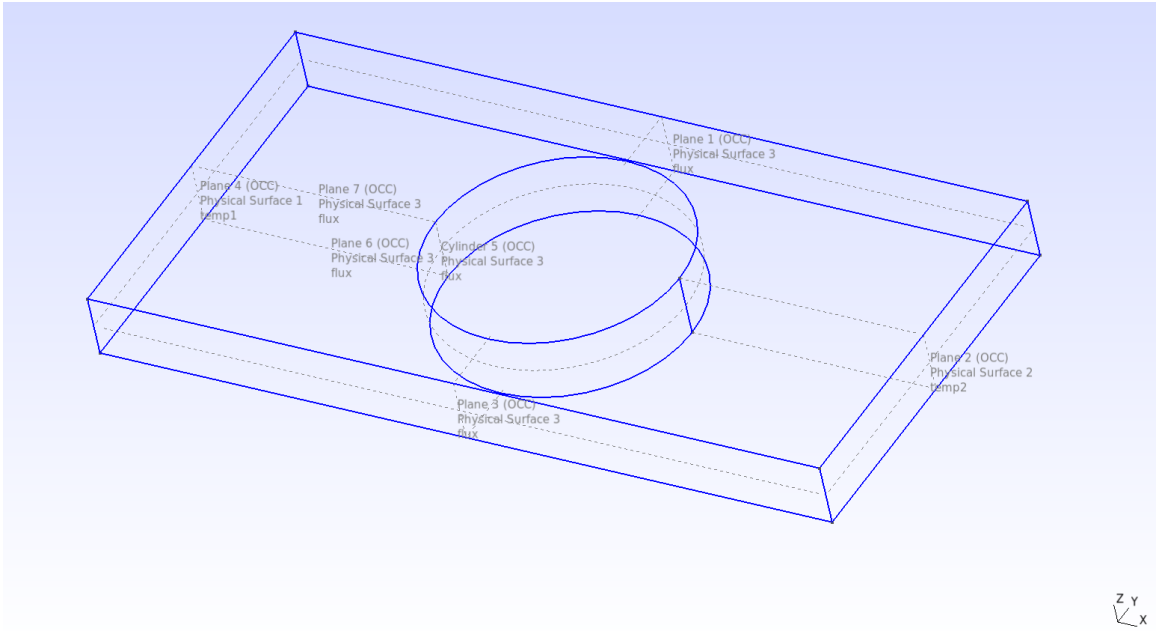
O modelo físico a ser analisado é o de uma placa com um furo conforme ilustrado pelas vistas na figura 24.

Figura 24 – Vistas - Placa com Furo



As condições de contorno do problema são definidas para cada superfície. Conforme verificável na figura 25, a superfície física 1 tem condição de temperatura conhecida T_1, a superfície física 2 tem condição de temperatura conhecida T_2, as demais superfícies possuem condição de fluxo conhecido $k = 1\frac{W}{m^2K}$.

Figura 25 – Perspectiva - Placa com Furo



A simulação será feita com três malhas diferentes para que fique mais fácil observar o impacto da paralelização sobre o código. As malhas foram geradas automaticamente pelo programa "GMSH" e discretizam o problema por elementos triangulares 2D no contorno do modelo físico.

Figura 26 – Malha 1 - Placa com Furo

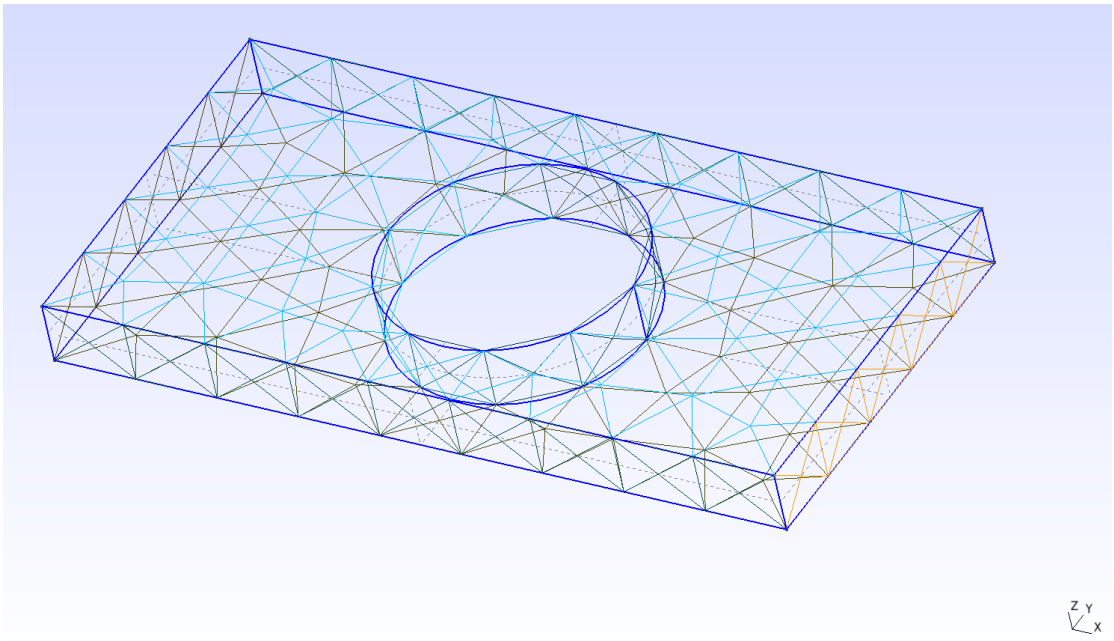


Figura 27 – Malha 2 - Placa com Furo

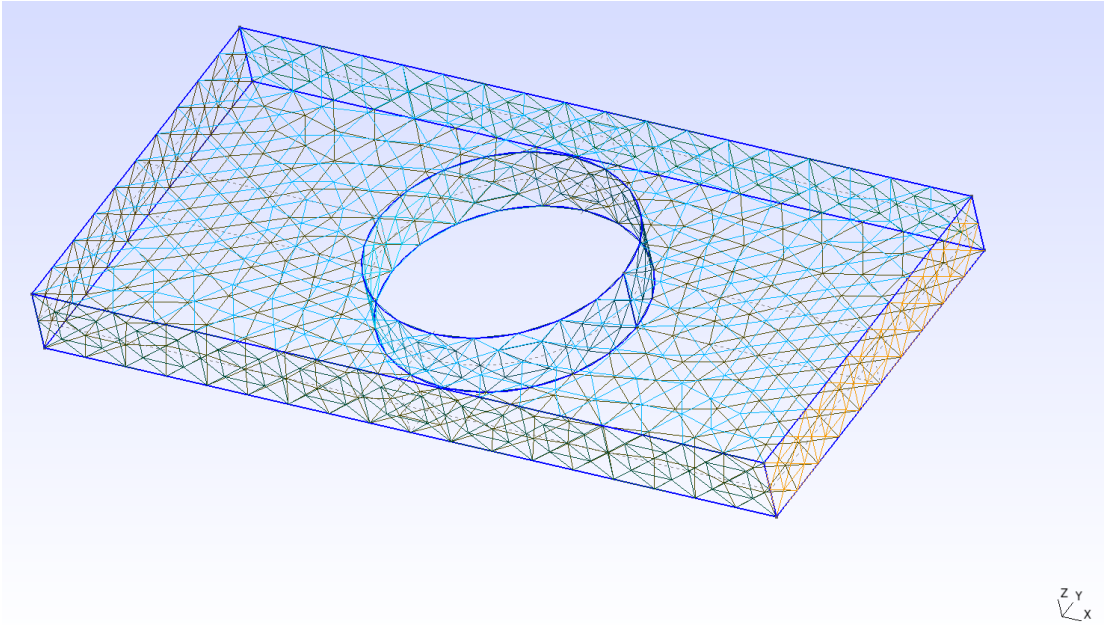
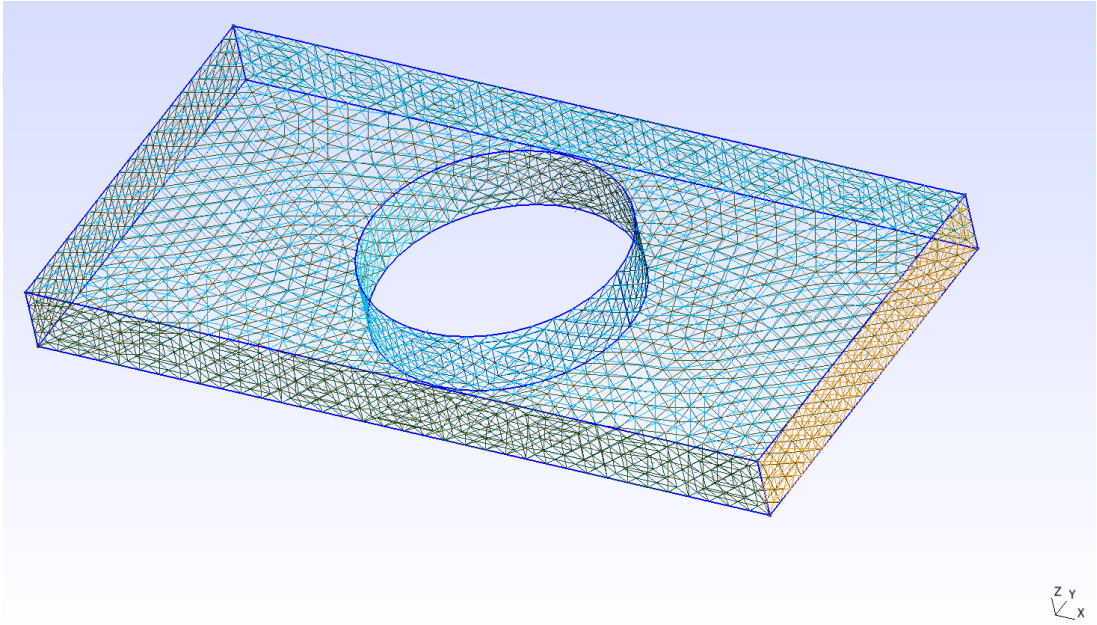


Figura 28 – Malha 3 - Placa com Furo



A malha 1 possui 364 elementos; a malha 2 possui 1456 elementos; a malha 3 possui 5824 elementos. Como é possível notar, o número de elementos entre as malhas cresce por um fator de 4. Como um código de elementos de contorno tem complexidade N^2, espera-se que o tempo de processamento aumente por um fator de 16 a cada troca de malha. A máquina utilizada para a análise está equipada com um processador com as especificações listadas na figura 29.

Figura 29 – Especificações do Processador

```
(base) miura@Mugen:~$ lscpu
Architecture:        x86_64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
CPU(s):              8
On-line CPU(s) list: 0-7
Thread(s) per core:  2
Core(s) per socket:  4
Socket(s):           1
NUMA node(s):        1
Vendor ID:           GenuineIntel
CPU family:          6
Model:               142
Model name:          Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz
Stepping:            10
CPU MHz:             698.077
CPU max MHz:         3400,0000
CPU min MHz:         400,0000
BogoMIPS:            3600.00
Virtualization:      VT-x
L1d cache:           32K
L1i cache:           32K
L2 cache:            256K
L3 cache:            6144K
NUMA node0 CPU(s):   0-7
```

6.1.4 Códigos Utilizados

A região de interesse do código "PotLinear3D.py" é a rotina de cálculo dos elementos das matrizes [H] e [G], portanto, as análises comparativas se concentraram nesse trecho. A figura 30 traz o código serial responsável por estes cálculos e a otimização será aplicada neste algoritmo. Observa-se que são usadas três funções, são elas: *calc_vetnormal; calcula_Gs; calcula_HeGns*. Estas funções não serão mostradas pois são de pouco interesse nesta etapa visto que permanecerão inalteradas, porém, os códigos estarão disponíveis nos anexos.

Inicialmente foi feita uma tentativa ingênua de implementação da paralelização deste trecho, mostrada na figura 31. A diferença entre os dois algoritmos está no cabeçalho onde se usa "*with nogil*" para desabilitar o GIL do Python e o loop for utiliza a função "*prange*" ao invés de "*range*". O "*prange*" é a função do "*Cython*" que indica que um loop deve ser executado por múltiplas threads. O parâmetro do tipo de *schedule* "*static*" significa que cada *thread* executará trechos com aproximadamente a mesma carga de trabalho. O parâmetro "*num_threads*" indica em quantas *threads* o loop deverá ser dividido; se este parâmetro não for indicado, o comportamento padrão é de utilizar todas as *threads* disponíveis.

Figura 30 – Cálculo Serial das Matrizes [H] e [G]

```
    #%% = = = = = CÁLCULO DOS ELEMENTOS DAS MATRIZES H e G = = = = =
    for pc in range(nelem):
        nos[0] = ELEM[pc,0]      # Nós que compõem o elemento
        nos[1] = ELEM[pc,1]      # Nós que compõem o elemento
        nos[2] = ELEM[pc,2]      # Nós que compõem o elemento
        X1[0] = NOS[nos[0],0]    # Coordenadas do nó 1 do elemento
        X1[1] = NOS[nos[0],1]    # Coordenadas do nó 1 do elemento
        X1[2] = NOS[nos[0],2]    # Coordenadas do nó 1 do elemento
        X2[0] = NOS[nos[1],0]    # Coordenadas do nó 2 do elemento
        X2[1] = NOS[nos[1],1]    # Coordenadas do nó 2 do elemento
        X2[2] = NOS[nos[1],2]    # Coordenadas do nó 2 do elemento
        X3[0] = NOS[nos[2],0]    # Coordenadas do nó 3 do elemento
        X3[1] = NOS[nos[2],1]    # Coordenadas do nó 3 do elemento
        X3[2] = NOS[nos[2],2]    # Coordenadas do nó 3 do elemento
        n[0]=0
        n[1]=0
        n[2]=0
        J=calc_vetnormal(X1,X2,X3,n)    # Vetor unitário normal ao elemento
        for pf in range(0,nnos): # Laço sobre os pontos fonte
            Xd[0] = NOS[pf,0]         # Coordenadas do ponto fonte
            Xd[1] = NOS[pf,1]         # Coordenadas do ponto fonte
            Xd[2] = NOS[pf,2]         # Coordenadas do ponto fonte
            if (pf==nos[0] or pf==nos[1] or pf==nos[2]): # Integração singular (o ponto fonte pertence ao elemento)
                if pf==nos[0]:   # O ponto fonte está no nó 1
                    ipf = 1               # Sinaliza que o ponto fonte está sobre o primeiro nó
                elif pf==nos[1]: # O ponto fonte está no nó 2
                    ipf = 2               # Sinaliza que o ponto fonte está sobre o segundo nó
                elif pf==nos[2]: # O ponto fonte está no nó 3
                    ipf = 3               # Sinaliza que o ponto fonte está sobre o terceiro nó
                g[0]=0
                g[1]=0
                g[2]=0
                calcula_Gs(X1,X2,X3,Xd,ipf,N1q,N2q,N3q,N4q,qsi_quad,w_quad,k,g)
                h[0]=0
                h[1]=0
                h[2]=0
            else: # Integração regular (o ponto fonte não pertence ao elemento)
                g[0]=0
                g[1]=0
                g[2]=0
                h[0]=0
                h[1]=0
                h[2]=0
            calcula_HeGns(X1,X2,X3,Xd,qsil,w1,n,k,J,N1,N2,N3,g,h)
            #printf("g[0] = %f8.5",g[0] )
            for nolocal in range(0,3):
                noglobal = ELEM[pc,nolocal]   # Índice da matriz global H
                H[pf,noglobal] += h[nolocal]
            G[pf,3*pc] = g[0]
            G[pf,3*pc+1] = g[1]
            G[pf,3*pc+2] = g[2]

    for m in range(0,nnos):
        H[m,m] = 0
        for l in range(0,nnos):
            if l != m:
                H[m,m] += - H[m,l]
    return H,G
#
```

