UNIVERSIDADE FEDERAL DO RIO DE JANEIRO

Mecânica dos fluidos e transmissão de calor computacional
Prof. Gustavo dos Anjos | 18/07/2020

Aluno: Felipe Rodrigues de Mello Alves

DRE: 113278558

Graduando em Engenharia Mecânica | CT/UFRJ

Eq de calor 2D transiente por Método de elementos finitos

1. **Introdução**

   Este relatório tem por objetivo apresentar a solução do problema térmico 2D transiente.

   Primeiramente é apresentado a motivação do uso do método de elementos finitos e quais são os passos para fazer o estudo através dese método.

2. **Motivação**

   Para transformar um problema real em um modelo compatível com soluções computacionais existem algumas metodologias de idealização do fenômeno físico - onde traduzimos suas características através de modelos matemáticos - a fim de discretizá-lo e resolvê-lo através de métodos matemáticos, como esquematizado na 1.

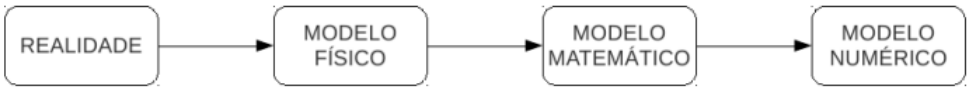

Figura 1: Fluxo de modelagem de problemas

   (a) **Modelo matemático**

   O modelo matemático pode assumir a forma fraca e a forma forte, que tem soluções equivalentes - ou seja, a solução para uma delas representa a solução para a outra. A forma forte, ou forma diferencial, consiste na representação do problema através de um sistema de equações diferenciais, parciais ou ordinárias, no espaço ou no tempo. Problemas simples podem ser resolvidos analiticamente na forma forte através de técnicas de integração, porém problemas mais complexos podem ter soluções muito difíceis de serem obtidas analiticamente, sendo indicado a utilização de métodos precisos para obtenção de soluções aproximadas. A forma fraca consiste na representação do problema através de uma equação integral ponderada, que é definida como uma equação que, a partir do método de solução, distribui diferentes "pesos"a determinados elementos da expressão. O termo forma fraca diz respeito ao fato de a equação ser resolvida adotando uma média em vez de ser solucionada ponto a ponto como na forma forte.

   (b) **Método dos elementos finitos (MEF)**

   O método dos elementos finitos é um procedimento que busca soluções aproximadas para os modelos numéricos e se aplica a uma grande variedade de problemas físicos. Sua acurácia e estabilidade estão largamente estudados, o que confere ao método uma robustez e sólida confiabilidade.

   Para viabilizar a solução, a modelagem divide uma geometria grande e complexa, que é submetida a carregamentos térmicos e/ou mecânicos em pequenas partes – denominadas elementos - os quais passam a representar o domínio contínuo do problema. A filosofia por trás do método é reduzir um problema grande (seu objeto de estudo) em problemas menores (os elementos) e calcular também a relação entre eles. O conjunto dos elementos e de seus pontos de contorno – denominados nós – é conhecido como malha. A malha, portanto, é composta por um número finito de elementos de comportamento bem definido. O MEF é um método poderoso para discretização de geometrias complexas

pois não exige esforço adicional comparado a sua utilização em geometrias regulares.

A precisão do método dos elementos finitos (MEF) depende da quantidade de nós e elementos, do tamanho e dos tipos de elementos da malha. Ou seja, quanto menor for o tamanho e maior for o número dos elementos em uma determinada malha, maior a precisão nos resultados da análise.

(c) **Passos para realizar uma análise por MEF**

Para utilizarmos o método dos elementos finitos, seguimos os seguintes passos:

i. Tranformar a expressão da forma forte para a forma fraca;
ii. Utilizar integração por partes para atingir a redução da ordem da expressão;
iii. Uso do método de aproximação de funções (método de Galerkin);
iv. Montagem das matrizes - conhecido como assembling;
v. Imposição das condições de contorno;
vi. Solução do sistema linear encontrado.

(d) **Funções de forma**

A solução do problema se dá através de funções interpoladoras calculadas em cada nó. Essas funções tem as seguintes características:

i. O valor da função em seu respectivo nó vale 1, e nos outros nós tem valor nulo;
ii. A função entre os nós pode ser linear, quadrática, cúbica ou até de graus maiores;
iii. A função apresenta continuidade no interior de cada elemento, mas não na fronteira dos nós.

Com isso, foram solucionados problemas de transferência de calor 2D permanente e transiente em aula seguindo os passos explicitados acima. A proposta deste relatório é apresentar a solução transiente do problema mantendo a matriz principal do problema simétrica, utilizando matrizes esparsas para melhorar a eficiência do programa e automatizar o problema recebendo qualquer tipo e número de elementos.

3. **Recebendo as matrizes da malha**

Para a geração da malha dos problemas bidimensionais, a construção da malha foi modularizada em um outro código com a classe "$mesh2d$" que define a malha desejada pelo usuário no código do problema, como mostrado na figura 2:

```python
from Mesh_2D import mesh2d
# 2) Definicao da malha pelo usuario
nx = 10
ny = 10
Lx = 1
Ly = 1
lados = 4  # num de lados do elemento da malha

# 2.1) Importacao da malha
malha = mesh2d(Lx, Ly, nx, ny, lados)
npoints = malha.npoints
ne = malha.ne
X = malha.X
Y = malha.Y
IEN = malha.matriz_IEN()
bound = malha.bound
inner = malha.inner
```

Figura 2: Configuração inicial de temperaturas

O codigo da malha 2D é melhor descrito no Relatório escrito para isso.

4. **Recebendo as matrizes do problema**

   Para a geração das matrizes do MEF para realizar o assembling, o código recebe de outra classe denominada "$matriz2D$", como mostrado na figura 3:

```python
# 3) Assembling
# LIL is a convenient format for constructing sparse matrices
from scipy.sparse import lil_matrix
K = lil_matrix((npoints, npoints), dtype='double')
M = lil_matrix((npoints, npoints), dtype='double')
for e in range(0, ne):
    # construir as matrizes do elemento
    v1 = IEN[e, 0]
    v2 = IEN[e, 1]
    v3 = IEN[e, 2]

    from Matrizes2D import matriz2D
    sq = matriz2D(v1=v1, v2=v2, v3=v3, X=malha.X, Y=malha.Y)
    area = sq.areaCalc()
    melem = sq.matrizm()
    kelem = sq.matrizk()

    for ilocal in range(0, 3):
        iglobal = IEN[e, ilocal]
        for jlocal in range(0, 3):
            jglobal = IEN[e, jlocal]

            K[iglobal, jglobal] = K[iglobal, jglobal] + kelem[ilocal, jlocal]
            M[iglobal, jglobal] = M[iglobal, jglobal] + melem[ilocal, jlocal]
```

Figura 3: Recebendo as matrizes da classe matriz2D para realizar o assembling

(a) **Classe** "$matriz2D$"**:**

O código da classe "$matriz2D$" utiliza as formulas sugeridas no material do curso para calcular as matrizes dos elementos do MEF, como mostrado na figura 4:

```python
class matriz2D():
    def __init__(self, X, Y, v1, v2, v3):
        import numpy as np

        self.bi = Y[v2] - Y[v3]
        self.bj = Y[v3] - Y[v1]
        self.bk = Y[v1] - Y[v2]
        self.ci = X[v3] - X[v2]
        self.cj = X[v1] - X[v3]
        self.ck = X[v2] - X[v1]

        self.area = (1.0/2.0)*np.linalg.det(np.array([[1.0, X[v1], Y[v1]],
                                                      [1.0, X[v2], Y[v2]],
                                                      [1.0, X[v3], Y[v3]]]))
    def areaCalc(self):
        return self.area

    def matrizm(self):
        import numpy as np
        melem = (self.area/12.0)*np.array([[2.0, 1.0, 1.0],
                                           [1.0, 2.0, 1.0],
                                           [1.0, 1.0, 2.0]])
        return melem

    def matrizk(self):
        import numpy as np
        B = (1.0/(2.0*self.area))*np.array([[self.bi, self.bj, self.bk],
                                            [self.ci, self.cj, self.ck]])
        BT = np.transpose(B)

        kelem = self.area * np.dot(BT, B)
        return kelem
```

Figura 4: classe "$matriz2D$"

(b) **Matrizes esparsas**

Como mostrado na 3, a construção das matrizes K e M utiliza um módulo $scipy.sparse$ que deixa o problema mais eficiente, ignorando os elementos nulos na memória. Existem diversos tipos de matrizes esparsas nesse modulo, por vezes é aconselhável modificar o tipo (no código utilizo a transformação $lilmatrix$ para $csrmatrix$, por conta dos seguintes motivos:

- $lil\ matrix$ é boa para a construção de matrizes, e é utilizada no assembling.
- $csr\ matrix$ é boa para a operação de matrizes, e é utilizada na equação e no loop transiente.

5. **Solução do problema:**

Este exercício descreve a distribuição de temperatura bidimensional em regime transiente uma placa.

$$\frac{du}{dt} - \alpha \bigtriangledown^2 u = 0$$

Para a derivada temporal foi utilizado o método de diferenças finitas. Após realizar os passos i, ii e iii do MEF, temos:

$$\sum_{i=0}^{N}\sum_{j=0}^{N}\left(\int_0^1 N_i N_j \left(\frac{a_i^{n+1} - a_i^n}{\triangle t}\right) d\Omega\right) - \sum_{i=0}^{N}\sum_{j=0}^{N}\left(\alpha \int_0^1 \frac{dN_i}{dx}\frac{dN_i}{dx} + \frac{dN_j}{dy}\frac{dN_j}{dy}\, d\Omega\right) a_i = 0$$

Após denominar as matrizes e fazer o assembling, temos:

$$\frac{M}{\triangle t}a_i^{n+1} - \frac{M}{\triangle t}a_i^n - \alpha K a_i^{n+1} = 0$$

Temos, portanto:

$$(M - \alpha \triangle tK)a_i^{n+1} = Ma_i^n$$
$$Ha_i^{n+1} = f^n$$

a_i representando a temperatura T no nó i.

Os dados usados para a simulaçao são:

(a) Comprimentos laterais da barra $\Omega = 1x1$;

(b) Temperatura constante na extremidade esquerda $T = y$;

(c) Temperatura constante na extremidade direita $T = y^2 + 1$;

(d) Temperatura constante na extremidade superior $T = x^2 + 1$;

(e) Temperatura constante na extremidade inferior $T = x$;

(f) Condutividade térmica $\alpha = 1.0$;

Resultados:

O problema foi resolvido considerando os dois tipos de elemento bidimensional: triangular e quadrático. Sendo assim os resultados serão plotados para ambas as opções de forma distinta. Para o elemento triangular uso um metodo 3D que plota a temperatura como um eixo vertical. Para o elemento quadrangular, uso o método 2D com a temperatura representada por uma *colorbar*.