Figura 31 – Cálculo Paralelo das Matrizes [H] e [G]

```
with nogil:

    for pc in prange(nelem, schedule='static', num_threads=ncores):
        #if(openmp.omp_get_thread_num()==0 and pc ==0):
        #    printf("Numero de threads\n")
        #    printf("%d\n",openmp.omp_get_num_threads())

    #    print(openmp.omp_get_num_threads())

        nos[0] = ELEM[pc,0]      # Nós que compõem o elemento
        nos[1] = ELEM[pc,1]      # Nós que compõem o elemento
        nos[2] = ELEM[pc,2]      # Nós que compõem o elemento
        X1[0] = NOS[nos[0],0]    # Coordenadas do nó 1 do elemento
        X1[1] = NOS[nos[0],1]    # Coordenadas do nó 1 do elemento
        X1[2] = NOS[nos[0],2]    # Coordenadas do nó 1 do elemento
        X2[0] = NOS[nos[1],0]    # Coordenadas do nó 2 do elemento
        X2[1] = NOS[nos[1],1]    # Coordenadas do nó 2 do elemento
        X2[2] = NOS[nos[1],2]    # Coordenadas do nó 2 do elemento
        X3[0] = NOS[nos[2],0]    # Coordenadas do nó 3 do elemento
        X3[1] = NOS[nos[2],1]    # Coordenadas do nó 3 do elemento
        X3[2] = NOS[nos[2],2]    # Coordenadas do nó 3 do elemento
        n[0]=0
        n[1]=0
        n[2]=0
        J=calc_vetnormal(X1,X2,X3,n)    # Vetor unitário normal ao elemento
        for pf in range(0,nnos): # Laço sobre os pontos fonte
            Xd[0] = NOS[pf,0]         # Coordenadas do ponto fonte
            Xd[1] = NOS[pf,1]         # Coordenadas do ponto fonte
            Xd[2] = NOS[pf,2]         # Coordenadas do ponto fonte
            if (pf==nos[0] or pf==nos[1] or pf==nos[2]): # Integração singular (o ponto fonte pertence ao elemento)
                if pf==nos[0]:   # O ponto fonte está no nó 1
                    ipf = 1              # Sinaliza que o ponto fonte está sobre o primeiro nó
                elif pf==nos[1]: # O ponto fonte está no nó 2
                    ipf = 2              # Sinaliza que o ponto fonte está sobre o segundo nó
                elif pf==nos[2]: # O ponto fonte está no nó 3
                    ipf = 3              # Sinaliza que o ponto fonte está sobre o terceiro nó
                g[0]=0
                g[1]=0
                g[2]=0
                calcula_Gs(X1,X2,X3,Xd,ipf,N1q,N2q,N3q,N4q,qsi_quad,w_quad,k,g)
                h[0]=0
                h[1]=0
                h[2]=0
            else: # Integração regular (o ponto fonte não pertence ao elemento)
                g[0]=0
                g[1]=0
                g[2]=0
                h[0]=0
                h[1]=0
                h[2]=0
                calcula_HeGns(X1,X2,X3,Xd,qsil,w1,n,k,J,N1,N2,N3,g,h)
            for nolocal in range(0,3):
                noglobal = ELEM[pc,nolocal]   # Índice da matriz global H
                H[pf,noglobal] += h[nolocal]
            G[pf,3*pc] = g[0]
            G[pf,3*pc+1] = g[1]
            G[pf,3*pc+2] = g[2]

for m in range(0,nnos):
    H[m,m] = 0
    for l in range(0,nnos):
        if l != m:
            H[m,m] += - H[m,l]
return H,G
```

Em um teste superficial, verificou-se que o código paralelizado teve ganhos de aceleração. O ganho de aceleração, por si só, não significa muita coisa pois de nada adianta chegar a uma resposta errada mais rapidamente, portanto, foi feito o teste de exatidão do código com o *PotLinear3D_teste.ipynb* e os resultados obtidos foram espúrios, com resultados que se alteram a cada iteração. Este resultado indica que há uma *Race Condition* no código.

Existem algumas formas de evitar as *race conditions*, como tornar as variáveis de iteração privadas para evitar compartilhamento de informações antes do fim de cada

divisão de carga de trabalho (*Chunk*) ou a criação de barreiras que forçam a sincronização das informações entre todas as *threads*. Seria trivial tornar as variáveis privadas em linguagem C/C++ utilizando a diretiva "***#pragma omp threadprivate*** *(lista de variáveis)*" ou declarando a variável dentro da região paralela. Aqui surge um problema pois não é possível declarar variáveis dentro da região paralela em Cython e as diretivas de OpenMP não funcionam de forma nativa. Apesar de ser fácil o uso do OpenMP em Cython, existem poucas implementações e pouca documentação sobre este assunto.

Uma possível solução foi a de substituir todos os argumentos do *loop* para uma função que só seria inicializada dentro da região paralela. Desta forma, a segunda tentativa de implementação ficou conforme ilustrado na figura 32 e a função utilizada é apresentada na figura 33.

Figura 32 – Cálculo Paralelo das Matrizes [H] e [G] - Tentativa 2

```
    #%% = = = = = CÁLCULO DOS ELEMENTOS DAS MATRIZES H e G = = = = =
    with nogil:
        for pc in prange(nelem, schedule='static', num_threads=ncores):
            calcula(pc,ELEM,NOS,k, N1q,N2q,N3q,N4q,N1,N2,N3,qsil,w1,qsi_quad,w_quad,npg_s,npg_r,H,G)
    for m in range(0,nnos):
        H[m,m] = 0
        for l in range(0,nnos):
            if l != m:
                H[m,m] = H[m,m] - H[m,l]
#
    return H,G
```

Após a mudança, o código passou nos testes de exatidão feitos pelo "*PotLinear3D_teste.ipynb*". O autor do código e responsável pela implementação da correção, professor Éder Lima, não sabe dizer ao certo por que esta mudança foi capaz de corrigir o erro. Uma possível explicação é que, como as variáveis estão sendo inicializadas dentro da região paralela, indiretamente por conta da nova função, estas se tornam variáveis privadas. Não haveria outra maneira de inicializar uma variável dentro da região paralela utilizando Cython.

Figura 33 – Função de Cálculo Paralelo das Matrizes [H] e [G]

```
#
@cython.boundscheck(False)
@cython.wraparound(False)
@cython.cdivision(True) # to avoid the exception checking
@cython.nonecheck(False)
cdef void  calcula(long pc, long[:,:] ELEM, double[:,:] NOS,double k,
                   double[:,:] N1q,double[:,:] N2q,double[:,:] N3q,
                   double[:,:] N4q,double[:,:] N1,double[:,:] N2,
                   double[:,:] N3,double[:] qsil,double[:] w1,
                   double[:] qsi_quad,double[:] w_quad,long npg_s,long npg_r,
                   double[:,:] H,double[:,:] G) nogil:
    cdef long nelem,nnos,i,l,m,noglobal,ipf
    cdef long[3] nos
    nelem = ELEM.shape[0]           # Número de elementos
    nnos = NOS.shape[0]             # Número de nós
    cdef double[3] X1,X2,X3,Xd,n,g,h
    cdef double J
    nos[0] = ELEM[pc,0]      # Nós que compõem o elemento
    nos[1] = ELEM[pc,1]      # Nós que compõem o elemento
    nos[2] = ELEM[pc,2]      # Nós que compõem o elemento
    X1[0] = NOS[nos[0],0]    # Coordenadas do nó 1 do elemento
    X1[1] = NOS[nos[0],1]    # Coordenadas do nó 1 do elemento
    X1[2] = NOS[nos[0],2]    # Coordenadas do nó 1 do elemento
    X2[0] = NOS[nos[1],0]    # Coordenadas do nó 2 do elemento
    X2[1] = NOS[nos[1],1]    # Coordenadas do nó 2 do elemento
    X2[2] = NOS[nos[1],2]    # Coordenadas do nó 2 do elemento
    X3[0] = NOS[nos[2],0]    # Coordenadas do nó 3 do elemento
    X3[1] = NOS[nos[2],1]    # Coordenadas do nó 3 do elemento
    X3[2] = NOS[nos[2],2]    # Coordenadas do nó 3 do elemento
    n[0]=0
    n[1]=0
    n[2]=0
    J=calc_vetnormal(X1,X2,X3,n)    # Vetor unitário normal ao elemento
    for pf in range(0,nnos): # Laço sobre os pontos fonte
        Xd[0] = NOS[pf,0]         # Coordenadas do ponto fonte
        Xd[1] = NOS[pf,1]         # Coordenadas do ponto fonte
        Xd[2] = NOS[pf,2]         # Coordenadas do ponto fonte

        if (pf==nos[0] or pf==nos[1] or pf==nos[2]): # Integração singular (o ponto fonte pertence ao elemento)
            if pf==nos[0]:   # O ponto fonte está no nó 1
                ipf = 1              # Sinaliza que o ponto fonte está sobre o primeiro nó
            elif pf==nos[1]: # O ponto fonte está no nó 2
                ipf = 2              # Sinaliza que o ponto fonte está sobre o segundo nó
            elif pf==nos[2]: # O ponto fonte está no nó 3
                ipf = 3              # Sinaliza que o ponto fonte está sobre o terceiro nó
            g[0]=0
            g[1]=0
            g[2]=0
            calcula_Gs(X1,X2,X3,Xd,ipf,N1q,N2q,N3q,N4q,qsi_quad,w_quad,k,g)
            h[0]=0
            h[1]=0
            h[2]=0
        else: # Integração regular (o ponto fonte não pertence ao elemento)
            g[0]=0
            g[1]=0
            g[2]=0

            h[0]=0
            h[1]=0
            h[2]=0
            calcula_HeGns(X1,X2,X3,Xd,qsil,w1,n,k,J,N1,N2,N3,g,h)
        for nolocal in range(0,3):
            noglobal = ELEM[pc,nolocal]   # Índice da matriz global H
            H[pf,noglobal] = H[pf,noglobal] + h[nolocal]

        G[pf,3*pc] = g[0]
        G[pf,3*pc+1] = g[1]
        G[pf,3*pc+2] = g[2]
```

6.2 Resultados e Discussão

Com o código funcionando corretamente, nesta sessão é feita a análise de seu desempenho conforme descrito na sessão 6.1.1. Os dados tratados são apresentados com precisão numérica de duas casas decimais na tabela 1. A tabela 2 traz informações sobre as três malhas utilizadas. Não foi possível fazer uma malha maior do que a 3 pois a memória do computador não suportava a operação.

Tabela 1 – Dados Obtidos

10 Iterações									
	Tempo serial (s) +- STD		Tempo n=2 (s) +- STD		Tempo n=4 (s) +- STD		Tempo n=6 (s) +- STD		Tempo n=8 (s) +- STD	
Malha 1	0,18	0,01	0,09	0,00	0,04	0,00	0,04	0,00	0,03	0,00
Malha 2	2,85	0,04	1,44	0,01	0,73	0,00	0,71	0,00	0,56	0,00
Malha 3	45,93	0,18	23,43	0,08	12,73	0,32	11,75	0,06	10,48	0,38

Tabela 2 – Informações Sobre as Malhas

	Nós	**Elementos**
Malha 1	**182**	**364**
Malha 2	**728**	**1456**
Malha 3	**2912**	**5824**

Algumas informações podem ser extraídas desses dados, são elas:

1. O tempo do código serial escala, entre as malhas, com um fator aproximado de 16, como era esperado, pois o problema tem complexidade N^2.
2. Os desvios padrão para as malhas 1 e 2 foram muito pequenos para serem significativos com precisão numérica de duas casas decimais. Desvios pequenos são um bom sinal pois o tempo de execução varia muito quando existem *bugs* no código.
3. Um ponto interessante a se notar é que o tempo de execução cai aproximadamente pela metade entre o código serial e para $n = 2$. Novamente o tempo se reduz pela metade entre $n = 2$ e $n = 4$, no entanto, não ocorre uma redução proporcional para $n = 6$ e $n = 8$. Este comportamento era esperado e faz sentido pois o processador da máquina que rodou as simulações só possui 4 núcleos físicos, e cada núcleo se divide em duas *threads*, então, existe sim um ganho mas não é o mesmo que com um núcleo físico diferente.

A segunda parte da análise visa observar o ganho de aceleração para cada caso específico.

Figura 34 – Aceleração para o caso $n = 2$

Figura 35 – Aceleração para o caso $n = 4$

Figura 36 – Aceleração para o caso $n = 6$

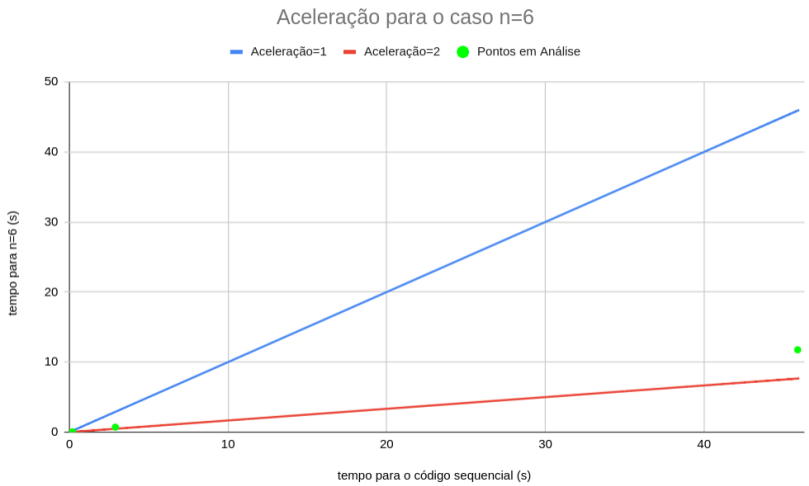
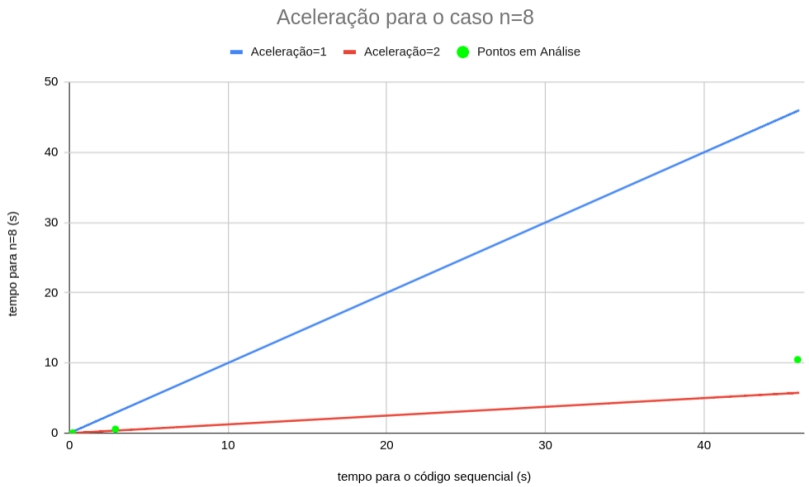


Figura 37 – Aceleração para o caso $n = 8$



As informações extraídas dos gráficos são:

1. Os pontos verdes estão praticamente em cima da linha vermelha nos casos $n = 2$ e $n = 4$. Isto significa que este trecho está quase tão rápido quanto pode ser.

2. Os casos $n = 6$ e $n = 8$ registraram acelerações menores, como era esperado.

Os resultados foram bons, mas, deve se ressaltar que a rotina de teste dos tempos, *PotLinear3D_tempos.py*, só executou a parte paralela do código. Não haveria beneficio nas partes sequenciais e portanto os ganhos de aceleração do *PotLinear3D.py* serão menores. Outro ponto relevante é que esta análise não traz informações sobre quão bom é o desempenho deste código quando comparado a outras linguagens de programação, apenas verificando o sucesso da implementação da paralelização.

7 CONCLUSÕES

O código *PotLinear3D.py* foi otimizado por executar partes em Cython e foi novamente otimizado por implementação da paralelização de trechos importantes. O tempo de programação, para as partes menos importantes, foi reduzido pela utilização da linguagem Python e os *Softwares Open Source* se mostraram de fácil manuseio, apesar de haver uma curva de aprendizado. O ganho da paralelização pode ser potencializado, em aplicações científicas, a um custo reduzido por conta dos servidores de processamento em nuvem que oferecem a opções de processadores modernos com muitos núcleos como é o caso do *Xeon Phi* da *Intel*, que apresenta versões com até 72 núcleos físicos e 4 *threads* por CPU.

Este trabalho não explorou as opções de processamento por MPI (*Message Passing Interface*). Para este caso, seria possível escalar ainda mais os ganhos de aceleração de simulações com grandes grupos de dados. Com o OpenMP e o Cython, foi demonstrado um ganho de aceleração da ordem de dezenas. Com uma posterior implementação de MPI, este ganho facilmente atinge a ordem de grandeza de centenas.

Outra conclusão que se tira é que o Python/Cython tornou muito fácil a paralelização, no entanto, existe grande dificuldade de encontrar material de consulta sobre OpenMP nesta linguagem e seu uso se resume a casos mais simples por não haverem muitas opções de customização. Uma estratégia para contornar este problema é a de programar as regiões paralelas em C/C++, as regiões seriais em Python e utilizar o Cython como *wrapper* para juntar os códigos em um único processo. Desta forma, é possível chegar a um ponto ótimo em que se alcança um bom desempenho com um tempo reduzido de programação.

Referências

AHRENS, J.; GEVECI, B.; LAW, C. Paraview: An end-user tool for large data visualization. *The visualization handbook*, Elsevier München, v. 717, 2005.

ALBUQUERQUE, E. L. *Introdução ao método dos elementos de contorno*. [S.l.], 2012.

AYACHIT, U. *The paraview guide: a parallel visualization application*. [S.l.]: Kitware, Inc., 2015.

CAI, X.; LANGTANGEN, H. P.; MOE, H. On the performance of the python programming language for serial and parallel scientific computations. *Scientific Programming*, Hindawi, v. 13, n. 1, p. 31–56, 2005.

CHAPMAN, B.; JOST, G.; PAS, R. V. D. *Using OpenMP: portable shared memory parallel programming*. [S.l.]: MIT press, 2008. v. 10.

FALCK, B.; FALCK, D.; COLLETTE, B. *Freecad [How-To]*. [S.l.]: Packt Publishing Ltd, 2012.