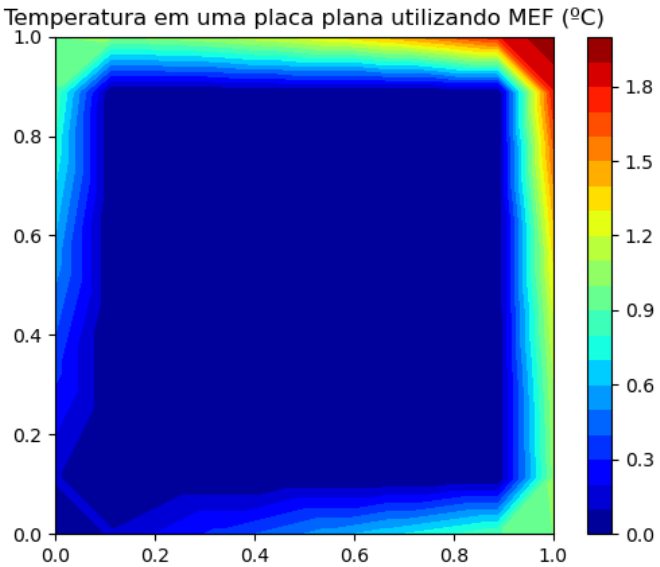

Figura 5: Passo inicial do problema com pontos iniciais zerados

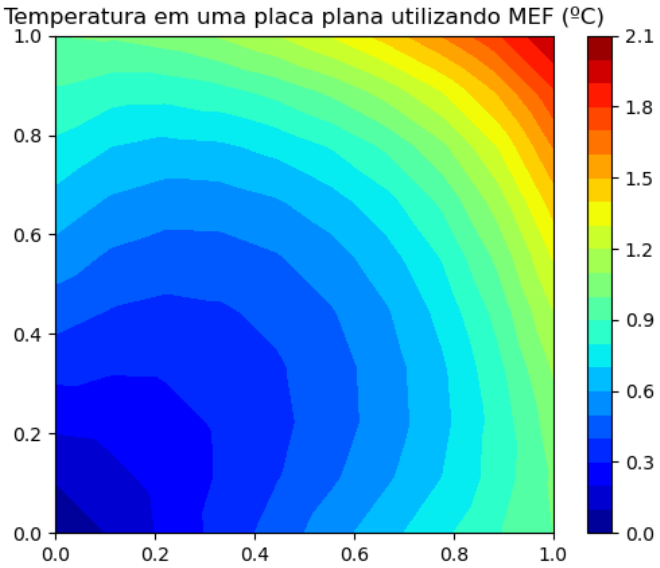

Figura 6: iteração 1

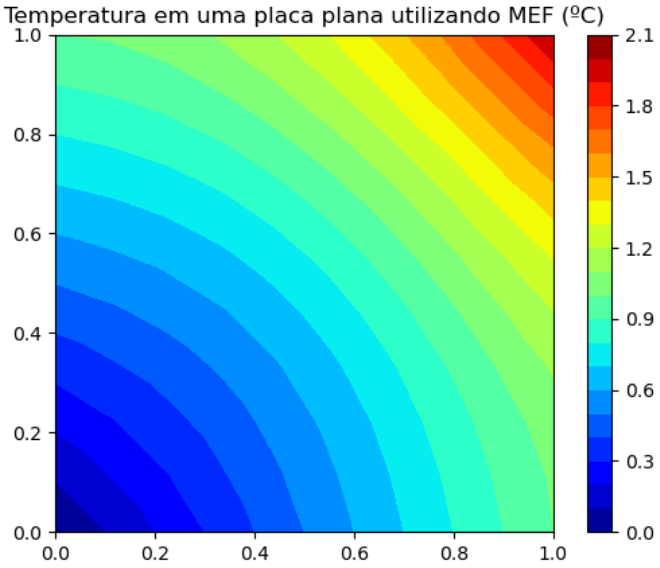

Figura 7: iteração 2

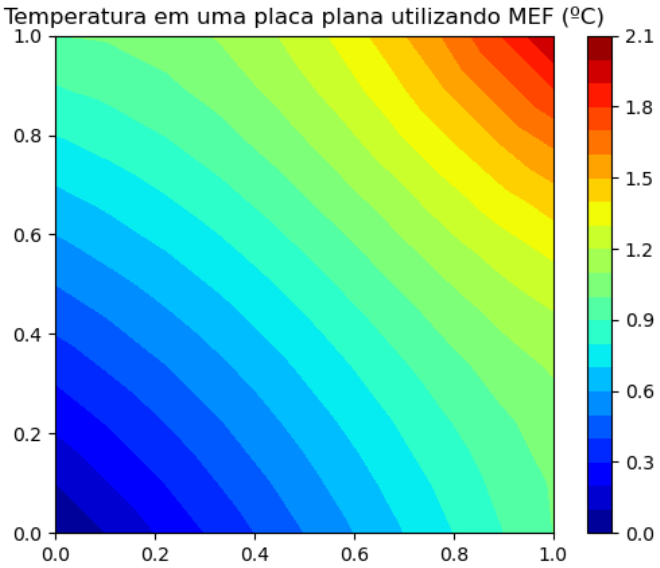

Figura 8: iteração 3

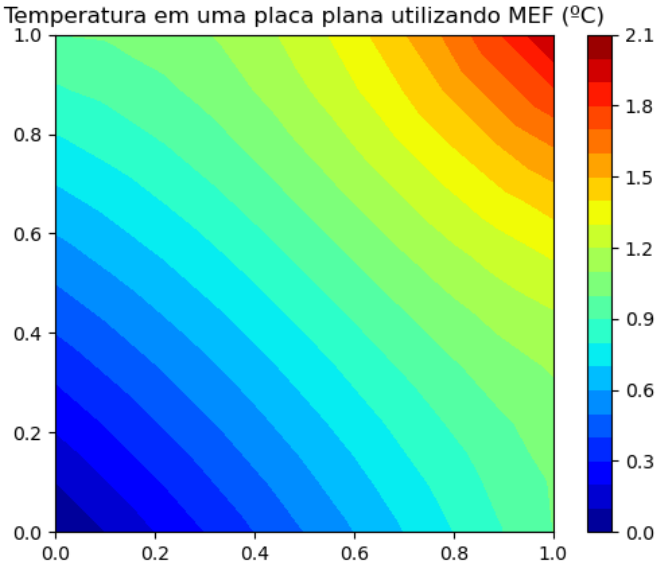

Figura 9: iteração 4

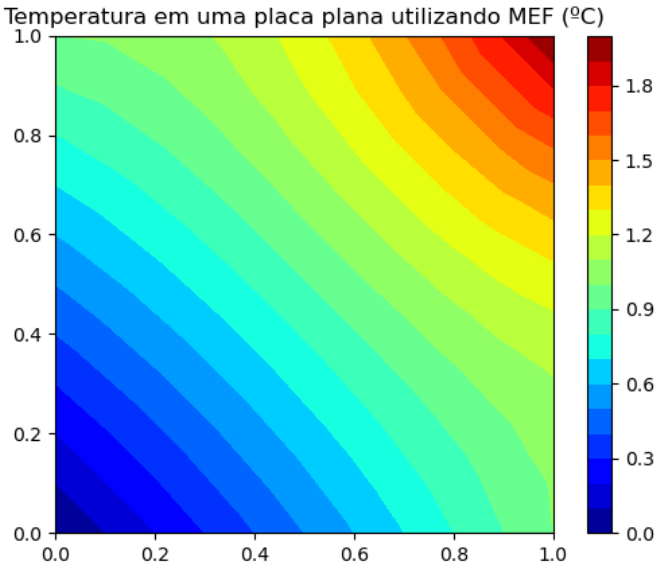

Figura 10: iteração 5

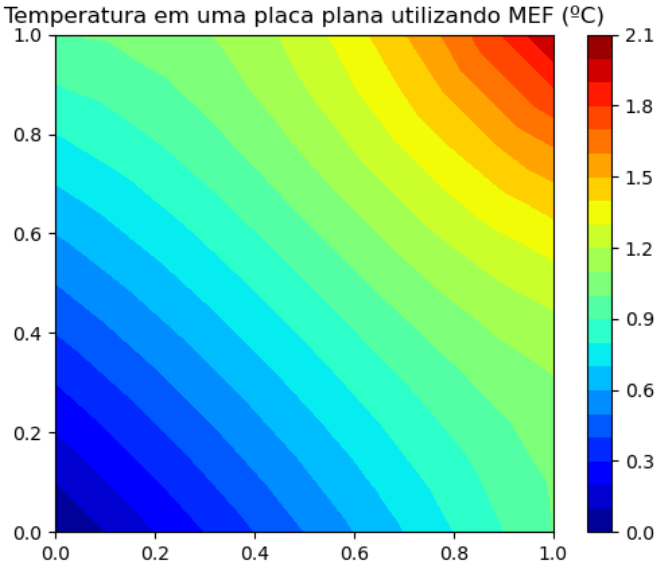

Figura 11: iteração 6

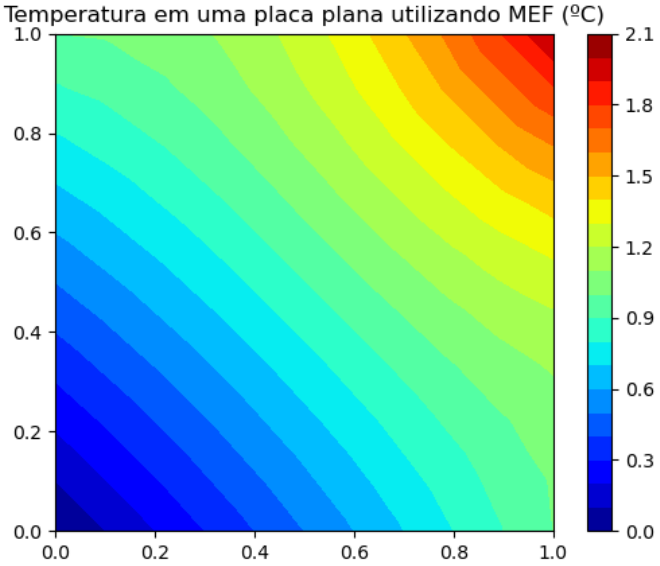

Figura 12: iteração 7

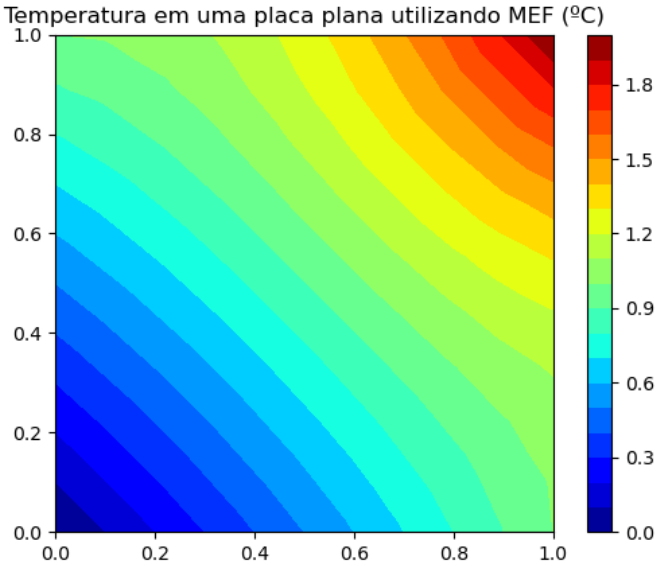

Figura 13: iteração 8

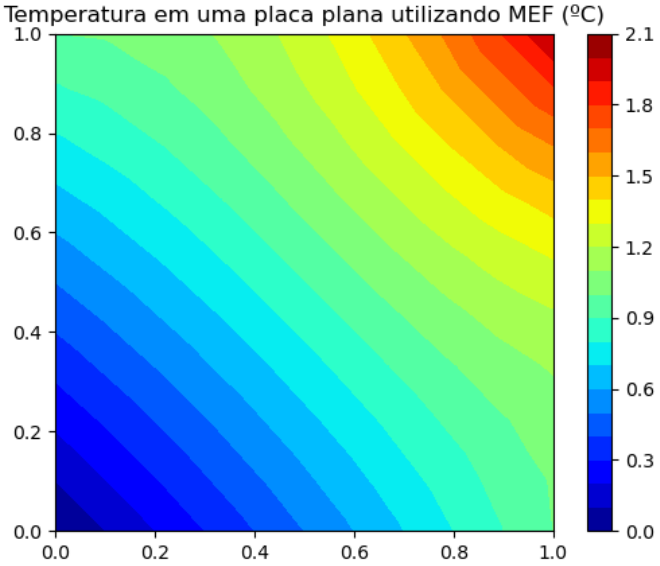

Figura 14: iteração 9

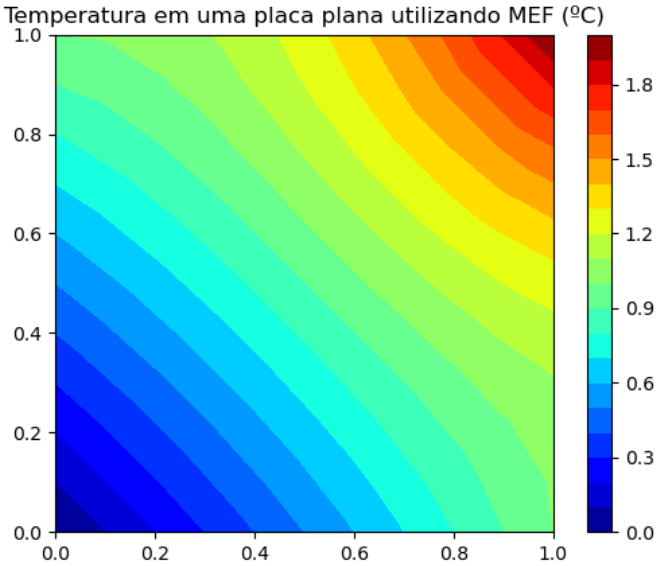

Figura 15: iteração 10

Após algumas iterações, o resultado fica praticamente igual ao resultado do caso permanente, como mostram as figuras 16 e 17

```
T= [0.        0.11111113 0.22222225 0.33333338 0.4444445  0.55555563
 0.66666676 0.77777788 0.88888901 1.00000013 0.11111113 0.2166546
 0.32219943 0.42766321 0.53276773 0.63699349 0.73939819 0.83838852
 0.93120035 1.01234581 0.22222225 0.32219943 0.42226452 0.5223402
 0.62197558 0.72024886 0.81552524 0.9051797  0.98518055 1.04938286
 0.33333338 0.42766321 0.5223402  0.61744617 0.71254519 0.80650852
 0.89727696 0.9816297  1.05496152 1.11111126 0.4444445  0.53276773
 0.62197558 0.71254519 0.80425694 0.89597272 0.98543674 1.06909083
 1.14193421 1.19753102 0.55555563 0.63699349 0.72024886 0.80650852
 0.89597272 0.9876885  1.07940007 1.16736411 1.24615997 1.30864215
 0.66666676 0.73939819 0.81552524 0.89727696 0.98543674 1.07940007
 1.17710775 1.27481474 1.3666965  1.44444464 0.77777788 0.83838852
 0.9051797  0.9816297  1.06909083 1.16736411 1.27481474 1.38809488
 1.50136964 1.60493848 0.88888901 0.93120035 0.98518055 1.05496152
 1.14193421 1.24615997 1.3666965  1.50136964 1.64574611 1.79012369
 1.00000013 1.01234581 1.04938286 1.11111126 1.19753102 1.30864215
 1.44444464 1.60493848 1.79012369 2.00000027]
```

Figura 16: Matriz de temperaturas do caso permanente

```
T= [0.        0.11111078 0.22222157 0.33333235 0.44444313 0.55555391