GEUZAINE, C.; REMACLE, J.-F. Gmsh: A 3-d finite element mesh generator with built-in pre-and post-processing facilities. *International journal for numerical methods in engineering*, Wiley Online Library, v. 79, n. 11, p. 1309–1331, 2009.

HAGER, G.; WELLEIN, G. *Introduction to high performance computing for scientists and engineers*. [S.l.]: CRC Press, 2010.

HERRON, P. *Learning Cython Programming*. [S.l.]: Packt Publishing Ltd, 2016.

KATSIKADELIS, J. T. *The boundary element method for engineers and scientists: theory and applications*. [S.l.]: Academic Press, 2016.

KONRAD, M. Vectorization and parallelization in python with numpy and pandas. 2018. Disponível em: <https://datascience.blog.wzb.eu/2018/02/02/vectorization-and-parallelization-in-python-with-numpy-and-pandas/>. Acesso em: 8 out. 2019.

LAMPORT, L. How to make a correct multiprocess program execute correctly on a multiprocessor. *IEEE Transactions on Computers SRC Research Report 96*, v. 46, 2016.

LOYOLA, F. M. d. Modelagem tridimensional de problemas potenciais usando o método dos elementos de contorno. 2017.

RAUBER, T.; RÜNGER, G. Parallel programming models. In: *Parallel Programming*. [S.l.]: Springer, 2010. p. 93–149.

SKUSE, B. The third pillar. *Physics World*, IOP Publishing, v. 32, n. 3, p. 40–43, mar 2019. Disponível em: <https://doi.org/10.1088%2F2058-7058%2F32%2F3%2F33>.

SMITH, K. W. *Cython: A Guide for Python Programmers*. [S.l.]: "O'Reilly Media, Inc.", 2015.

STACKOVERFLOW. Developer survey results 2019. 2019. Disponível em: <https://insights.stackoverflow.com/survey/2019#methodology>. Acesso em: 8 out. 2019.

WEST, J.; GALLAGHER, S. Challenges of open innovation: the paradox of firm investment in open-source software. *R&d Management*, Wiley Online Library, v. 36, n. 3, p. 319–331, 2006.

A Codigo PotLinear3D.py

```
# -*- coding: utf-8 -*-
"""
Universidade de Brasilia
Departamento de Engenharia Mecanica
Brasilia, setembro de 2019

Programa de elementos de contorno aplicado a problemas de conducao de
calor tri-dimensional sem fontes de calor concentradas

Tipo de elementos: Triangulares lineares continuos

Autores: Eder Lima de Albuquerque
         Gustavo Silva Vaz Gontijo (ggontijo@gmail.com)

ultima modificacao: 30/09/2019 - 09h02min
"""

#%% BIBLIOTECAS E ARQUIVOS NECESSARIOS

import meshio
import entrada_de_dados
import contorno
import sistema
import numpy as np
import time
import integracao2 as integ


#%% ENTRADA DE DADOS E PRE-PROCESSAMENTO

t_inicio = time.time()
print('\nPrograma iniciado.')

# Le o arquivo de entrada de dados
arquivo, CCSup, k = entrada_de_dados.dad1()
#arquivo, CCSup, k = entrada_de_dados.dad2()
#arquivo, CCSup, k = entrada_de_dados.dad3()

# Cria a malha
```

```
malha = meshio.read(arquivo + '.msh') # Le a malha do arquivo .msh

# Cria a matriz NOS a partir da malha
NOS = malha.points
  # NOS: Matriz [NNx3] que contem as coordenadas dos vertices da malha
      criada,
  #       onde NN e o numero de nos do problema

# Cria a matriz ELEM a partir da malha
ELEM = malha.cells_dict['triangle']
  # ELEM: Matriz [NEx3] que contem os numeros dos nos que formam cada
      elemento,
  #        onde NE e o numero de elementos do problema

print('Numero de nos:',NOS.shape[0])
print('Numero de elementos:',ELEM.shape[0])


# Cria a matriz de condicoes de contorno dos elementos
CDC = contorno.gera_elem_cdc(malha,CCSup)
  # CDC: Matriz [NEx3] que contem a condicao de contorno de cada no dos
  #       elementos

#%% MONTAGEM E SOLUCAO DO SISTEMA

# Calcula as matrizes H e G

# Calcula as matrizes H e G
npg_s = 8   # Numero de pontos de Gauss para a integracao singular
npg_r = 6   # Numero de pontos de Gauss para a integracao regular
qsi,w = np.polynomial.legendre.leggauss(npg_r); # Pontos e pesos de Gauss
    para a integracao regular (triangulo)
qsi_quad,w_quad = np.polynomial.legendre.leggauss(npg_s) # Pontos e pesos
    de Gauss para a integracao singular (quadrilatero)
nelem = ELEM.shape[0]             # Numero de elementos
nnos = NOS.shape[0]               # Numero de nos
H=np.zeros((nnos, nnos))
G=np.zeros((nnos,3*nelem))

t_gera_dados=time.time()-t_inicio

t_matriz_inicio = time.time()
print('Calculando as matrizes H e G.')

ncores = 4       #numero de threads
H[:,:],G[:,:] = integ.cal_HeG(NOS, ELEM, k,qsi,w,qsi_quad,w_quad,ncores) #
    Cython

```

```
83 t_matriz=time.time()-t_matriz_inicio # tempo para montagens das matrizes H
       e G
84   # H: Matriz [NNxNN] que contem o resultado da integracao de q* no
         contorno
85   # G: Matriz [NNx3NE] que contem o resultado da integracao de T* no
         contorno
86 print('Aplicando as condicoes de contorno.')
87 t_aplica_cdc_inicio = time.time()
88
89 A, b, T_pr = sistema.aplica_cdc(G, H, NOS, ELEM, CDC)
90 t_aplica_cdc = time.time()-t_aplica_cdc_inicio
91
92   # A: Matriz [NNxNN] contendo colunas de H e G
93   # b: Vetor [NNx1] resultante da multiplicacao de N colunas de H e G pelas
94   #    CDC's conhecidas
95 print('Resolvendo o sistema linear.')
96
97 t_sistema_inicio = time.time()
98
99 # Resolve o sistema de equacoes A.x = b
100 x = np.linalg.solve(A, b)
101   # x: Vetor [NNx1] que contem os termos calculados (antes desconhecidos)
         de
102   #    temperatura e fluxo
103
104 t_sistema=time.time()-t_sistema_inicio # tempo para montagens das matrizes
       H e G
105
106
107 t_ordena_inicio=time.time()
108 # Separa os valores de temperatura e fluxo
109 print('Separando as variaveis.')
110
111 T, q = sistema.monta_Teq(NOS, ELEM, CDC, x, T_pr)
112   # T: Vetor [NNx1] que contem os valores de temperatura calculados
113   # q: Vetor [3NEx1] que contem os valores de fluxo calculados
114 t_ordena=time.time()-t_ordena_inicio
115
116 print('Gerando o arquivo de pos-processamento.')
117
118 t_saida_inicio=time.time()
119
120 # Calcula os valores de temperatura no centroide do elemento
121 T_centroide = contorno.calcTcentroide(T,ELEM,NOS)
122
123 # Mostra os valores de temperatura no mapa de cor
124 malha.cell_data_dict['gmsh:physical']['Temp_med']=T_centroide
125 malha.point_data={"Temperatura":T}
126 meshio.write(arquivo + '.vtk', malha)
```

```
127
128 t_saida=time.time()-t_saida_inicio
129
130
131 print('Tempo para ler e gerar os dados iniciais:',t_gera_dados)
132 print('Tempo para montagem das matrizes H e G:',t_matriz)
133 print('Tempo para aplicas as condicoes de contorno:',t_aplica_cdc)
134 print('Tempo para resolver o sistema linear:',t_sistema)
135 print('Tempo para ordenar os dados:',t_ordena)
136 print('Tempo para gerar o arquivo de pos-processamento:',t_saida)
137 print('Tempo de processamento:',time.time()-t_inicio,'s.\nPrograma
    finalizado.')
```

B Codigo PotLinear3D_teste.py

```
# -*- coding: utf-8 -*-
"""PotLinear3D_teste.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1ZtGwttcYnix9IswvaeOSaMFgnjbLYN9R
"""

from google.colab import drive
drive.mount('/content/drive')

!pip install meshio

# In order to compile pyx files, in the terminal use the command:
!python setup.py build_ext -i
!python setup2.py build_ext -i
!python setup3.py build_ext -i
!python setup4.py build_ext -i

import contorno
import sistema
import numpy as np
import integracao as integ
import integracao2 as integ2
import integracao3 as integ3
import integracao4 as integ4

NOS =np.array([[ 0.  , 0.,   0.],
   [1. ,  0. ,  0.],
   [0.,   1 ,   0],
   [1  ,  1 ,   0],
   [0  ,  0 ,   1],
   [1   , 0 ,   1],
   [0  ,  1 ,   1],
   [1   , 1  ,  1]])
# A matriz ELEM tem 4 colunas e o numero de linhas e igual ao numero de
#   elementos, ou seja de triangulos. Neste caso sao 12 elementow
```

```
# ELEM = [ numero do elemento, no 1, no2, no 3]
ELEM =np.array([[1   ,   4   ,   2],
    [1  ,    3    , 4],
    [1  ,    6   ,   5],
    [1  ,    2   ,   6],
    [2  ,    8  ,    6],
    [2  ,    4  ,    8],
    [3  ,    8  ,    4],
    [3   ,   7   ,   8],
    [1   ,   7  ,    3],
    [1   ,   5   ,   7],
    [5  ,    8   ,   7],
    [5   ,   6   ,   8]])-1

CDC=np.array([[0.,  0.,  0.,  0.],
        [0.,  0.,  0.,  0.],
        [1.,  0.,  0.,  0.],
        [1.,  0.,  0.,  0.],
        [1.,  0.,  0.,  0.],
        [1.,  0.,  0.,  0.],
        [1.,  0.,  0.,  0.],
        [1.,  0.,  0.,  0.],
        [1.,  0.,  0.,  0.],
        [1.,  0.,  0.,  0.],
        [0.,  1.,  1.,  1.],
        [0.,  1.,  1.,  1.]])
k=1.
print('Numero de nos:',NOS.shape[0])
print('Numero de elementos:',ELEM.shape[0])
npg_s = 8   # Numero de pontos de Gauss para a integracao singular
npg_r = 6   # Numero de pontos de Gauss para a integracao regular
qsi,w = np.polynomial.legendre.leggauss(npg_r) # Pontos e pesos de Gauss
    para a integracao regular (triangulo)
qsi_quad,w_quad = np.polynomial.legendre.leggauss(npg_s) # Pontos e pesos
    de Gauss para a integracao singular (quadrilatero)
nelem = ELEM.shape[0]             # Numero de elementos
nnos = NOS.shape[0]
H=np.zeros((nnos, nnos))
G=np.zeros((nnos,3*nelem))
ncores = 2                        # Numero de nucleos para o integ2
# python setup.py build_ext -i

H[:,:],G[:,:] = integ.cal_HeG(NOS, ELEM, k,qsi,w,qsi_quad,w_quad) # Cython
#H[:,:],G[:,:] = integ2.cal_HeG(NOS, ELEM, k,qsi,w,qsi_quad,w_quad, ncores)
    # Cython
#H[:,:],G[:,:] = integ3.cal_HeG(NOS, ELEM, k,qsi,w,qsi_quad,w_quad) #
    Cython
#H[:,:],G[:,:] = integ4.cal_HeG(NOS, ELEM, k,qsi,w,qsi_quad,w_quad, ncores)
    # Cython
```

```
83
84 A, b, T_pr = sistema.aplica_cdc(G, H, NOS, ELEM, CDC)
85 x = np.linalg.solve(A, b)
86 T, q = sistema.monta_Teq(NOS, ELEM, CDC, x, T_pr)
87 print("x = ",x)
88 print("T: ",T)
89 print("q: ",q)
```