 0.66666469 0.77777548 0.88888626 0.99999704 0.11111078 0.21674143
 0.3221842  0.4276398  0.53278157 0.63702958 0.73943052 0.83842052
 0.93124968 1.01234268 0.22222157 0.3221842  0.42219853 0.52225889
 0.62196855 0.72028145 0.81557347 0.90520113 0.98520555 1.04937961
 0.33333235 0.4276398  0.52225889 0.61739946 0.71252563 0.80655479
 0.89728785 0.98162401 1.05494658 1.11110782 0.44444313 0.53278157
 0.62196855 0.71252562 0.8042684  0.89597815 0.98543072 1.06905037
 1.14191126 1.19752731 0.55555391 0.63702958 0.72028145 0.80655479
 0.89597814 0.98771353 1.07939048 1.16735135 1.24612152 1.3086381
 0.66666469 0.73943052 0.81557347 0.89728785 0.98543072 1.07939047
 1.17711948 1.27481542 1.36668643 1.44444017 0.77777548 0.83842051
 0.90520112 0.981624   1.06905036 1.16735133 1.2748154  1.38810831
 1.50135969 1.60493351 0.88888626 0.93124968 0.98520555 1.05494657
 1.14191124 1.24612148 1.36668636 1.50135965 1.64571176 1.79011816
 0.99999704 1.01234268 1.04937961 1.11110782 1.19752731 1.3086381
 1.44444264 1.60493195 1.79011577 1.99999408]
```