C Codigo PotLinear3D_time.py

```
# -*- coding: utf-8 -*-
"""
Universidade de Brasilia
Departamento de Engenharia Mecanica
Brasilia, setembro de 2019

Programa de elementos de contorno aplicado a problemas de conducao de
calor tri-dimensional sem fontes de calor concentradas

Tipo de elementos: Triangulares lineares continuos

Autores: Eder Lima de Albuquerque
         Gustavo Silva Vaz Gontijo (ggontijo@gmail.com)

Ultima modificacao: 30/09/2019 - 09h02min
"""

#%% BIBLIOTECAS E ARQUIVOS NECESSARIOS

import meshio
import entrada_de_dados
import contorno
import sistema
import numpy as np
import time
import integracao as integ
import integracao2 as integ2
import integracao3 as integ3


it=10         #numero de iteracoes
casos=4      #numero de casos com qtd. de nucleo diferente

t_matriz_1 = np.zeros((casos+1, it))
t_matriz_2 = np.zeros((casos+1, it))
t_matriz_3 = np.zeros((casos+1, it))

print (t_matriz_1)
print ('Numero de iteracoes:', it)
```

```
40
41 #%% ENTRADA DE DADOS E PRE-PROCESSAMENTO
42
43 t_inicio = time.time()
44 print('\nPrograma iniciado.')
45
46 # Le o arquivo de entrada de dados
47 arquivo, CCSup, k = entrada_de_dados.dad1()
48 print('Utilizando a malha 1')
49
50 #arquivo, CCSup, k = entrada_de_dados.dad2()
51 #print('Utilizando a malha 2')
52
53 #arquivo, CCSup, k = entrada_de_dados.dad3()
54 #print('Utilizando a malha 3')
55
56 # Cria a malha
57 malha = meshio.read(arquivo + '.msh') # Le a malha do arquivo .msh
58
59 # Cria a matriz NOS a partir da malha
60 NOS = malha.points
61   # NOS: Matriz [NNx3] que contem as coordenadas dos vertices da malha
         criada,
62   #       onde NN e o numero de nos do problema
63
64 # Cria a matriz ELEM a partir da malha
65 ELEM = malha.cells_dict['triangle']
66   # ELEM: Matriz [NEx3] que contem os numeros dos nos que formam cada
         elemento,
67   #        onde NE e o numero de elementos do problema
68
69 print('Numero de nos:',NOS.shape[0])
70 print('Numero de elementos:',ELEM.shape[0])
71
72
73 # Cria a matriz de condicoes de contorno dos elementos
74 CDC = contorno.gera_elem_cdc(malha,CCSup)
75   # CDC: Matriz [NEx3] que contem a condicao de contorno de cada no dos
76   #       elementos
77
78 #%% MONTAGEM E SOLUCAO DO SISTEMA
79
80 # Calcula as matrizes H e G
81
82 # Calcula as matrizes H e G
83 npg_s = 8   # Numero de pontos de Gauss para a integracao singular
84 npg_r = 6   # Numero de pontos de Gauss para a integracao regular
85 qsi,w = np.polynomial.legendre.leggauss(npg_r); # Pontos e pesos de Gauss
       para a integracao regular (triangulo)
```

```
86 qsi_quad,w_quad = np.polynomial.legendre.leggauss(npg_s) # Pontos e pesos
       de Gauss para a integracao singular (quadrilatero)
87 nelem = ELEM.shape[0]              # Numero de elementos
88 nnos = NOS.shape[0]                # Numero de nos
89 H=np.zeros((nnos, nnos))
90 G=np.zeros((nnos,3*nelem))
91
92 print('Calculando codigo serial')
93 for j in range (it):
94   t_gera_dados=time.time()-t_inicio
95
96   t_matriz_inicio = time.time()
97
98   H[:,:],G[:,:] = integ3.cal_HeG(NOS, ELEM, k,qsi,w,qsi_quad,w_quad) #
         Cython
99
100  t_matriz_1[0, j-1]=time.time()-t_matriz_inicio # tempo para montagens das
         matrizes H e G
101    # H: Matriz [NNxNN] que contem o resultado da integracao de q* no
         contorno
102    # G: Matriz [NNx3NE] que contem o resultado da integracao de T* no
         contorno
103
104
105 for i in range (casos):
106
107  ncores = i*2+2       #numero de threads
108  #%% ENTRADA DE DADOS E PRE-PROCESSAMENTO
109
110  t_inicio = time.time()
111  print('\nPrograma iniciado.')
112
113  # Le o arquivo de entrada de dados
114  arquivo, CCSup, k = entrada_de_dados.dad1()
115  print('Utilizando a malha 1')
116
117  #arquivo, CCSup, k = entrada_de_dados.dad2()
118  #print('Utilizando a malha 2')
119
120  #arquivo, CCSup, k = entrada_de_dados.dad3()
121  #print('Utilizando a malha 3')
122
123  # Cria a malha
124  malha = meshio.read(arquivo + '.msh') # Le a malha do arquivo .msh
125
126  # Cria a matriz NOS a partir da malha
127  NOS = malha.points
128    # NOS: Matriz [NNx3] que contem as coordenadas dos vertices da malha
         criada,
```

```
129     #       onde NN e o numero de nos do problema
130
131   # Cria a matriz ELEM a partir da malha
132   ELEM = malha.cells_dict['triangle']
133     # ELEM: Matriz [NEx3] que contem os numeros dos nos que formam cada
          elemento,
134     #       onde NE e o numero de elementos do problema
135
136   print('Numero de nos:',NOS.shape[0])
137   print('Numero de elementos:',ELEM.shape[0])
138
139
140   # Cria a matriz de condicoes de contorno dos elementos
141   CDC = contorno.gera_elem_cdc(malha,CCSup)
142     # CDC: Matriz [NEx3] que contem a condicao de contorno de cada no dos
143     #       elementos
144
145   #%% MONTAGEM E SOLUCAO DO SISTEMA
146
147   # Calcula as matrizes H e G
148
149   # Calcula as matrizes H e G
150   npg_s = 8   # Numero de pontos de Gauss para a integracao singular
151   npg_r = 6   # Numero de pontos de Gauss para a integracao regular
152   qsi,w = np.polynomial.legendre.leggauss(npg_r); # Pontos e pesos de Gauss
          para a integracao regular (triangulo)
153   qsi_quad,w_quad = np.polynomial.legendre.leggauss(npg_s) # Pontos e pesos
          de Gauss para a integracao singular (quadrilatero)
154   nelem = ELEM.shape[0]             # Numero de elementos
155   nnos = NOS.shape[0]               # Numero de nos
156   H=np.zeros((nnos, nnos))
157   G=np.zeros((nnos,3*nelem))
158
159   print('Calculando para n=',ncores)
160   for j in range (it):
161     t_gera_dados=time.time()-t_inicio
162
163     t_matriz_inicio = time.time()
164
165     H[:,:],G[:,:] = integ.cal_HeG(NOS, ELEM, k,qsi,w,qsi_quad,w_quad,ncores
          ) # Cython
166
167     t_matriz_1[i+1, j-1]=time.time()-t_matriz_inicio # tempo para montagens
          das matrizes H e G
168       # H: Matriz [NNxNN] que contem o resultado da integracao de q* no
            contorno
169       # G: Matriz [NNx3NE] que contem o resultado da integracao de T* no
            contorno
170
```

```
171 t_serial_med = np.mean(t_matriz_1[0,:])
172 t_serial_std = np.std(t_matriz_1[0,:])
173
174 t_n2_med = np.mean(t_matriz_1[1,:])
175 t_n2_std = np.std(t_matriz_1[1,:])
176
177 t_n4_med = np.mean(t_matriz_1[2,:])
178 t_n4_std = np.std(t_matriz_1[2,:])
179
180 t_n6_med = np.mean(t_matriz_1[3,:])
181 t_n6_std = np.std(t_matriz_1[3,:])
182
183 t_n8_med = np.mean(t_matriz_1[4,:])
184 t_n8_std = np.std(t_matriz_1[4,:])
185
186 print('\n tempo serial medio =', np.round(t_serial_med,2),'  desvio padrao
    =', np.round(t_serial_std,2))
187 print('\n tempo medio 2 nucleos =', np.round(t_n2_med,2),'  desvio padrao =
    ', np.round(t_n2_std,2))
188 print('\n tempo medio 4 nucleos =', np.round(t_n4_med,2),'  desvio padrao =
    ', np.round(t_n4_std,2))
189 print('\n tempo medio 6 nucleos =', np.round(t_n6_med,2),'  desvio padrao =
    ', np.round(t_n6_std,2))
190 print('\n tempo medio 8 nucleos =', np.round(t_n8_med,2),'  desvio padrao =
    ', np.round(t_n8_std,2))
191
192
193 ##########################################################################
194
195 #%% ENTRADA DE DADOS E PRE-PROCESSAMENTO
196
197 t_inicio = time.time()
198 print('\nPrograma iniciado.')
199
200 # Le o arquivo de entrada de dados
201 #arquivo, CCSup, k = entrada_de_dados.dad1()
202 #print('Utilizando a malha 1')
203
204 arquivo, CCSup, k = entrada_de_dados.dad2()
205 print('Utilizando a malha 2')
206
207 #arquivo, CCSup, k = entrada_de_dados.dad3()
208 #print('Utilizando a malha 3')
209
210 # Cria a malha
211 malha = meshio.read(arquivo + '.msh') # Le a malha do arquivo .msh
212
213 # Cria a matriz NOS a partir da malha
214 NOS = malha.points
```

```
215   # NOS: Matriz [NNx3] que contem as coordenadas dos vertices da malha
          criada,
216   #       onde NN e o numero de nos do problema
217
218 # Cria a matriz ELEM a partir da malha
219 ELEM = malha.cells_dict['triangle']
220   # ELEM: Matriz [NEx3] que contem os numeros dos nos que formam cada
          elemento,
221   #        onde NE e o numero de elementos do problema
222
223 print('Numero de nos:',NOS.shape[0])
224 print('Numero de elementos:',ELEM.shape[0])
225
226
227 # Cria a matriz de condicoes de contorno dos elementos
228 CDC = contorno.gera_elem_cdc(malha,CCSup)
229   # CDC: Matriz [NEx3] que contem a condicao de contorno de cada no dos
230   #       elementos
231
232 #%% MONTAGEM E SOLUCAO DO SISTEMA
233
234 # Calcula as matrizes H e G
235
236 # Calcula as matrizes H e G
237 npg_s = 8   # Numero de pontos de Gauss para a integracao singular
238 npg_r = 6   # Numero de pontos de Gauss para a integracao regular
239 qsi,w = np.polynomial.legendre.leggauss(npg_r); # Pontos e pesos de Gauss
        para a integracao regular (triangulo)
240 qsi_quad,w_quad = np.polynomial.legendre.leggauss(npg_s) # Pontos e pesos
        de Gauss para a integracao singular (quadrilatero)
241 nelem = ELEM.shape[0]              # Numero de elementos
242 nnos = NOS.shape[0]                # Numero de nos
243 H=np.zeros((nnos, nnos))
244 G=np.zeros((nnos,3*nelem))
245
246 print('Calculando codigo serial')
247 for j in range (it):
248   t_gera_dados=time.time()-t_inicio
249
250   t_matriz_inicio = time.time()
251
252   H[:,:],G[:,:] = integ3.cal_HeG(NOS, ELEM, k,qsi,w,qsi_quad,w_quad) #
          Cython
253
254   t_matriz_1[0, j-1]=time.time()-t_matriz_inicio # tempo para montagens das
          matrizes H e G
255     # H: Matriz [NNxNN] que contem o resultado da integracao de q* no
            contorno
```

```
256     # G: Matriz [NNx3NE] que contem o resultado da integracao de T* no
            contorno
257
258
259 for i in range (casos):
260
261     ncores = i*2+2      #numero de threads
262     #%% ENTRADA DE DADOS E PRE-PROCESSAMENTO
263
264     t_inicio = time.time()
265     print('\nPrograma iniciado.')
266
267     # Le o arquivo de entrada de dados
268     #arquivo, CCSup, k = entrada_de_dados.dad1()
269     #print('Utilizando a malha 1')
270
271     arquivo, CCSup, k = entrada_de_dados.dad2()
272     print('Utilizando a malha 2')
273
274     #arquivo, CCSup, k = entrada_de_dados.dad3()
275     #print('Utilizando a malha 3')
276
277     # Cria a malha
278     malha = meshio.read(arquivo + '.msh') # Le a malha do arquivo .msh
279
280     # Cria a matriz NOS a partir da malha
281     NOS = malha.points
282       # NOS: Matriz [NNx3] que contem as coordenadas dos vertices da malha
              criada,
283       #       onde NN e o numero de nos do problema
284
285     # Cria a matriz ELEM a partir da malha
286     ELEM = malha.cells_dict['triangle']
287       # ELEM: Matriz [NEx3] que contem os numeros dos nos que formam cada
              elemento,
288       #        onde NE e o numero de elementos do problema
289
290     print('Numero de nos:',NOS.shape[0])
291     print('Numero de elementos:',ELEM.shape[0])
292
293
294     # Cria a matriz de condicoes de contorno dos elementos
295     CDC = contorno.gera_elem_cdc(malha,CCSup)
296       # CDC: Matriz [NEx3] que contem a condicao de contorno de cada no dos
297       #       elementos
298
299     #%% MONTAGEM E SOLUCAO DO SISTEMA
300
301     # Calcula as matrizes H e G
```

```
302
303   # Calcula as matrizes H e G
304   npg_s = 8    # Numero de pontos de Gauss para a integracao singular
305   npg_r = 6    # Numero de pontos de Gauss para a integracao regular
306   qsi,w = np.polynomial.legendre.leggauss(npg_r); # Pontos e pesos de Gauss
          para a integracao regular (triangulo)
307   qsi_quad,w_quad = np.polynomial.legendre.leggauss(npg_s) # Pontos e pesos
          de Gauss para a integracao singular (quadrilatero)
308   nelem = ELEM.shape[0]              # Numero de elementos
309   nnos = NOS.shape[0]                # Numero de nos
310   H=np.zeros((nnos, nnos))
311   G=np.zeros((nnos,3*nelem))
312
313   print('Calculando para n=',ncores)
314   for j in range (it):
315     t_gera_dados=time.time()-t_inicio
316
317     t_matriz_inicio = time.time()
318
319     H[:,:],G[:,:] = integ.cal_HeG(NOS, ELEM, k,qsi,w,qsi_quad,w_quad,ncores
          ) # Cython
320
321     t_matriz_1[i+1, j-1]=time.time()-t_matriz_inicio # tempo para montagens
          das matrizes H e G
322       # H: Matriz [NNxNN] que contem o resultado da integracao de q* no
            contorno
323       # G: Matriz [NNx3NE] que contem o resultado da integracao de T* no
            contorno
324
325 t_serial_med = np.mean(t_matriz_1[0,:])
326 t_serial_std = np.std(t_matriz_1[0,:])
327
328 t_n2_med = np.mean(t_matriz_1[1,:])
329 t_n2_std = np.std(t_matriz_1[1,:])
330
331 t_n4_med = np.mean(t_matriz_1[2,:])
332 t_n4_std = np.std(t_matriz_1[2,:])
333
334 t_n6_med = np.mean(t_matriz_1[3,:])
335 t_n6_std = np.std(t_matriz_1[3,:])
336
337 t_n8_med = np.mean(t_matriz_1[4,:])
338 t_n8_std = np.std(t_matriz_1[4,:])
339
340 print('\n tempo serial medio =', np.round(t_serial_med,2),'  desvio padrao
      =', np.round(t_serial_std,2))
341 print('\n tempo medio 2 nucleos =', np.round(t_n2_med,2),'  desvio padrao =
      ', np.round(t_n2_std,2))
```

```
342 print('\n tempo medio 4 nucleos =', np.round(t_n4_med,2),'  desvio padrao =
    ', np.round(t_n4_std,2))
343 print('\n tempo medio 6 nucleos =', np.round(t_n6_med,2),'  desvio padrao =
    ', np.round(t_n6_std,2))
344 print('\n tempo medio 8 nucleos =', np.round(t_n8_med,2),'  desvio padrao =
    ', np.round(t_n8_std,2))
345
346
347 #
    ##############################################################################
348
349 #%% ENTRADA DE DADOS E PRE-PROCESSAMENTO
350
351 t_inicio = time.time()
352 print('\nPrograma iniciado.')
353
354 # Le o arquivo de entrada de dados
355 #arquivo, CCSup, k = entrada_de_dados.dad1()
356 #print('Utilizando a malha 1')
357
358 #arquivo, CCSup, k = entrada_de_dados.dad2()
359 #print('Utilizando a malha 2')
360
361 arquivo, CCSup, k = entrada_de_dados.dad3()
362 print('Utilizando a malha 3')
363
364 # Cria a malha
365 malha = meshio.read(arquivo + '.msh') # Le a malha do arquivo .msh
366
367 # Cria a matriz NOS a partir da malha
368 NOS = malha.points
369   # NOS: Matriz [NNx3] que contem as coordenadas dos vertices da malha
        criada,
370   #       onde NN e o numero de nos do problema
371
372 # Cria a matriz ELEM a partir da malha
373 ELEM = malha.cells_dict['triangle']
374   # ELEM: Matriz [NEx3] que contem os numeros dos nos que formam cada
        elemento,
375   #        onde NE e o numero de elementos do problema
376
377 print('Numero de nos:',NOS.shape[0])
378 print('Numero de elementos:',ELEM.shape[0])
379
380
381 # Cria a matriz de condicoes de contorno dos elementos
382 CDC = contorno.gera_elem_cdc(malha,CCSup)
383   # CDC: Matriz [NEx3] que contem a condicao de contorno de cada no dos
```

```
384   #       elementos
385
386 #%% MONTAGEM E SOLUCAO DO SISTEMA
387
388 # Calcula as matrizes H e G
389
390 # Calcula as matrizes H e G
391 npg_s = 8    # Numero de pontos de Gauss para a integracao singular
392 npg_r = 6    # Numero de pontos de Gauss para a integracao regular
393 qsi,w = np.polynomial.legendre.leggauss(npg_r); # Pontos e pesos de Gauss
        para a integracao regular (triangulo)
394 qsi_quad,w_quad = np.polynomial.legendre.leggauss(npg_s) # Pontos e pesos
        de Gauss para a integracao singular (quadrilatero)
395 nelem = ELEM.shape[0]              # Numero de elementos
396 nnos = NOS.shape[0]                # Numero de nos
397 H=np.zeros((nnos, nnos))
398 G=np.zeros((nnos,3*nelem))
399
400 print('Calculando codigo serial')
401 for j in range (it):
402   t_gera_dados=time.time()-t_inicio
403
404   t_matriz_inicio = time.time()
405
406   H[:,:],G[:,:] = integ3.cal_HeG(NOS, ELEM, k,qsi,w,qsi_quad,w_quad) #
        Cython
407
408   t_matriz_1[0, j-1]=time.time()-t_matriz_inicio # tempo para montagens das
        matrizes H e G
409     # H: Matriz [NNxNN] que contem o resultado da integracao de q* no
          contorno
410     # G: Matriz [NNx3NE] que contem o resultado da integracao de T* no
          contorno
411
412
413 for i in range (casos):
414
415   ncores = i*2+2       #numero de threads
416   #%% ENTRADA DE DADOS E PRE-PROCESSAMENTO
417
418   t_inicio = time.time()
419   print('\nPrograma iniciado.')
420
421   # Le o arquivo de entrada de dados
422   #arquivo, CCSup, k = entrada_de_dados.dad1()
423   #print('Utilizando a malha 1')
424
425   #arquivo, CCSup, k = entrada_de_dados.dad2()
426   #print('Utilizando a malha 2')
```

```
427
428  arquivo, CCSup, k = entrada_de_dados.dad3()
429  print('Utilizando a malha 3')
430
431  # Cria a malha
432  malha = meshio.read(arquivo + '.msh') # Le a malha do arquivo .msh
433
434  # Cria a matriz NOS a partir da malha
435  NOS = malha.points
436    # NOS: Matriz [NNx3] que contem as coordenadas dos vertices da malha
         criada,
437    #       onde NN e o numero de nos do problema
438
439  # Cria a matriz ELEM a partir da malha
440  ELEM = malha.cells_dict['triangle']
441    # ELEM: Matriz [NEx3] que contem os numeros dos nos que formam cada
         elemento,
442    #        onde NE e o numero de elementos do problema
443
444  print('Numero de nos:',NOS.shape[0])
445  print('Numero de elementos:',ELEM.shape[0])
446
447
448  # Cria a matriz de condicoes de contorno dos elementos
449  CDC = contorno.gera_elem_cdc(malha,CCSup)
450    # CDC: Matriz [NEx3] que contem a condicao de contorno de cada no dos
451    #       elementos
452
453  #%% MONTAGEM E SOLUCAO DO SISTEMA
454
455  # Calcula as matrizes H e G
456
457  # Calcula as matrizes H e G
458  npg_s = 8   # Numero de pontos de Gauss para a integracao singular
459  npg_r = 6   # Numero de pontos de Gauss para a integracao regular
460  qsi,w = np.polynomial.legendre.leggauss(npg_r); # Pontos e pesos de Gauss
       para a integracao regular (triangulo)
461  qsi_quad,w_quad = np.polynomial.legendre.leggauss(npg_s) # Pontos e pesos
       de Gauss para a integracao singular (quadrilatero)
462  nelem = ELEM.shape[0]              # Numero de elementos
463  nnos = NOS.shape[0]                # Numero de nos
464  H=np.zeros((nnos, nnos))
465  G=np.zeros((nnos,3*nelem))
466
467  print('Calculando para n=',ncores)
468  for j in range(it):
469    t_gera_dados=time.time()-t_inicio
470
471    t_matriz_inicio = time.time()
```

```
472
473     H[:,:],G[:,:] = integ.cal_HeG(NOS, ELEM, k,qsi,w,qsi_quad,w_quad,ncores
            ) # Cython
474
475     t_matriz_1[i+1, j-1]=time.time()-t_matriz_inicio # tempo para montagens
            das matrizes H e G
476       # H: Matriz [NNxNN] que contem o resultado da integracao de q* no
              contorno
477       # G: Matriz [NNx3NE] que contem o resultado da integracao de T* no
              contorno
478
479 t_serial_med = np.mean(t_matriz_1[0,:])
480 t_serial_std = np.std(t_matriz_1[0,:])
481
482 t_n2_med = np.mean(t_matriz_1[1,:])
483 t_n2_std = np.std(t_matriz_1[1,:])
484
485 t_n4_med = np.mean(t_matriz_1[2,:])
486 t_n4_std = np.std(t_matriz_1[2,:])
487
488 t_n6_med = np.mean(t_matriz_1[3,:])
489 t_n6_std = np.std(t_matriz_1[3,:])
490
491 t_n8_med = np.mean(t_matriz_1[4,:])
492 t_n8_std = np.std(t_matriz_1[4,:])
493
494 print('\n tempo serial medio =', np.round(t_serial_med,2),'  desvio padrao
        =', np.round(t_serial_std,2))
495 print('\n tempo medio 2 nucleos =', np.round(t_n2_med,2),'  desvio padrao =
        ', np.round(t_n2_std,2))
496 print('\n tempo medio 4 nucleos =', np.round(t_n4_med,2),'  desvio padrao =
        ', np.round(t_n4_std,2))
497 print('\n tempo medio 6 nucleos =', np.round(t_n6_med,2),'  desvio padrao =
        ', np.round(t_n6_std,2))
498 print('\n tempo medio 8 nucleos =', np.round(t_n8_med,2),'  desvio padrao =
        ', np.round(t_n8_std,2))
```