Figura 17: Matriz de temperaturas da iteração 50 no caso transiente

Plotagem do elemento triangular

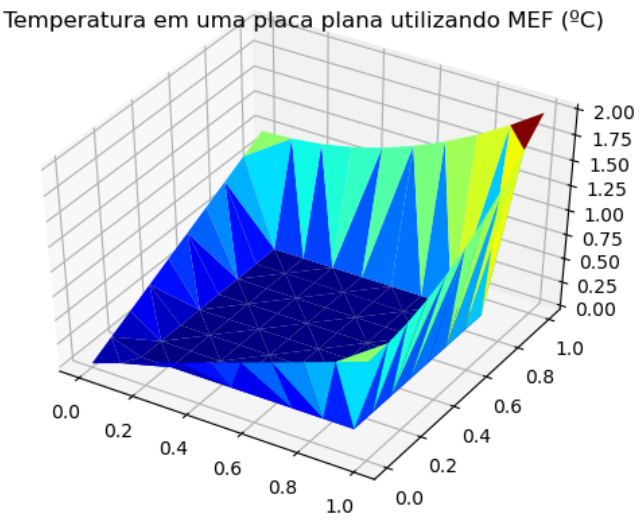

Figura 18: Passo inicial do problema com pontos iniciais zerados

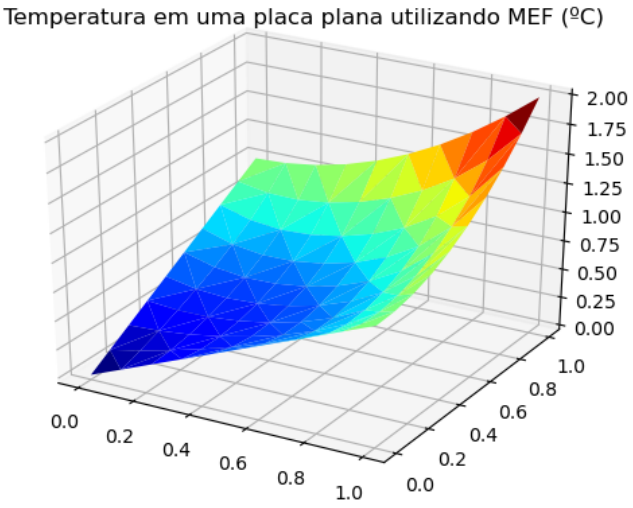

Figura 19: iteração 1

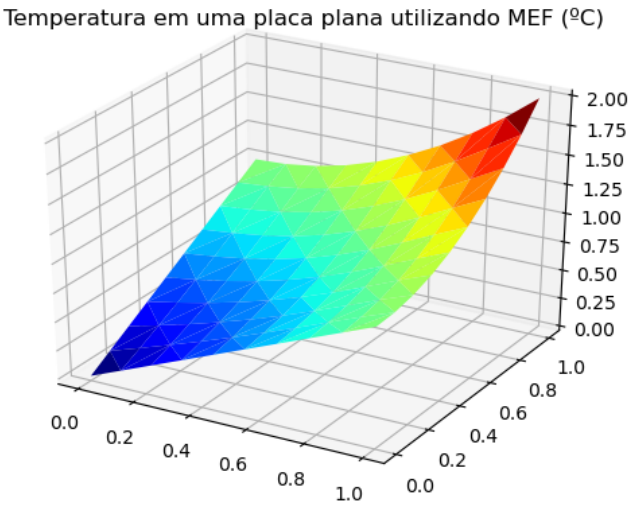

Figura 20: iteração 2

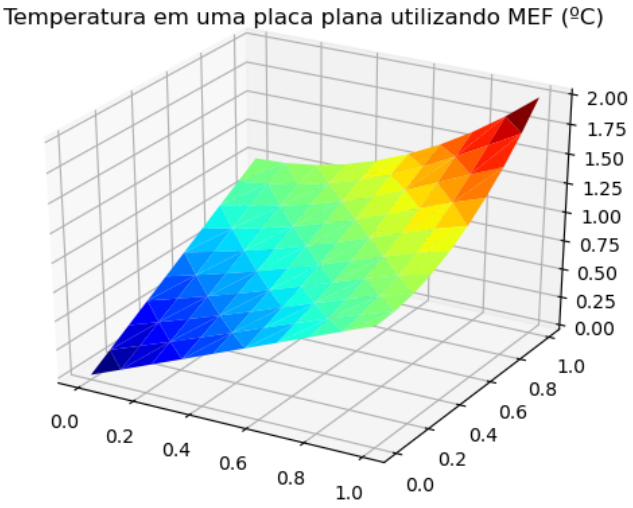

Figura 21: iteração 3

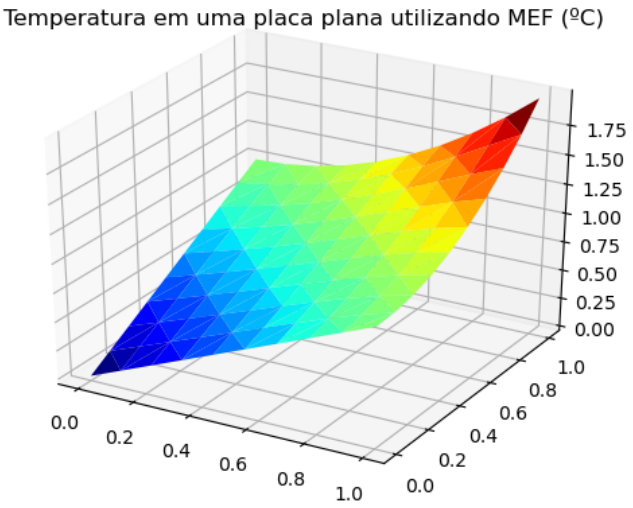

Figura 22: iteração 4

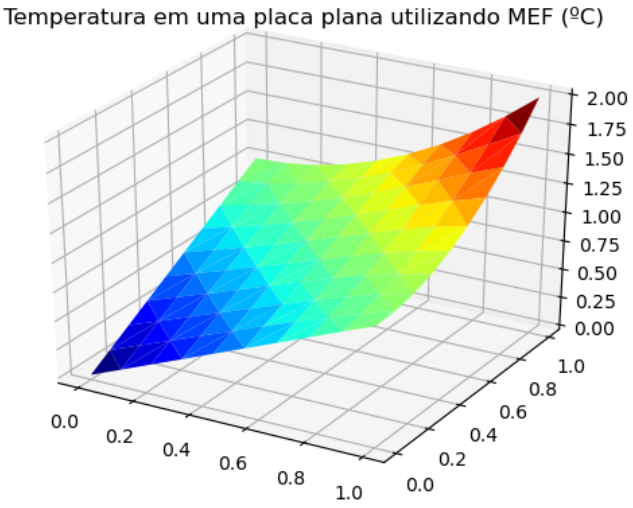

Figura 23: iteração 5

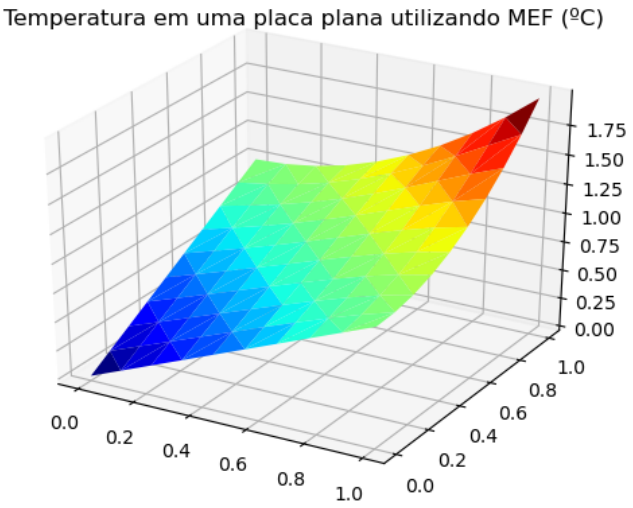

Figura 24: iteração 6

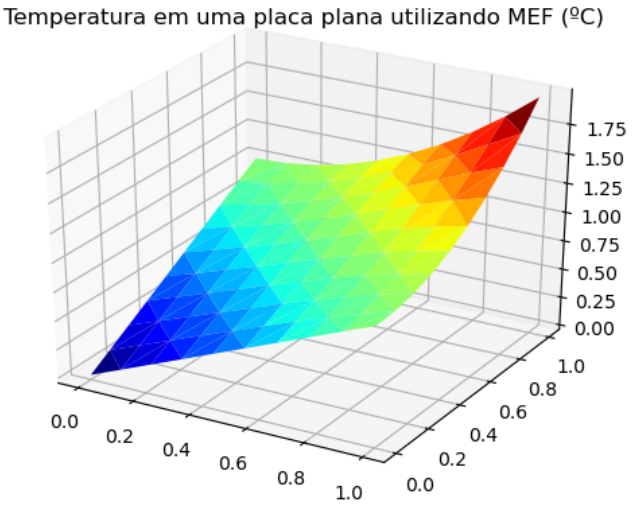

Figura 25: iteração 7

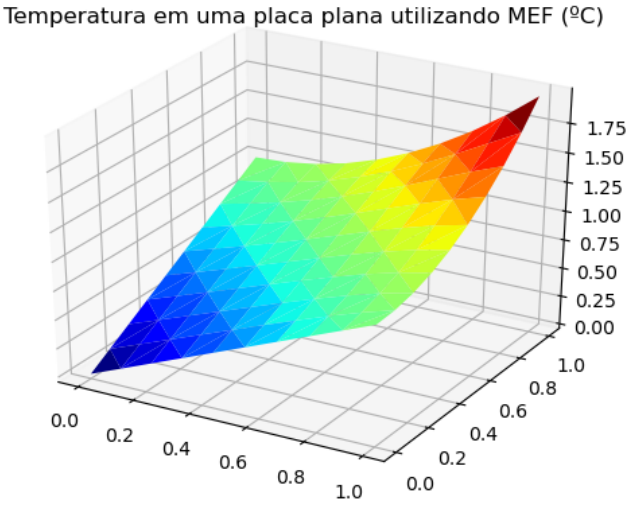

Figura 26: iteração 8

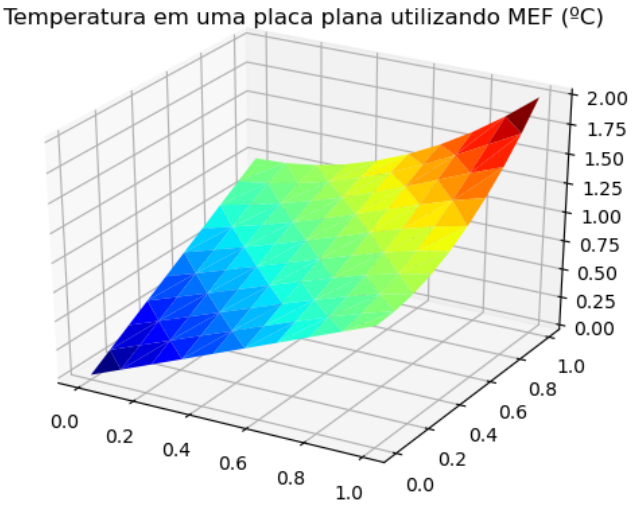

Figura 27: iteração 9

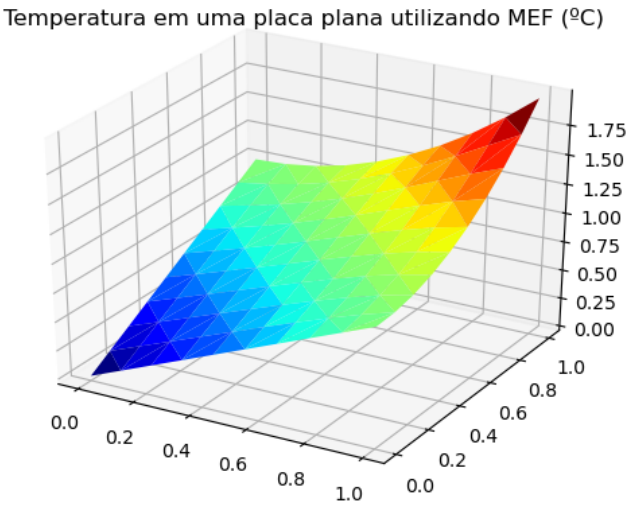

Figura 28: iteração 10

Anexo 1 – Código da malha bidimensional

```
class mesh2d():

    def __init__(self, Lx, Ly, nx, ny, lados):
        import numpy as np

        # Variáveis
        if lados == 4:
            f = 1
        if lados == 3:
            f = 2

        self.ne = f * (nx - 1) * (ny - 1)  # Número de elementos
        self.npoints = nx * ny  # Número de pontos
        self.dx = Lx / (nx - 1)  # Comprimento de cada elemento em x
        self.dy = Ly / (ny - 1)  # Comprimento de cada elemento em y

        # Criação das Listas
        # Matriz de coordenada x de cada ponto da malha
        self.X = np.zeros((self.npoints), dtype='float')
        # Matriz de coordenada y de cada ponto da malha
        self.Y = np.zeros((self.npoints), dtype='float')

        # 4. Preenchimento das listas X e Y
        for i in range(0, self.npoints):
            self.X[i] = Lx * i / (nx - 1)
            if i > (nx - 1):
                self.X[i] = self.X[i - nx]

        for i in range(nx, self.npoints):
            #  Malha com perturbação:
            A = 0.0  # "A = 0.0" é a malha sem perturbacao
            fi = 2 * np.pi / 4.0
            self.Y[i] = (self.Y[i - nx] + self.dy) + A * \
                np.sin((2 * np.pi / 5.0) * self.X[i] - fi)

        # Inner e Bound
        P = np.ones((self.npoints), dtype='int')
        for i in range(self.npoints):
            P[i] = i

        self.bound = list(P)
        inner = np.zeros((nx-2)*(ny-2), dtype='int')
        s = 0
```

```
        for j in range(1, ny-1):
            for i in range(self.npoints):
                if j * nx < i < (((j+1) * nx) - 1):
                    self.bound.remove(i)
                    inner[s] = i
                    s = s + 1

        self.inner = list(inner)
        self.tipo = lados