D Codigo integracao.pyx

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Wed Oct  2 11:39:46 2019

@author: eder
"""
cimport cython
cimport openmp
from cython.parallel cimport prange
from cython.parallel cimport parallel



import numpy as np
from libc.math cimport log
from libc.math cimport atan2
from libc.math cimport sqrt
from libc.math cimport asinh
from libc.stdio cimport printf

import cython

@cython.boundscheck(False)
@cython.wraparound(False)
@cython.cdivision(True) # to avoid the exception checking
@cython.nonecheck(False)


def cal_HeG(double[:,:] NOS, long[:,:] ELEM, double k, double[:] qsil, double
    [:] w1, double[:] qsi_quad, double[:] w_quad, int ncores):
    cdef long nelem, nnos, npg_s, npg_r, pc
    nelem = ELEM.shape[0]              # Numero de elementos
    nnos = NOS.shape[0]                # Numero de nos
    cdef double[:,:] H,G,N1,N2,N3,N1q,N2q,N3q,N4q
    H=np.zeros((nnos, nnos))
    G=np.zeros((nnos,3*nelem))
    ## = = = = = PONTOS E PESOS DE GAUSS = = = = =
```

```
38      npg_s = qsi_quad.shape[0]    # Numero de pontos de Gauss para a
            integracao singular
39      npg_r = qsil.shape[0]    # Numero de pontos de Gauss para a integracao
            regular
40      for i in range(npg_r):
41          qsil[i] = 0.5*(qsil[i] + 1)                                  # Converte
                os pontos de Gauss para o intervalo [0, 1]
42          w1[i] = w1[i]*0.5                                            # Converte
                os pesos de Gauss para o intervalo [0, 1]
43  #   %% = = = = = FUNCOES DE FORMA = = = = =
44      N1q = np.zeros((npg_s,npg_s))
45      N2q = np.zeros((npg_s,npg_s))
46      N3q = np.zeros((npg_s,npg_s))
47      N4q = np.zeros((npg_s,npg_s))
48
49      N1 = np.zeros((npg_r,npg_r))
50      N2 = np.zeros((npg_r,npg_r))
51      N3 = np.zeros((npg_r,npg_r))
52
53      for l in range(0,npg_r):
54          for m in range(0,npg_r):
55              N1[l,m] = (1-qsil[m])*qsil[l] # qsi escrito como qsi_linha e
                    eta
56              N2[l,m] = qsil[m]
57              N3[l,m] = 1-N1[l,m]-N2[l,m]
58
59      for l in range(0,npg_s):
60          for m in range(0,npg_s):
61              N1q[l,m] = (1./4.)*(1. - qsi_quad[l])*(1. - qsi_quad[m])
62              N2q[l,m] = (1./4.)*(1. + qsi_quad[l])*(1. - qsi_quad[m])
63              N3q[l,m] = (1./4.)*(1. + qsi_quad[l])*(1. + qsi_quad[m])
64              N4q[l,m] = (1./4.)*(1. - qsi_quad[l])*(1. + qsi_quad[m])
65
66
67      #%% = = = = = CALCULO DOS ELEMENTOS DAS MATRIZES H e G = = = = =
68      with nogil:
69          for pc in prange(nelem, schedule='static', num_threads=ncores):
70              calcula(pc,ELEM,NOS,k, N1q,N2q,N3q,N4q,N1,N2,N3,qsil,w1,
                    qsi_quad,w_quad,npg_s,npg_r,H,G)
71      for m in range(0,nnos):
72          H[m,m] = 0
73          for l in range(0,nnos):
74              if l != m:
75                  H[m,m] = H[m,m] - H[m,l]
76  #
77      return H,G
78  #
79  @cython.boundscheck(False)
80  @cython.wraparound(False)
```

```
81 @cython.cdivision(True) # to avoid the exception checking
82 @cython.nonecheck(False)
83 cdef void  calcula(long pc, long[:,:] ELEM, double[:,:] NOS,double k,
84                    double[:,:] N1q,double[:,:] N2q,double[:,:] N3q,
85                    double[:,:] N4q,double[:,:] N1,double[:,:] N2,
86                    double[:,:] N3,double[:] qsil,double[:] w1,
87                    double[:] qsi_quad,double[:] w_quad,long npg_s,long
                          npg_r,
88                    double[:,:] H,double[:,:] G) nogil:
89     cdef long nelem,nnos,i,l,m,noglobal,ipf
90     cdef long[3] nos
91     nelem = ELEM.shape[0]             # Numero de elementos
92     nnos = NOS.shape[0]               # Numero de nos
93     cdef double[3] X1,X2,X3,Xd,n,g,h
94     cdef double J
95     nos[0] = ELEM[pc,0]       # Nos que compoem o elemento
96     nos[1] = ELEM[pc,1]       # Nos que compoem o elemento
97     nos[2] = ELEM[pc,2]       # Nos que compoem o elemento
98     X1[0] = NOS[nos[0],0]     # Coordenadas do no 1 do elemento
99     X1[1] = NOS[nos[0],1]     # Coordenadas do no 1 do elemento
100    X1[2] = NOS[nos[0],2]     # Coordenadas do no 1 do elemento
101    X2[0] = NOS[nos[1],0]     # Coordenadas do no 2 do elemento
102    X2[1] = NOS[nos[1],1]     # Coordenadas do no 2 do elemento
103    X2[2] = NOS[nos[1],2]     # Coordenadas do no 2 do elemento
104    X3[0] = NOS[nos[2],0]     # Coordenadas do no 3 do elemento
105    X3[1] = NOS[nos[2],1]     # Coordenadas do no 3 do elemento
106    X3[2] = NOS[nos[2],2]     # Coordenadas do no 3 do elemento
107    n[0]=0
108    n[1]=0
109    n[2]=0
110    J=calc_vetnormal(X1,X2,X3,n)    # Vetor unitario normal ao elemento
111    for pf in range(0,nnos): # Laco sobre os pontos fonte
112        Xd[0] = NOS[pf,0]          # Coordenadas do ponto fonte
113        Xd[1] = NOS[pf,1]          # Coordenadas do ponto fonte
114        Xd[2] = NOS[pf,2]          # Coordenadas do ponto fonte
115
116        if (pf==nos[0] or pf==nos[1] or pf==nos[2]): # Integracao singular
               (o ponto fonte pertence ao elemento)
117            if pf==nos[0]:   # O ponto fonte esta no no 1
118                ipf = 1              # Sinaliza que o ponto fonte esta
                       sobre o primeiro no
119            elif pf==nos[1]: # O ponto fonte esta no no 2
120                ipf = 2              # Sinaliza que o ponto fonte esta
                       sobre o segundo no
121            elif pf==nos[2]: # O ponto fonte esta no no 3
122                ipf = 3              # Sinaliza que o ponto fonte esta
                       sobre o terceiro no
123            g[0]=0
124            g[1]=0
```

```
125             g[2]=0
126             calcula_Gs(X1,X2,X3,Xd,ipf,N1q,N2q,N3q,N4q,qsi_quad,w_quad,k,g)
127             h[0]=0
128             h[1]=0
129             h[2]=0
130         else: # Integracao regular (o ponto fonte nao pertence ao elemento)
131             g[0]=0
132             g[1]=0
133             g[2]=0
134 
135             h[0]=0
136             h[1]=0
137             h[2]=0
138             calcula_HeGns(X1,X2,X3,Xd,qsil,w1,n,k,J,N1,N2,N3,g,h)
139         for nolocal in range(0,3):
140             noglobal = ELEM[pc,nolocal]   # Indice da matriz global H
141             H[pf,noglobal] = H[pf,noglobal] + h[nolocal]
142 
143         G[pf,3*pc] = g[0]
144         G[pf,3*pc+1] = g[1]
145         G[pf,3*pc+2] = g[2]
146 
147 
148 #    %% = = = = = DIAGONAL DA MATRIZ H = = = = =
149 #      Calcula os termos da diagonal da matriz H (consideracao de corpo a
        temperatura constante)
150 @cython.boundscheck(False)
151 @cython.wraparound(False)
152 @cython.cdivision(True) # to avoid the exception checking
153 @cython.nonecheck(False)
154 
155 
156 cdef double calc_vetnormal(double[3] X1,double[3] X2, double[3] X3,double
        [3] n) nogil:
157     # Function que calcula o vetor unitario normal ao elemento
158     cdef double[3] v1,v2
159     v1[0] = X3[0] - X2[0]           # Vetor formado pela aresta 3-2 do
            elemento
160     v1[1] = X3[1] - X2[1]           # Vetor formado pela aresta 3-2 do
            elemento
161     v1[2] = X3[2] - X2[2]           # Vetor formado pela aresta 3-2 do
            elemento
162     v2[0] = X1[0] - X2[0]           # Vetor formado pela aresta 1-2 do
            elemento
163     v2[1] = X1[1] - X2[1]           # Vetor formado pela aresta 1-2 do
            elemento
164     v2[2] = X1[2] - X2[2]           # Vetor formado pela aresta 1-2 do
            elemento
165     n[0]=v1[1]*v2[2]-v1[2]*v2[1]
```

```
166     n[1]=v1[2]*v2[0]-v1[0]*v2[2]
167     n[2]=v1[0]*v2[1]-v1[1]*v2[0]
168     J=sqrt(n[0]**2+n[1]**2+n[2]**2)
169     n[0]=n[0]/J
170     n[1]=n[1]/J
171     n[2]=n[2]/J
172     return J
173
174 @cython.boundscheck(False)
175 @cython.wraparound(False)
176 @cython.cdivision(True) # to avoid the exception checking
177 @cython.nonecheck(False)
178
179
180 cdef void calcula_HeGns(double[3] X1,double[3] X2,double[3] X3,double[3] Xd
        ,double[:] eta,double[:] w2,double[3] n,double k,double J,double[:,:] N1
        ,double[:,:] N2,double[:,:] N3,double[3] g,double[3] h) nogil:
181     """
182     Integracao numerica nao-singular de elementos de contorno triangulares
183     lineares continuos
184     """
185     cdef double[3] R,Xc
186     g[0] = 0;     # Inicializa o somatorio de g
187     g[1] = 0;     # Inicializa o somatorio de g
188     g[2] = 0;     # Inicializa o somatorio de g
189     h[0] = 0;     # Inicializa o somatorio de g
190     h[1] = 0;     # Inicializa o somatorio de g
191     h[2] = 0;     # Inicializa o somatorio de g
192
193 #    n_pint1 = qsil.shape[0]     # Nro de pontos de integracao na direcao qsi
194     cdef long n_pint
195     n_pint = eta.shape[0]      # Nro de pontos de integracao na direcao eta
196
197     cdef long l,m
198     cdef double r,Tast,qast,pi
199     pi = 3.141592654
200
201     for l in range(0,n_pint):      # Laco sobre os pontos de integracao
202        for m in range(0,n_pint):  # Laco sobre os pontos de integracao
203           Xc[0] = N1[l,m]*X1[0] + N2[l,m]*X2[0] + N3[l,m]*X3[0] #
              coordenadas dos pontos de integracao
204           Xc[1] = N1[l,m]*X1[1] + N2[l,m]*X2[1] + N3[l,m]*X3[1] #
              coordenadas dos pontos de integracao
205           Xc[2] = N1[l,m]*X1[2] + N2[l,m]*X2[2] + N3[l,m]*X3[2] #
              coordenadas dos pontos de integracao
206
207           # Solucao fundamental: inicio
208           R[0] = Xc[0] - Xd[0]
209           R[1] = Xc[1] - Xd[1]
```

```
210             R[2] = Xc[2] - Xd[2]
211             r = sqrt(R[0]**2 + R[1]**2 + R[2]**2)
212             Tast = 1.0/(4.0*k*pi*r)
213             qast = (R[0]*n[0] + R[1]*n[1] + R[2]*n[2])/(4.0*pi*r**3.0)
214             # Solucao fundamental: fim
215
216             # Integral da matriz H
217             h[0] += qast*N1[l,m]*(1-eta[m])*w2[l]*w2[m]*J
218             h[1] += qast*N2[l,m]*(1-eta[m])*w2[l]*w2[m]*J
219             h[2] += qast*N3[l,m]*(1-eta[m])*w2[l]*w2[m]*J
220             # Integral da matriz G
221             g[0] += Tast*N1[l,m]*(1-eta[m])*w2[l]*w2[m]*J
222             g[1] += Tast*N2[l,m]*(1-eta[m])*w2[l]*w2[m]*J
223             g[2] += Tast*N3[l,m]*(1-eta[m])*w2[l]*w2[m]*J
224
225 @cython.boundscheck(False)
226 @cython.wraparound(False)
227 @cython.cdivision(True) # to avoid the exception checking
228 @cython.nonecheck(False)
229
230
231 cdef void calcula_Gs(double[3] X1,double[3] X2,double[3] X3,double[3] Xd,
232                      long pf,double[:,:] N1q,double[:,:] N2q,double[:,:]
                             N3q,
233                      double[:,:] N4q,double[:] qsi_quad, double[:] w_quad,
234                      double k,double[3] g) nogil:
235     """
236     Function: calcula_Gs
237     Descricao: Integracao singular da matriz G. O elemento triangular e
238                transformado em um elemento quadrilateral degenerado. Os
                       dois
239                primeiros nos deste quadrilatero sao coincidentes (formam um
240                so vertice do triangulo) e correspondem ao ponto onde existe
                        a
241                singularidade, ou seja, ao ponto fonte. Isto faz com que
                       haja
242                uma concentracao de pontos de integracao junto a
243                singularidade, alem do jacobiano ser igual a zero na
244                singularidade. No caso da matriz H, os elementos da diagonal
245                sao calculados pela consideracao de corpo a temperatura
246                constante.
247     Autor:     Gustavo Gontijo, adaptado de Eder Lima de Albuquerque
248
249     Ultima modificacao: 05/05/2014 - 19h23min
250     """
251     cdef long npg,l,m
252     cdef double[3] R,Xc,X1t,X2t,X3t,X4t
253
254     cdef double J,r,Tast,pi
```

```
255     cdef double[4] dNdqsi,dNdeta
256     cdef double dxdqsi,dydqsi,dzdqsi,dxdeta,dydeta,dzdeta,g1,g2,g3
257     g[0] = 0            # Inicializacao da matriz g
258     g[1] = 0            # Inicializacao da matriz g
259     g[2] = 0            # Inicializacao da matriz g
260     npg = qsi_quad.shape[0]    # Numero de pontos de integracao
261     pi = 3.141592654
262     # Transformacao do triangulo em quadrilatero
263
264     X1t[0] = Xd[0]    # coordenadas do primeiro no do quadrilatero
            degenerado
265     X1t[1] = Xd[1]    # coordenadas do primeiro no do quadrilatero
            degenerado
266     X1t[2] = Xd[2]    # coordenadas do primeiro no do quadrilatero
            degenerado
267
268     X4t[0] = Xd[0]    # coordenadas do quarto no do quadrilatero degenerado
269     X4t[1] = Xd[1]    # coordenadas do quarto no do quadrilatero degenerado
270     X4t[2] = Xd[2]    # coordenadas do quarto no do quadrilatero degenerado
271
272     if (pf == 1):    # O ponto fonte esta no no 1
273         X2t[0] = X2[0]              # coordenadas do segundo no do
                quadrilatero degenerado
274         X2t[1] = X2[1]              # coordenadas do segundo no do
                quadrilatero degenerado
275         X2t[2] = X2[2]              # coordenadas do segundo no do
                quadrilatero degenerado
276
277         X3t[0] = X3[0]              # coordenadas do terceiro no do
                quadrilatero degenerado
278         X3t[1] = X3[1]              # coordenadas do terceiro no do
                quadrilatero degenerado
279         X3t[2] = X3[2]              # coordenadas do terceiro no do
                quadrilatero degenerado
280     elif (pf == 2): # O ponto fonte esta no no 2
281         X2t[0] = X3[0]              # coordenadas do segundo no do
                quadrilatero degenerado
282         X2t[1] = X3[1]              # coordenadas do segundo no do
                quadrilatero degenerado
283         X2t[2] = X3[2]              # coordenadas do segundo no do
                quadrilatero degenerado
284
285         X3t[0] = X1[0]              # coordenadas do terceiro no do
                quadrilatero degenerado
286         X3t[1] = X1[1]              # coordenadas do terceiro no do
                quadrilatero degenerado
287         X3t[2] = X1[2]              # coordenadas do terceiro no do
                quadrilatero degenerado
288     elif (pf == 3): # O ponto fonte esta no no 3
```