        self.nx = nx
        self.ny = ny

    def matriz_IEN(self):
        import numpy as np
        self.IEN = np.zeros((self.ne, self.tipo), dtype='int')
        # 5. Matriz IEN
        # 5.1. Malha de quadriláteros
        if self.tipo == 4:
            s = -1
            for e in range(0, self.ne):
                if e % (self.nx - 1) == 0:
                    s = s + 1
                self.IEN[e] = [e + s, e + 1 + s, e +
                               self.nx + 1 + s, e + self.nx + s]

        # 5.2. Malha de triângulos
        if self.tipo == 3:
            i = 1
            for a in range(self.ny - 1):
                for b in range(self.nx - 1):
                    self.IEN[i] = [self.nx * a + b, self.nx +
                                   1 + b + self.nx * a, self.nx * a + b + 1]
                    self.IEN[i - 1] = [self.nx * a + b,
                                       self.nx + self.nx * a + b,
                                       self.nx + self.nx * a + b + 1]
                    i += 2
        return self.IEN

    def plotMalha(self):
        import numpy as np
        import matplotlib.pyplot as plt
        plt.plot(self.X, self.Y, 'ro')
        if self.tipo == 3:
            plt.triplot(self.X, self.Y, self.IEN)
```

```
        if self.tipo == 4:  # OBS: Tambem funciona para tipo == 3
            for i in range(0, len(self.IEN)):
                cx = np.zeros((self.tipo + 1), dtype='float')
                cy = np.zeros((self.tipo + 1), dtype='float')
                for l in range(0, self.tipo):
                    cx[l] = self.X[self.IEN[i][l]]
                    cy[l] = self.Y[self.IEN[i][l]]
                cx[self.tipo] = cx[0]
                cy[self.tipo] = cy[0]
                plt.plot(cx, cy, 'b-')

        plt.gca().set_aspect('equal', adjustable='box')
        # plt.show()
        # plt.savefig("2dmesh")

    def plotSol(self, var):
        import numpy as np
        import matplotlib.pyplot as plt
        from mpl_toolkits.mplot3d import Axes3D
        if self.tipo == 3:
            fig = plt.figure()
            ax = fig.gca(projection='3d')
            plot = ax.plot_trisurf(self.X, self.Y, var, cmap='jet')
            ax.set_title("Temperatura em uma placa plana utilizando MEF (ºC)
")

        elif self.tipo == 4:
            levels = 20
            fig = plt.figure()
            ax = fig.add_subplot(111)
            ax.set_aspect('equal')
            plot = ax.tricontourf(self.X, self.Y, var, levels, cmap='jet')
            fig.colorbar(plot)
            ax.set_title("Temperatura em uma placa plana utilizando MEF (ºC)
")

        return plot
```

Anexo 2 – Código das matrizes MEF 2D

```
class matriz2D():
    def __init__(self, X, Y, v1, v2, v3):
        import numpy as np

        self.bi = Y[v2] - Y[v3]
        self.bj = Y[v3] - Y[v1]
        self.bk = Y[v1] - Y[v2]
        self.ci = X[v3] - X[v2]
        self.cj = X[v1] - X[v3]
        self.ck = X[v2] - X[v1]

        self.area = (1.0/2.0)*np.linalg.det(np.array([[1.0, X[v1], Y[v1]],
                                                      [1.0, X[v2], Y[v2]],
                                                      [1.0, X[v3], Y[v3]]]))
    def areaCalc(self):
        return self.area

    def matrizm(self):
        import numpy as np
        melem = (self.area/12.0)*np.array([[2.0, 1.0, 1.0],
                                           [1.0, 2.0, 1.0],
                                           [1.0, 1.0, 2.0]])
        return melem

    def matrizk(self):
        import numpy as np
        B = (1.0/(2.0*self.area))*np.array([[self.bi, self.bj, self.bk],
                                            [self.ci, self.cj, self.ck]])
        BT = np.transpose(B)

        kelem = self.area * np.dot(BT, B)
        return kelem
```

Anexo 3 – Código do problema térmico 2D transiente

```
from Mesh_2D import mesh2d
import numpy as np
import matplotlib.pyplot as plt
from scipy.sparse.linalg import cg
from scipy.sparse import lil_matrix
from scipy.sparse import csr_matrix
from mpl_toolkits.mplot3d import Axes3D


# 1) Definicoes da simulacao
alpha = 1
time = 0.0
dt = 0.1
nIter = 50

# 2) Definicao da malha pelo usuario
nx = 10
ny = 10
Lx = 1
Ly = 1
lados = 3  # num de lados do elemento da malha

# 2.1) Importacao da malha
malha = mesh2d(Lx, Ly, nx, ny, lados)
npoints = malha.npoints
ne = malha.ne
X = malha.X
Y = malha.Y
IEN = malha.matriz_IEN()
bound = malha.bound
inner = malha.inner

# 2) Condicao de contorno
bval = np.zeros((npoints), dtype='double')
for b in range(nx):
    bval[b] = X[b]
for b in range(nx, (nx*(ny-1)+1), nx):
    bval[b] = Y[b]
```

```
for b in range((2*nx)-1, npoints, nx):
    bval[b] = ((Y[b]) ** 2 + 1)
for b in range(nx * (ny-1), npoints):
    bval[b] = ((X[b]) ** 2 + 1)
# print('bval=',bval)

# 3) Assembling
# LIL is a convenient format for constructing sparse matrices
K = lil_matrix((npoints, npoints), dtype='double')
M = lil_matrix((npoints, npoints), dtype='double')
for e in range(0, ne):
    # construir as matrizes do elemento
    v1 = IEN[e, 0]
    v2 = IEN[e, 1]
    v3 = IEN[e, 2]

    from Matrizes2D import matriz2D
    sq = matriz2D(v1=v1, v2=v2, v3=v3, X=malha.X, Y=malha.Y)
    area = sq.areaCalc()
    melem = sq.matrizm()
    kelem = sq.matrizk()

    for ilocal in range(0, 3):
        iglobal = IEN[e, ilocal]
        for jlocal in range(0, 3):
            jglobal = IEN[e, jlocal]

            K[iglobal, jglobal] = K[iglobal, jglobal] + kelem[ilocal, jlocal
]
            M[iglobal, jglobal] = M[iglobal, jglobal] + melem[ilocal, jlocal
]

    print(f'{round(100*e/ne, 0)} % - calculando as matrizes...')

# change to csr: efficient arithmetic operations CSR + CSR, CSR * CSR, etc.
M = M.tocsr()
K = K.tocsr()

# lado direito do sistema linear eh fixo
H = M + dt*alpha*K

print(type(H))
print(type(M))
print(type(K))
```

```
# imposicao das condicoes de contorno de Dirichlet
f = np.zeros((npoints), dtype='double')
T = np.zeros((npoints), dtype='double')

# save H into H2
H2 = H.copy()
H2 = H2.todense()

# imposicao das condicoes de contorno de Dirichlet
# deixando a matriz k simetrica (passa os valores para o outro lado)
for i in bound:
    H[i, :] = 0.0  # zera a linha toda
    H[:, i] = 0.0
    H[i, i] = 1.0
    T[i] = bval[i]

# check if H is symmetric
# def check_symmetric(a, tol=1e-8):
#     return np.all(np.abs(a-a.T) < tol)
# print(check_symmetric(H, tol=1e-8))

# visualizacao da condicao inicial
plotado = malha.plotSol(var=T)
plt.show()

for n in range(0, nIter):
    f = M * T

    # aplicar c.c. de Dirichlet no vetor f a cada iteração
    for i in bound:
        for j in range(npoints):
            # passa os valores para o outro lado da equacao
            f[j] = f[j] - H2[j, i]*bval[i]

        # f[i] = bval[i]  ## PQ NAO FUNCIONA ???

    for i in bound:
        f[i] = bval[i]

    # solucao do sistema linear
    T = cg(H,f)[0]
    # print('T=', T)

    if lados == 3:
        fig = plt.figure()
```

```
        ax = fig.gca(projection='3d')
        plot = ax.plot_trisurf(X, Y, T,cmap='jet')
        ax.set_title(
            "Temperatura em uma placa plana utilizando MEF (ºC)")
        plt.pause(1)

    elif lados == 4:
        levels = 20
        fig = plt.figure()
        ax = fig.add_subplot(111)
        ax.set_aspect('equal')
        plot = ax.tricontourf(X, Y, T, levels, cmap='jet')
        fig.colorbar(plot)
        ax.set_title(
            "Temperatura em uma placa plana utilizando MEF (ºC)")
        plt.pause(1)

    plt.savefig(f'iteracao {n} triang.png')
```