```
289         X2t[0] = X1[0]              # coordenada do segundo no do
                quadrilatero degenerado
290         X2t[1] = X1[1]              # coordenada do segundo no do
                quadrilatero degenerado
291         X2t[2] = X1[2]              # coordenada do segundo no do
                quadrilatero degenerado
292
293         X3t[0] = X2[0]              # coordenada do terceiro no do
                quadrilatero degenerado
294         X3t[1] = X2[1]              # coordenada do terceiro no do
                quadrilatero degenerado
295         X3t[2] = X2[2]              # coordenada do terceiro no do
                quadrilatero degenerado
296
297 #     Integracao regular do elemento
298     for l in range(0,npg):          # Laco sobre a primeira variavel de
            integracao
299         for m in range(0,npg):      # Laco sobre a segunda variavel de
                integracao
300             # Calculo das funcoes de forma
301             # Coordenadas do ponto campo
302             Xc[0] = N1q[l,m]*X1t[0] + N2q[l,m]*X2t[0] + N3q[l,m]*X3t[0] +
                    N4q[l,m]*X4t[0] # coordenadas dos pontos de integracao
303             Xc[1] = N1q[l,m]*X1t[1] + N2q[l,m]*X2t[1] + N3q[l,m]*X3t[1] +
                    N4q[l,m]*X4t[1] # coordenadas dos pontos de integracao
304             Xc[2] = N1q[l,m]*X1t[2] + N2q[l,m]*X2t[2] + N3q[l,m]*X3t[2] +
                    N4q[l,m]*X4t[2]# coordenadas dos pontos de integracao
305
306             # Jacobiano (varia ao longo do elemento degenerado)
307             dNdqsi[0] = (1./4.)*(-(1. - qsi_quad[m]))
308             dNdqsi[1] = (1./4.)*(1.-qsi_quad[m])
309             dNdqsi[2] = (1./4.)*(1. + qsi_quad[m])
310             dNdqsi[3] = (1./4.)*(-(1. + qsi_quad[m]))
311
312             dNdeta[0] = (1./4.)*(-(1. - qsi_quad[l]))
313             dNdeta[1] = (1./4.)*(-(1. + qsi_quad[l]))
314             dNdeta[2] = (1./4.)*((1. + qsi_quad[l]))
315             dNdeta[3] = (1./4.)*((1. - qsi_quad[l]))
316
317
318             dxdqsi = X1t[0]*dNdqsi[0] + X2t[0]*dNdqsi[1] + X3t[0]*dNdqsi[2]
                    + X4t[0]*dNdqsi[3]
319             dydqsi = X1t[1]*dNdqsi[0] + X2t[1]*dNdqsi[1] + X3t[1]*dNdqsi[2]
                    + X4t[1]*dNdqsi[3]
320             dzdqsi = X1t[2]*dNdqsi[0] + X2t[2]*dNdqsi[1] + X3t[2]*dNdqsi[2]
                    + X4t[2]*dNdqsi[3]
321
322             dxdeta = X1t[0]*dNdeta[0] + X2t[0]*dNdeta[1] + X3t[0]*dNdeta[2]
                    + X4t[0]*dNdeta[3]
```

```
323             dydeta = X1t[1]*dNdeta[0] + X2t[1]*dNdeta[1] + X3t[1]*dNdeta[2]
                    + X4t[1]*dNdeta[3]
324             dzdeta = X1t[2]*dNdeta[0] + X2t[2]*dNdeta[1] + X3t[2]*dNdeta[2]
                    + X4t[2]*dNdeta[3]
325
326             g1 = dydqsi*dzdeta - dzdqsi*dydeta
327             g2 = dzdqsi*dxdeta - dxdqsi*dzdeta
328             g3 = dxdqsi*dydeta - dydqsi*dxdeta
329             J = sqrt(g1**2.0 + g2**2.0 + g3**2.0)
330
331             # Solucao fundamental: inicio
332             R[0] = Xc[0] - Xd[0]
333             R[1] = Xc[1] - Xd[1]
334             R[2] = Xc[2] - Xd[2]
335
336             r = sqrt(R[0]**2 + R[1]**2 + R[2]**2)
337             Tast = 1.0/(4.0*k*pi*r)
338             # Solucao fundamental: fim
339
340             # Integral da matriz G
341             if pf==1:
342                 g[0] += Tast*(N1q[l,m]+N4q[l,m])*w_quad[l]*w_quad[m]*J
343                 g[1] += Tast*N2q[l,m]*w_quad[l]*w_quad[m]*J
344                 g[2] += Tast*N3q[l,m]*w_quad[l]*w_quad[m]*J
345             elif pf==2:
346                 g[0] += Tast*N3q[l,m]*w_quad[l]*w_quad[m]*J
347                 g[1] += Tast*(N1q[l,m]+N4q[l,m])*w_quad[l]*w_quad[m]*J
348                 g[2] += Tast*N2q[l,m]*w_quad[l]*w_quad[m]*J
349             else:
350                 g[0] += Tast*N2q[l,m]*w_quad[l]*w_quad[m]*J
351                 g[1] += Tast*N3q[l,m]*w_quad[l]*w_quad[m]*J
352                 g[2] += Tast*(N1q[l,m]+N4q[l,m])*w_quad[l]*w_quad[m]*J
```

E Codigo integracao2.pyx

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Wed Oct  2 11:39:46 2019

@author: eder
"""
cimport cython
cimport openmp
from cython.parallel cimport prange
from cython.parallel cimport parallel



import numpy as np
from libc.math cimport log
from libc.math cimport atan2
from libc.math cimport sqrt
from libc.math cimport asinh
from libc.stdio cimport printf

import cython

@cython.boundscheck(False)
@cython.wraparound(False)
@cython.cdivision(True) # to avoid the exception checking
@cython.nonecheck(False)


def cal_HeG(double[:,:] NOS, long[:,:] ELEM, double k, double[:] qsil, double
    [:] w1, double[:] qsi_quad, double[:] w_quad, int ncores):
    cdef long nelem, nnos, npg_s, npg_r, pc
    cdef double[:,:] H,G,N1,N2,N3,N1q,N2q,N3q,N4q
    cdef long i,l,m, noglobal, ipf, pf, nolocal
    cdef long[3] nos
    cdef double[3] X1,X2,X3,Xd,n,g,h
    cdef double J

    nelem = ELEM.shape[0]             # Numero de elementos
```

```
39      nnos = NOS.shape[0]                 # Numero de nos
40
41      H=np.zeros((nnos, nnos))
42      G=np.zeros((nnos,3*nelem))
43      ## = = = = = PONTOS E PESOS DE GAUSS = = = = =
44      npg_s = qsi_quad.shape[0]    # Numero de pontos de Gauss para a
            integracao singular
45      npg_r = qsil.shape[0]    # Numero de pontos de Gauss para a integracao
            regular
46      for i in range(npg_r):
47          qsil[i] = 0.5*(qsil[i] + 1)                                  # Converte
                os pontos de Gauss para o intervalo [0, 1]
48          w1[i] = w1[i]*0.5                                            # Converte
                os pesos de Gauss para o intervalo [0, 1]
49  #   %% = = = = = FUNCOES DE FORMA = = = = =
50      N1q = np.zeros((npg_s,npg_s))
51      N2q = np.zeros((npg_s,npg_s))
52      N3q = np.zeros((npg_s,npg_s))
53      N4q = np.zeros((npg_s,npg_s))
54
55      N1 = np.zeros((npg_r,npg_r))
56      N2 = np.zeros((npg_r,npg_r))
57      N3 = np.zeros((npg_r,npg_r))
58
59      for l in range(0,npg_r):
60          for m in range(0,npg_r):
61              N1[l,m] = (1-qsil[m])*qsil[l] # qsi escrito como qsi_linha e
                    eta
62              N2[l,m] = qsil[m]
63              N3[l,m] = 1-N1[l,m]-N2[l,m]
64
65      for l in range(0,npg_s):
66          for m in range(0,npg_s):
67              N1q[l,m] = (1./4.)*(1. - qsi_quad[l])*(1. - qsi_quad[m])
68              N2q[l,m] = (1./4.)*(1. + qsi_quad[l])*(1. - qsi_quad[m])
69              N3q[l,m] = (1./4.)*(1. + qsi_quad[l])*(1. + qsi_quad[m])
70              N4q[l,m] = (1./4.)*(1. - qsi_quad[l])*(1. + qsi_quad[m])
71
72
73      #%% = = = = = CALCULO DOS ELEMENTOS DAS MATRIZES H e G = = = = =
74
75      with nogil:
76
77          for pc in prange(nelem, schedule='static', num_threads=ncores):
78              #if(openmp.omp_get_thread_num()==0 and pc ==0):
79              #    printf("Numero de threads\n")
80              #    printf("%d\n",openmp.omp_get_num_threads())
81
82          #    print(openmp.omp_get_num_threads())
```

```
83
84              nos[0] = ELEM[pc,0]       # Nos que compoem o elemento
85              nos[1] = ELEM[pc,1]       # Nos que compoem o elemento
86              nos[2] = ELEM[pc,2]       # Nos que compoem o elemento
87              X1[0] = NOS[nos[0],0]     # Coordenadas do no 1 do elemento
88              X1[1] = NOS[nos[0],1]     # Coordenadas do no 1 do elemento
89              X1[2] = NOS[nos[0],2]     # Coordenadas do no 1 do elemento
90              X2[0] = NOS[nos[1],0]     # Coordenadas do no 2 do elemento
91              X2[1] = NOS[nos[1],1]     # Coordenadas do no 2 do elemento
92              X2[2] = NOS[nos[1],2]     # Coordenadas do no 2 do elemento
93              X3[0] = NOS[nos[2],0]     # Coordenadas do no 3 do elemento
94              X3[1] = NOS[nos[2],1]     # Coordenadas do no 3 do elemento
95              X3[2] = NOS[nos[2],2]     # Coordenadas do no 3 do elemento
96              n[0]=0
97              n[1]=0
98              n[2]=0
99              J=calc_vetnormal(X1,X2,X3,n)     # Vetor unitario normal ao
                    elemento
100             for pf in range(0,nnos): # Laco sobre os pontos fonte
101                 Xd[0] = NOS[pf,0]          # Coordenadas do ponto fonte
102                 Xd[1] = NOS[pf,1]          # Coordenadas do ponto fonte
103                 Xd[2] = NOS[pf,2]          # Coordenadas do ponto fonte
104                 if (pf==nos[0] or pf==nos[1] or pf==nos[2]): # Integracao
                        singular (o ponto fonte pertence ao elemento)
105                     if pf==nos[0]:   # O ponto fonte esta no no 1
106                         ipf = 1               # Sinaliza que o ponto fonte
                            esta sobre o primeiro no
107                     elif pf==nos[1]: # O ponto fonte esta no no 2
108                         ipf = 2               # Sinaliza que o ponto fonte
                            esta sobre o segundo no
109                     elif pf==nos[2]: # O ponto fonte esta no no 3
110                         ipf = 3               # Sinaliza que o ponto fonte
                            esta sobre o terceiro no
111                     g[0]=0
112                     g[1]=0
113                     g[2]=0
114                     calcula_Gs(X1,X2,X3,Xd,ipf,N1q,N2q,N3q,N4q,qsi_quad,
                            w_quad,k,g)
115                     h[0]=0
116                     h[1]=0
117                     h[2]=0
118                 else: # Integracao regular (o ponto fonte nao pertence ao
                        elemento)
119                     g[0]=0
120                     g[1]=0
121                     g[2]=0
122                     h[0]=0
123                     h[1]=0
124                     h[2]=0
```

```
125                     calcula_HeGns(X1,X2,X3,Xd,qsil,w1,n,k,J,N1,N2,N3,g,h)
126                 for nolocal in range(0,3):
127                     noglobal = ELEM[pc,nolocal]    # Indice da matriz global
                            H
128                     H[pf,noglobal] += h[nolocal]
129                 G[pf,3*pc] = g[0]
130                 G[pf,3*pc+1] = g[1]
131                 G[pf,3*pc+2] = g[2]
132
133     for m in range(0,nnos):
134         H[m,m] = 0
135         for l in range(0,nnos):
136             if l != m:
137                 H[m,m] += - H[m,l]
138     return H,G
139 #
140 @cython.boundscheck(False)
141 @cython.wraparound(False)
142 @cython.cdivision(True) # to avoid the exception checking
143 @cython.nonecheck(False)
144 cdef void  calcula(long pc, long[:,:] ELEM, double[:,:] NOS,double k,
145                    double[:,:] N1q,double[:,:] N2q,double[:,:] N3q,
146                    double[:,:] N4q,double[:,:] N1,double[:,:] N2,
147                    double[:,:] N3,double[:] qsil,double[:] w1,
148                    double[:] qsi_quad,double[:] w_quad,long npg_s,long
                           npg_r,
149                    double[:,:] H,double[:,:] G) nogil:
150     cdef long nelem,nnos,i,l,m,noglobal,ipf
151     cdef long[3] nos
152     nelem = ELEM.shape[0]              # Numero de elementos
153     nnos = NOS.shape[0]                # Numero de nos
154     cdef double[3] X1,X2,X3,Xd,n,g,h
155     cdef double J
156     nos[0] = ELEM[pc,0]        # Nos que compoem o elemento
157     nos[1] = ELEM[pc,1]        # Nos que compoem o elemento
158     nos[2] = ELEM[pc,2]        # Nos que compoem o elemento
159     X1[0] = NOS[nos[0],0]      # Coordenadas do no 1 do elemento
160     X1[1] = NOS[nos[0],1]      # Coordenadas do no 1 do elemento
161     X1[2] = NOS[nos[0],2]      # Coordenadas do no 1 do elemento
162     X2[0] = NOS[nos[1],0]      # Coordenadas do no 2 do elemento
163     X2[1] = NOS[nos[1],1]      # Coordenadas do no 2 do elemento
164     X2[2] = NOS[nos[1],2]      # Coordenadas do no 2 do elemento
165     X3[0] = NOS[nos[2],0]      # Coordenadas do no 3 do elemento
166     X3[1] = NOS[nos[2],1]      # Coordenadas do no 3 do elemento
167     X3[2] = NOS[nos[2],2]      # Coordenadas do no 3 do elemento
168     n[0]=0
169     n[1]=0
170     n[2]=0
171     J=calc_vetnormal(X1,X2,X3,n)     # Vetor unitario normal ao elemento
```

```
172     for pf in range(0,nnos): # Laco sobre os pontos fonte
173         Xd[0] = NOS[pf,0]          # Coordenadas do ponto fonte
174         Xd[1] = NOS[pf,1]          # Coordenadas do ponto fonte
175         Xd[2] = NOS[pf,2]          # Coordenadas do ponto fonte
176 
177         if (pf==nos[0] or pf==nos[1] or pf==nos[2]): # Integracao singular
                (o ponto fonte pertence ao elemento)
178             if pf==nos[0]:   # O ponto fonte esta no no 1
179                 ipf = 1              # Sinaliza que o ponto fonte esta
                        sobre o primeiro no
180             elif pf==nos[1]: # O ponto fonte esta no no 2
181                 ipf = 2              # Sinaliza que o ponto fonte esta
                        sobre o segundo no
182             elif pf==nos[2]: # O ponto fonte esta no no 3
183                 ipf = 3              # Sinaliza que o ponto fonte esta
                        sobre o terceiro no
184             g[0]=0
185             g[1]=0
186             g[2]=0
187             calcula_Gs(X1,X2,X3,Xd,ipf,N1q,N2q,N3q,N4q,qsi_quad,w_quad,k,g)
188             h[0]=0
189             h[1]=0
190             h[2]=0
191         else: # Integracao regular (o ponto fonte nao pertence ao elemento)
192             g[0]=0
193             g[1]=0
194             g[2]=0
195 
196             h[0]=0
197             h[1]=0
198             h[2]=0
199             calcula_HeGns(X1,X2,X3,Xd,qsil,w1,n,k,J,N1,N2,N3,g,h)
200         for nolocal in range(0,3):
201             noglobal = ELEM[pc,nolocal]   # Indice da matriz global H
202             H[pf,noglobal] = H[pf,noglobal] + h[nolocal]
203 
204         G[pf,3*pc] = g[0]
205         G[pf,3*pc+1] = g[1]
206         G[pf,3*pc+2] = g[2]
207 
208 
209 #    %% = = = = = DIAGONAL DA MATRIZ H = = = = =
210 #      Calcula os termos da diagonal da matriz H (consideracao de corpo a
        temperatura constante)
211 @cython.boundscheck(False)
212 @cython.wraparound(False)
213 @cython.cdivision(True) # to avoid the exception checking
214 @cython.nonecheck(False)
215 
```

```
216
217 cdef double calc_vetnormal(double[3] X1,double[3] X2, double[3] X3,double
        [3] n) nogil:
218     # Function que calcula o vetor unitario normal ao elemento
219     cdef double[3] v1,v2
220     v1[0] = X3[0] - X2[0]           # Vetor formado pela aresta 3-2 do
            elemento
221     v1[1] = X3[1] - X2[1]           # Vetor formado pela aresta 3-2 do
            elemento
222     v1[2] = X3[2] - X2[2]           # Vetor formado pela aresta 3-2 do
            elemento
223     v2[0] = X1[0] - X2[0]           # Vetor formado pela aresta 1-2 do
            elemento
224     v2[1] = X1[1] - X2[1]           # Vetor formado pela aresta 1-2 do
            elemento
225     v2[2] = X1[2] - X2[2]           # Vetor formado pela aresta 1-2 do
            elemento
226     n[0]=v1[1]*v2[2]-v1[2]*v2[1]
227     n[1]=v1[2]*v2[0]-v1[0]*v2[2]
228     n[2]=v1[0]*v2[1]-v1[1]*v2[0]
229     J=sqrt(n[0]**2+n[1]**2+n[2]**2)
230     n[0]=n[0]/J
231     n[1]=n[1]/J
232     n[2]=n[2]/J
233     return J
234
235 @cython.boundscheck(False)
236 @cython.wraparound(False)
237 @cython.cdivision(True) # to avoid the exception checking
238 @cython.nonecheck(False)
239
240
241 cdef void calcula_HeGns(double[3] X1,double[3] X2,double[3] X3,double[3] Xd
        ,double[:] eta,double[:] w2,double[3] n,double k,double J,double[:,:] N1
        ,double[:,:] N2,double[:,:] N3,double[3] g,double[3] h) nogil:
242     """
243     Integracao numerica nao-singular de elementos de contorno triangulares
244     lineares continuos
245     """
246     cdef double[3] R,Xc
247     g[0] = 0;     # Inicializa o somatorio de g
248     g[1] = 0;     # Inicializa o somatorio de g
249     g[2] = 0;     # Inicializa o somatorio de g
250     h[0] = 0;     # Inicializa o somatorio de g
251     h[1] = 0;     # Inicializa o somatorio de g
252     h[2] = 0;     # Inicializa o somatorio de g
253
254 #    n_pint1 = qsil.shape[0]     # Nro de pontos de integracao na direcao qsi
255     cdef long n_pint
```

```
256     n_pint = eta.shape[0]      # Nro de pontos de integracao na direcao eta
257
258     cdef long l,m
259     cdef double r,Tast,qast,pi
260     pi = 3.141592654
261
262     for l in range(0,n_pint):      # Laco sobre os pontos de integracao
263         for m in range(0,n_pint):  # Laco sobre os pontos de integracao
264             Xc[0] = N1[l,m]*X1[0] + N2[l,m]*X2[0] + N3[l,m]*X3[0] #
                    coordenadas dos pontos de integracao
265             Xc[1] = N1[l,m]*X1[1] + N2[l,m]*X2[1] + N3[l,m]*X3[1] #
                    coordenadas dos pontos de integracao
266             Xc[2] = N1[l,m]*X1[2] + N2[l,m]*X2[2] + N3[l,m]*X3[2] #
                    coordenadas dos pontos de integracao
267
268             # Solucao fundamental: inicio
269             R[0] = Xc[0] - Xd[0]
270             R[1] = Xc[1] - Xd[1]
271             R[2] = Xc[2] - Xd[2]
272             r = sqrt(R[0]**2 + R[1]**2 + R[2]**2)
273             Tast = 1.0/(4.0*k*pi*r)
274             qast = (R[0]*n[0] + R[1]*n[1] + R[2]*n[2])/(4.0*pi*r**3.0)
275             # Solucao fundamental: fim
276
277             # Integral da matriz H
278             h[0] += qast*N1[l,m]*(1-eta[m])*w2[l]*w2[m]*J
279             h[1] += qast*N2[l,m]*(1-eta[m])*w2[l]*w2[m]*J
280             h[2] += qast*N3[l,m]*(1-eta[m])*w2[l]*w2[m]*J
281             # Integral da matriz G
282             g[0] += Tast*N1[l,m]*(1-eta[m])*w2[l]*w2[m]*J
283             g[1] += Tast*N2[l,m]*(1-eta[m])*w2[l]*w2[m]*J
284             g[2] += Tast*N3[l,m]*(1-eta[m])*w2[l]*w2[m]*J
285
286 @cython.boundscheck(False)
287 @cython.wraparound(False)
288 @cython.cdivision(True) # to avoid the exception checking
289 @cython.nonecheck(False)
290
291
292 cdef void calcula_Gs(double[3] X1,double[3] X2,double[3] X3,double[3] Xd,
293                      long pf,double[:,:] N1q,double[:,:] N2q,double[:,:]
                             N3q,
294                      double[:,:] N4q,double[:] qsi_quad, double[:] w_quad,
295                      double k,double[3] g) nogil:
296     """
297     Function: calcula_Gs
298     Descricao: Integracao singular da matriz G. O elemento triangular e
299                transformado em um elemento quadrilateral degenerado. Os
                       dois
```

```
300                 primeiros nos deste quadrilatero sao coincidentes (formam um
301                 so vertice do triangulo) e correspondem ao ponto onde existe
                        a
302                 singularidade, ou seja, ao ponto fonte. Isto faz com que
                        haja
303                 uma concentracao de pontos de integracao junto a
304                 singularidade, alem do jacobiano ser igual a zero na
305                 singularidade. No caso da matriz H, os elementos da diagonal
306                 sao calculados pela consideracao de corpo a temperatura
307                 constante.
308     Autor:      Gustavo Gontijo, adaptado de Eder Lima de Albuquerque
309
310     Ultima modificacao: 05/05/2014 - 19h23min
311     """
312     cdef long npg,l,m
313     cdef double[3] R,Xc,X1t,X2t,X3t,X4t
314
315     cdef double J,r,Tast,pi
316     cdef double[4] dNdqsi,dNdeta
317     cdef double dxdqsi,dydqsi,dzdqsi,dxdeta,dydeta,dzdeta,g1,g2,g3
318     g[0] = 0            # Inicializacao da matriz g
319     g[1] = 0            # Inicializacao da matriz g
320     g[2] = 0            # Inicializacao da matriz g
321     npg = qsi_quad.shape[0]    # Numero de pontos de integracao
322     pi = 3.141592654
323     # Transformacao do triangulo em quadrilatero
324
325     X1t[0] = Xd[0]    # coordenadas do primeiro no do quadrilatero
            degenerado
326     X1t[1] = Xd[1]    # coordenadas do primeiro no do quadrilatero
            degenerado
327     X1t[2] = Xd[2]    # coordenadas do primeiro no do quadrilatero
            degenerado
328
329     X4t[0] = Xd[0]    # coordenadas do quarto no do quadrilatero degenerado
330     X4t[1] = Xd[1]    # coordenadas do quarto no do quadrilatero degenerado
331     X4t[2] = Xd[2]    # coordenadas do quarto no do quadrilatero degenerado
332
333     if (pf == 1):    # O ponto fonte esta no no 1
334         X2t[0] = X2[0]              # coordenadas do segundo no do
                quadrilatero degenerado
335         X2t[1] = X2[1]              # coordenadas do segundo no do
                quadrilatero degenerado
336         X2t[2] = X2[2]              # coordenadas do segundo no do
                quadrilatero degenerado
337
338         X3t[0] = X3[0]              # coordenadas do terceiro no do
                quadrilatero degenerado
```

```
339         X3t[1] = X3[1]              # coordenadas do terceiro no do
                quadrilatero degenerado
340         X3t[2] = X3[2]              # coordenadas do terceiro no do
                quadrilatero degenerado
341     elif (pf == 2): # O ponto fonte esta no no 2
342         X2t[0] = X3[0]              # coordenadas do segundo no do
                quadrilatero degenerado
343         X2t[1] = X3[1]              # coordenadas do segundo no do
                quadrilatero degenerado
344         X2t[2] = X3[2]              # coordenadas do segundo no do
                quadrilatero degenerado
345
346         X3t[0] = X1[0]              # coordenadas do terceiro no do
                quadrilatero degenerado
347         X3t[1] = X1[1]              # coordenadas do terceiro no do
                quadrilatero degenerado
348         X3t[2] = X1[2]              # coordenadas do terceiro no do
                quadrilatero degenerado
349     elif (pf == 3): # O ponto fonte esta no no 3
350         X2t[0] = X1[0]              # coordenada do segundo no do
                quadrilatero degenerado
351         X2t[1] = X1[1]              # coordenada do segundo no do
                quadrilatero degenerado
352         X2t[2] = X1[2]              # coordenada do segundo no do
                quadrilatero degenerado
353
354         X3t[0] = X2[0]              # coordenada do terceiro no do
                quadrilatero degenerado
355         X3t[1] = X2[1]              # coordenada do terceiro no do
                quadrilatero degenerado
356         X3t[2] = X2[2]              # coordenada do terceiro no do
                quadrilatero degenerado
357
358 #     Integracao regular do elemento
359     for l in range(0,npg):          # Laco sobre a primeira variavel de
            integracao
360         for m in range(0,npg):      # Laco sobre a segunda variavel de
                integracao
361             # Calculo das funcoes de forma
362             # Coordenadas do ponto campo
363             Xc[0] = N1q[l,m]*X1t[0] + N2q[l,m]*X2t[0] + N3q[l,m]*X3t[0] +
                    N4q[l,m]*X4t[0] # coordenadas dos pontos de integracao
364             Xc[1] = N1q[l,m]*X1t[1] + N2q[l,m]*X2t[1] + N3q[l,m]*X3t[1] +
                    N4q[l,m]*X4t[1] # coordenadas dos pontos de integracao
365             Xc[2] = N1q[l,m]*X1t[2] + N2q[l,m]*X2t[2] + N3q[l,m]*X3t[2] +
                    N4q[l,m]*X4t[2]# coordenadas dos pontos de integracao
366
367             # Jacobiano (varia ao longo do elemento degenerado)
368             dNdqsi[0] = (1./4.)*(-(1. - qsi_quad[m]))
```

```
369             dNdqsi[1] = (1./4.)*(1.-qsi_quad[m])
370             dNdqsi[2] = (1./4.)*(1. + qsi_quad[m])
371             dNdqsi[3] = (1./4.)*(-(1. + qsi_quad[m]))
372
373             dNdeta[0] = (1./4.)*(-(1. - qsi_quad[l]))
374             dNdeta[1] = (1./4.)*(-(1. + qsi_quad[l]))
375             dNdeta[2] = (1./4.)*((1. + qsi_quad[l]))
376             dNdeta[3] = (1./4.)*((1. - qsi_quad[l]))
377
378
379             dxdqsi = X1t[0]*dNdqsi[0] + X2t[0]*dNdqsi[1] + X3t[0]*dNdqsi[2]
                    + X4t[0]*dNdqsi[3]
380             dydqsi = X1t[1]*dNdqsi[0] + X2t[1]*dNdqsi[1] + X3t[1]*dNdqsi[2]
                    + X4t[1]*dNdqsi[3]
381             dzdqsi = X1t[2]*dNdqsi[0] + X2t[2]*dNdqsi[1] + X3t[2]*dNdqsi[2]
                    + X4t[2]*dNdqsi[3]
382
383             dxdeta = X1t[0]*dNdeta[0] + X2t[0]*dNdeta[1] + X3t[0]*dNdeta[2]
                    + X4t[0]*dNdeta[3]
384             dydeta = X1t[1]*dNdeta[0] + X2t[1]*dNdeta[1] + X3t[1]*dNdeta[2]
                    + X4t[1]*dNdeta[3]
385             dzdeta = X1t[2]*dNdeta[0] + X2t[2]*dNdeta[1] + X3t[2]*dNdeta[2]
                    + X4t[2]*dNdeta[3]
386
387             g1 = dydqsi*dzdeta - dzdqsi*dydeta
388             g2 = dzdqsi*dxdeta - dxdqsi*dzdeta
389             g3 = dxdqsi*dydeta - dydqsi*dxdeta
390             J = sqrt(g1**2.0 + g2**2.0 + g3**2.0)
391
392             # Solucao fundamental: inicio
393             R[0] = Xc[0] - Xd[0]
394             R[1] = Xc[1] - Xd[1]
395             R[2] = Xc[2] - Xd[2]
396
397             r = sqrt(R[0]**2 + R[1]**2 + R[2]**2)
398             Tast = 1.0/(4.0*k*pi*r)
399             # Solucao fundamental: fim
400
401             # Integral da matriz G
402             if pf==1:
403                 g[0] += Tast*(N1q[l,m]+N4q[l,m])*w_quad[l]*w_quad[m]*J
404                 g[1] += Tast*N2q[l,m]*w_quad[l]*w_quad[m]*J
405                 g[2] += Tast*N3q[l,m]*w_quad[l]*w_quad[m]*J
406             elif pf==2:
407                 g[0] += Tast*N3q[l,m]*w_quad[l]*w_quad[m]*J
408                 g[1] += Tast*(N1q[l,m]+N4q[l,m])*w_quad[l]*w_quad[m]*J
409                 g[2] += Tast*N2q[l,m]*w_quad[l]*w_quad[m]*J
410             else:
411                 g[0] += Tast*N2q[l,m]*w_quad[l]*w_quad[m]*J
```

```
412                     g[1] += Tast*N3q[l,m]*w_quad[l]*w_quad[m]*J
413                     g[2] += Tast*(N1q[l,m]+N4q[l,m])*w_quad[l]*w_quad[m]*J
```

F Codigo integracao3.pyx

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Wed Oct  2 11:39:46 2019

@author: eder
"""
cimport cython



import numpy as np
from libc.math cimport log
from libc.math cimport atan2
from libc.math cimport sqrt
from libc.math cimport asinh
from libc.stdio cimport printf

import cython

@cython.boundscheck(False)
@cython.wraparound(False)
@cython.cdivision(True) # to avoid the exception checking
@cython.nonecheck(False)


def cal_HeG(double[:,:] NOS, long[:,:] ELEM, double k, double[:] qsil, double
    [:] w1, double[:] qsi_quad, double[:] w_quad):
    cdef long nelem, nnos, npg_s, npg_r, pc
    cdef double[:,:] H,G,N1,N2,N3,N1q,N2q,N3q,N4q
    cdef long i,l,m,noglobal,ipf,pf,nolocal
    cdef long[3] nos
    cdef double[3] X1,X2,X3,Xd,n,g,h
    cdef double J

    nelem = ELEM.shape[0]             # Numero de elementos
    nnos = NOS.shape[0]               # Numero de nos

    H=np.zeros((nnos, nnos))
```

```
39      G=np.zeros((nnos,3*nelem))
40      ## = = = = = PONTOS E PESOS DE GAUSS = = = = =
41      npg_s = qsi_quad.shape[0]    # Numero de pontos de Gauss para a
            integracao singular
42      npg_r = qsil.shape[0]    # Numero de pontos de Gauss para a integracao
            regular
43      for i in range(npg_r):
44          qsil[i] = 0.5*(qsil[i] + 1)                                    # Converte
                os pontos de Gauss para o intervalo [0, 1]
45          w1[i] = w1[i]*0.5                                              # Converte
                os pesos de Gauss para o intervalo [0, 1]
46  #   %% = = = = = FUNCOES DE FORMA = = = = =
47      N1q = np.zeros((npg_s,npg_s))
48      N2q = np.zeros((npg_s,npg_s))
49      N3q = np.zeros((npg_s,npg_s))
50      N4q = np.zeros((npg_s,npg_s))
51
52      N1 = np.zeros((npg_r,npg_r))
53      N2 = np.zeros((npg_r,npg_r))
54      N3 = np.zeros((npg_r,npg_r))
55
56      for l in range(0,npg_r):
57          for m in range(0,npg_r):
58              N1[l,m] = (1-qsil[m])*qsil[l] # qsi escrito como qsi_linha e
                    eta
59              N2[l,m] = qsil[m]
60              N3[l,m] = 1-N1[l,m]-N2[l,m]
61
62      for l in range(0,npg_s):
63          for m in range(0,npg_s):
64              N1q[l,m] = (1./4.)*(1. - qsi_quad[l])*(1. - qsi_quad[m])
65              N2q[l,m] = (1./4.)*(1. + qsi_quad[l])*(1. - qsi_quad[m])
66              N3q[l,m] = (1./4.)*(1. + qsi_quad[l])*(1. + qsi_quad[m])
67              N4q[l,m] = (1./4.)*(1. - qsi_quad[l])*(1. + qsi_quad[m])
68
69
70      #%% = = = = = CALCULO DOS ELEMENTOS DAS MATRIZES H e G = = = = =
71      for pc in range(nelem):
72          nos[0] = ELEM[pc,0]        # Nos que compoem o elemento
73          nos[1] = ELEM[pc,1]        # Nos que compoem o elemento
74          nos[2] = ELEM[pc,2]        # Nos que compoem o elemento
75          X1[0] = NOS[nos[0],0]      # Coordenadas do no 1 do elemento
76          X1[1] = NOS[nos[0],1]      # Coordenadas do no 1 do elemento
77          X1[2] = NOS[nos[0],2]      # Coordenadas do no 1 do elemento
78          X2[0] = NOS[nos[1],0]      # Coordenadas do no 2 do elemento
79          X2[1] = NOS[nos[1],1]      # Coordenadas do no 2 do elemento
80          X2[2] = NOS[nos[1],2]      # Coordenadas do no 2 do elemento
81          X3[0] = NOS[nos[2],0]      # Coordenadas do no 3 do elemento
82          X3[1] = NOS[nos[2],1]      # Coordenadas do no 3 do elemento
```

```
83          X3[2] = NOS[nos[2],2]     # Coordenadas do no 3 do elemento
84          n[0]=0
85          n[1]=0
86          n[2]=0
87          J=calc_vetnormal(X1,X2,X3,n)     # Vetor unitario normal ao elemento
88          for pf in range(0,nnos): # Laco sobre os pontos fonte
89              Xd[0] = NOS[pf,0]          # Coordenadas do ponto fonte
90              Xd[1] = NOS[pf,1]          # Coordenadas do ponto fonte
91              Xd[2] = NOS[pf,2]          # Coordenadas do ponto fonte
92              if (pf==nos[0] or pf==nos[1] or pf==nos[2]): # Integracao
                    singular (o ponto fonte pertence ao elemento)
93                  if pf==nos[0]:   # O ponto fonte esta no no 1
94                      ipf = 1               # Sinaliza que o ponto fonte esta
                            sobre o primeiro no
95                  elif pf==nos[1]: # O ponto fonte esta no no 2
96                      ipf = 2               # Sinaliza que o ponto fonte esta
                            sobre o segundo no
97                  elif pf==nos[2]: # O ponto fonte esta no no 3
98                      ipf = 3               # Sinaliza que o ponto fonte esta
                            sobre o terceiro no
99                  g[0]=0
100                 g[1]=0
101                 g[2]=0
102                 calcula_Gs(X1,X2,X3,Xd,ipf,N1q,N2q,N3q,N4q,qsi_quad,w_quad,
                        k,g)
103                 h[0]=0
104                 h[1]=0
105                 h[2]=0
106             else: # Integracao regular (o ponto fonte nao pertence ao
                    elemento)
107                 g[0]=0
108                 g[1]=0
109                 g[2]=0
110                 h[0]=0
111                 h[1]=0
112                 h[2]=0
113             calcula_HeGns(X1,X2,X3,Xd,qsil,w1,n,k,J,N1,N2,N3,g,h)
114             #printf("g[0] = %f8.5",g[0] )
115             for nolocal in range(0,3):
116                 noglobal = ELEM[pc,nolocal]   # Indice da matriz global H
117                 H[pf,noglobal] += h[nolocal]
118             G[pf,3*pc] = g[0]
119             G[pf,3*pc+1] = g[1]
120             G[pf,3*pc+2] = g[2]
121
122     for m in range(0,nnos):
123         H[m,m] = 0
124         for l in range(0,nnos):
125             if l != m:
```

```
126                     H[m,m] += - H[m,l]
127         return H,G
128 #
129
130
131 #     %% = = = = = DIAGONAL DA MATRIZ H = = = = =
132 #       Calcula os termos da diagonal da matriz H (consideracao de corpo a
        temperatura constante)
133 @cython.boundscheck(False)
134 @cython.wraparound(False)
135 @cython.cdivision(True) # to avoid the exception checking
136 @cython.nonecheck(False)
137
138
139 cdef double calc_vetnormal(double[3] X1,double[3] X2, double[3] X3,double
        [3] n) nogil:
140     # Function que calcula o vetor unitario normal ao elemento
141     cdef double[3] v1,v2
142     v1[0] = X3[0] - X2[0]           # Vetor formado pela aresta 3-2 do
            elemento
143     v1[1] = X3[1] - X2[1]           # Vetor formado pela aresta 3-2 do
            elemento
144     v1[2] = X3[2] - X2[2]           # Vetor formado pela aresta 3-2 do
            elemento
145     v2[0] = X1[0] - X2[0]           # Vetor formado pela aresta 1-2 do
            elemento
146     v2[1] = X1[1] - X2[1]           # Vetor formado pela aresta 1-2 do
            elemento
147     v2[2] = X1[2] - X2[2]           # Vetor formado pela aresta 1-2 do
            elemento
148     n[0]=v1[1]*v2[2]-v1[2]*v2[1]
149     n[1]=v1[2]*v2[0]-v1[0]*v2[2]
150     n[2]=v1[0]*v2[1]-v1[1]*v2[0]
151     J=sqrt(n[0]**2+n[1]**2+n[2]**2)
152     n[0]=n[0]/J
153     n[1]=n[1]/J
154     n[2]=n[2]/J
155     return J
156
157 @cython.boundscheck(False)
158 @cython.wraparound(False)
159 @cython.cdivision(True) # to avoid the exception checking
160 @cython.nonecheck(False)
161
162
163 cdef void calcula_HeGns(double[3] X1,double[3] X2,double[3] X3,double[3] Xd
        ,double[:] eta,double[:] w2,double[3] n,double k,double J,double[:,:] N1
        ,double[:,:] N2,double[:,:] N3,double[3] g,double[3] h) nogil:
164     """
```

```
165     Integracao numerica nao-singular de elementos de contorno triangulares
166     lineares continuos
167     """
168     cdef double[3] R,Xc
169     g[0] = 0;     # Inicializa o somatorio de g
170     g[1] = 0;     # Inicializa o somatorio de g
171     g[2] = 0;     # Inicializa o somatorio de g
172     h[0] = 0;     # Inicializa o somatorio de g
173     h[1] = 0;     # Inicializa o somatorio de g
174     h[2] = 0;     # Inicializa o somatorio de g
175 
176 #   n_pint1 = qsil.shape[0]     # Nro de pontos de integracao na direcao qsi
177     cdef long n_pint
178     n_pint = eta.shape[0]      # Nro de pontos de integracao na direcao eta
179 
180     cdef long l,m
181     cdef double r,Tast,qast,pi
182     pi = 3.141592654
183 
184     for l in range(0,n_pint):      # Laco sobre os pontos de integracao
185         for m in range(0,n_pint):  # Laco sobre os pontos de integracao
186             Xc[0] = N1[l,m]*X1[0] + N2[l,m]*X2[0] + N3[l,m]*X3[0] #
                    coordenadas dos pontos de integracao
187             Xc[1] = N1[l,m]*X1[1] + N2[l,m]*X2[1] + N3[l,m]*X3[1] #
                    coordenadas dos pontos de integracao
188             Xc[2] = N1[l,m]*X1[2] + N2[l,m]*X2[2] + N3[l,m]*X3[2] #
                    coordenadas dos pontos de integracao
189 
190             # Solucao fundamental: inicio
191             R[0] = Xc[0] - Xd[0]
192             R[1] = Xc[1] - Xd[1]
193             R[2] = Xc[2] - Xd[2]
194             r = sqrt(R[0]**2 + R[1]**2 + R[2]**2)
195             Tast = 1.0/(4.0*k*pi*r)
196             qast = (R[0]*n[0] + R[1]*n[1] + R[2]*n[2])/(4.0*pi*r**3.0)
197             # Solucao fundamental: fim
198 
199             # Integral da matriz H
200             h[0] += qast*N1[l,m]*(1-eta[m])*w2[l]*w2[m]*J
201             h[1] += qast*N2[l,m]*(1-eta[m])*w2[l]*w2[m]*J
202             h[2] += qast*N3[l,m]*(1-eta[m])*w2[l]*w2[m]*J
203             # Integral da matriz G
204             g[0] += Tast*N1[l,m]*(1-eta[m])*w2[l]*w2[m]*J
205             g[1] += Tast*N2[l,m]*(1-eta[m])*w2[l]*w2[m]*J
206             g[2] += Tast*N3[l,m]*(1-eta[m])*w2[l]*w2[m]*J
207 
208 @cython.boundscheck(False)
209 @cython.wraparound(False)
210 @cython.cdivision(True) # to avoid the exception checking
```

```
211 @cython.nonecheck(False)
212
213
214 cdef void calcula_Gs(double[3] X1,double[3] X2,double[3] X3,double[3] Xd,
215                      long pf,double[:,:] N1q,double[:,:] N2q,double[:,:]
                             N3q,
216                      double[:,:] N4q,double[:] qsi_quad, double[:] w_quad,
217                      double k,double[3] g) nogil:
218     """
219     Function:  calcula_Gs
220     Descricao: Integracao singular da matriz G. O elemento triangular e
221                transformado em um elemento quadrilateral degenerado. Os
                       dois
222                primeiros nos deste quadrilatero sao coincidentes (formam um
223                so vertice do triangulo) e correspondem ao ponto onde existe
                        a
224                singularidade, ou seja, ao ponto fonte. Isto faz com que
                       haja
225                uma concentracao de pontos de integracao junto a
226                singularidade, alem do jacobiano ser igual a zero na
227                singularidade. No caso da matriz H, os elementos da diagonal
228                sao calculados pela consideracao de corpo a temperatura
229                constante.
230     Autor:     Gustavo Gontijo, adaptado de Eder Lima de Albuquerque
231
232     Ultima modificacao: 05/05/2014 - 19h23min
233     """
234     cdef long npg,l,m
235     cdef double[3] R,Xc,X1t,X2t,X3t,X4t
236
237     cdef double J,r,Tast,pi
238     cdef double[4] dNdqsi,dNdeta
239     cdef double dxdqsi,dydqsi,dzdqsi,dxdeta,dydeta,dzdeta,g1,g2,g3
240     g[0] = 0            # Inicializacao da matriz g
241     g[1] = 0            # Inicializacao da matriz g
242     g[2] = 0            # Inicializacao da matriz g
243     npg = qsi_quad.shape[0]   # Numero de pontos de integracao
244     pi = 3.141592654
245     # Transformacao do triangulo em quadrilatero
246
247     X1t[0] = Xd[0]   # coordenadas do primeiro no do quadrilatero
            degenerado
248     X1t[1] = Xd[1]   # coordenadas do primeiro no do quadrilatero
            degenerado
249     X1t[2] = Xd[2]   # coordenadas do primeiro no do quadrilatero
            degenerado
250
251     X4t[0] = Xd[0]   # coordenadas do quarto no do quadrilatero degenerado
252     X4t[1] = Xd[1]   # coordenadas do quarto no do quadrilatero degenerado
```

```
253     X4t[2] = Xd[2]    # coordenadas do quarto no do quadrilatero degenerado
254
255     if (pf == 1):    # O ponto fonte esta no no 1
256         X2t[0] = X2[0]              # coordenadas do segundo no do
                quadrilatero degenerado
257         X2t[1] = X2[1]              # coordenadas do segundo no do
                quadrilatero degenerado
258         X2t[2] = X2[2]              # coordenadas do segundo no do
                quadrilatero degenerado
259
260         X3t[0] = X3[0]              # coordenadas do terceiro no do
                quadrilatero degenerado
261         X3t[1] = X3[1]              # coordenadas do terceiro no do
                quadrilatero degenerado
262         X3t[2] = X3[2]              # coordenadas do terceiro no do
                quadrilatero degenerado
263     elif (pf == 2): # O ponto fonte esta no no 2
264         X2t[0] = X3[0]              # coordenadas do segundo no do
                quadrilatero degenerado
265         X2t[1] = X3[1]              # coordenadas do segundo no do
                quadrilatero degenerado
266         X2t[2] = X3[2]              # coordenadas do segundo no do
                quadrilatero degenerado
267
268         X3t[0] = X1[0]              # coordenadas do terceiro no do
                quadrilatero degenerado
269         X3t[1] = X1[1]              # coordenadas do terceiro no do
                quadrilatero degenerado
270         X3t[2] = X1[2]              # coordenadas do terceiro no do
                quadrilatero degenerado
271     elif (pf == 3): # O ponto fonte esta no no 3
272         X2t[0] = X1[0]              # coordenada do segundo no do
                quadrilatero degenerado
273         X2t[1] = X1[1]              # coordenada do segundo no do
                quadrilatero degenerado
274         X2t[2] = X1[2]              # coordenada do segundo no do
                quadrilatero degenerado
275
276         X3t[0] = X2[0]              # coordenada do terceiro no do
                quadrilatero degenerado
277         X3t[1] = X2[1]              # coordenada do terceiro no do
                quadrilatero degenerado
278         X3t[2] = X2[2]              # coordenada do terceiro no do
                quadrilatero degenerado
279
280 #     Integracao regular do elemento
281     for l in range(0,npg):        # Laco sobre a primeira variavel de
            integracao
```

```
282         for m in range(0,npg):   # Laco sobre a segunda variavel de
                integracao
283             # Calculo das funcoes de forma
284             # Coordenadas do ponto campo
285             Xc[0] = N1q[l,m]*X1t[0] + N2q[l,m]*X2t[0] + N3q[l,m]*X3t[0] +
                    N4q[l,m]*X4t[0] # coordenadas dos pontos de integracao
286             Xc[1] = N1q[l,m]*X1t[1] + N2q[l,m]*X2t[1] + N3q[l,m]*X3t[1] +
                    N4q[l,m]*X4t[1] # coordenadas dos pontos de integracao
287             Xc[2] = N1q[l,m]*X1t[2] + N2q[l,m]*X2t[2] + N3q[l,m]*X3t[2] +
                    N4q[l,m]*X4t[2]# coordenadas dos pontos de integracao
288
289             # Jacobiano (varia ao longo do elemento degenerado)
290             dNdqsi[0] = (1./4.)*(-(1. - qsi_quad[m]))
291             dNdqsi[1] = (1./4.)*(1.-qsi_quad[m])
292             dNdqsi[2] = (1./4.)*(1. + qsi_quad[m])
293             dNdqsi[3] = (1./4.)*(-(1. + qsi_quad[m]))
294
295             dNdeta[0] = (1./4.)*(-(1. - qsi_quad[l]))
296             dNdeta[1] = (1./4.)*(-(1. + qsi_quad[l]))
297             dNdeta[2] = (1./4.)*((1. + qsi_quad[l]))
298             dNdeta[3] = (1./4.)*((1. - qsi_quad[l]))
299
300
301             dxdqsi = X1t[0]*dNdqsi[0] + X2t[0]*dNdqsi[1] + X3t[0]*dNdqsi[2]
                    + X4t[0]*dNdqsi[3]
302             dydqsi = X1t[1]*dNdqsi[0] + X2t[1]*dNdqsi[1] + X3t[1]*dNdqsi[2]
                    + X4t[1]*dNdqsi[3]
303             dzdqsi = X1t[2]*dNdqsi[0] + X2t[2]*dNdqsi[1] + X3t[2]*dNdqsi[2]
                    + X4t[2]*dNdqsi[3]
304
305             dxdeta = X1t[0]*dNdeta[0] + X2t[0]*dNdeta[1] + X3t[0]*dNdeta[2]
                    + X4t[0]*dNdeta[3]
306             dydeta = X1t[1]*dNdeta[0] + X2t[1]*dNdeta[1] + X3t[1]*dNdeta[2]
                    + X4t[1]*dNdeta[3]
307             dzdeta = X1t[2]*dNdeta[0] + X2t[2]*dNdeta[1] + X3t[2]*dNdeta[2]
                    + X4t[2]*dNdeta[3]
308
309             g1 = dydqsi*dzdeta - dzdqsi*dydeta
310             g2 = dzdqsi*dxdeta - dxdqsi*dzdeta
311             g3 = dxdqsi*dydeta - dydqsi*dxdeta
312             J = sqrt(g1**2.0 + g2**2.0 + g3**2.0)
313
314             # Solucao fundamental: inicio
315             R[0] = Xc[0] - Xd[0]
316             R[1] = Xc[1] - Xd[1]
317             R[2] = Xc[2] - Xd[2]
318
319             r = sqrt(R[0]**2 + R[1]**2 + R[2]**2)
320             Tast = 1.0/(4.0*k*pi*r)
```

```
321             # Solucao fundamental: fim
322
323             # Integral da matriz G
324             if pf==1:
325                 g[0] += Tast*(N1q[l,m]+N4q[l,m])*w_quad[l]*w_quad[m]*J
326                 g[1] += Tast*N2q[l,m]*w_quad[l]*w_quad[m]*J
327                 g[2] += Tast*N3q[l,m]*w_quad[l]*w_quad[m]*J
328             elif pf==2:
329                 g[0] += Tast*N3q[l,m]*w_quad[l]*w_quad[m]*J
330                 g[1] += Tast*(N1q[l,m]+N4q[l,m])*w_quad[l]*w_quad[m]*J
331                 g[2] += Tast*N2q[l,m]*w_quad[l]*w_quad[m]*J
332             else:
333                 g[0] += Tast*N2q[l,m]*w_quad[l]*w_quad[m]*J
334                 g[1] += Tast*N3q[l,m]*w_quad[l]*w_quad[m]*J
335                 g[2] += Tast*(N1q[l,m]+N4q[l,m])*w_quad[l]*w_quad[m]*J
```