# UNIVERSIDADE ESTADUAL DE CAMPINAS
# FACULDADE DE ENGENHARIA MECÂNICA


Relatório Final

Trabalho de Conclusão de Curso

# Análise Estrutural e Dimensionamento à Tensão em Treliças Planas e Pórticos em Python


Autora: Gabriela Kishida Koreeda

Orientador: Prof. Dr. Marco Lúcio Bittencourt



Campinas, novembro de 2018

# UNIVERSIDADE ESTADUAL DE CAMPINAS
# FACULDADE DE ENGENHARIA MECÂNICA


Relatório Final

Trabalho de Conclusão de Curso

# Análise Estrutural e Dimensionamento à Tensão em Treliças Planas e Pórticos em Python


Autora: Gabriela Kishida Koreeda

Orientador: Prof. Dr. Marco Lúcio Bittencourt


Curso: Engenharia Mecânica

Trabalho de Conclusão de Curso apresentado à Comissão de Graduação da Faculdade de Engenharia Mecânica, como requisito para a obtenção do título de Engenheira Mecânica.


Campinas, novembro de 2018

S.P. - Brasil

# Agradecimentos

Este trabalho não poderia ter sido realizado sem a ajuda de algumas pessoas às quais presto minha homenagem:

Prof. Dr. Marco Lúcio Bittencourt por sua orientação e paciência na composição deste projeto. Além de ter proposto o tema e fornecido os materiais necessários.

Prof. Dr. Alberto Luiz Serpa e Prof. Dr. Waldyr Luiz Ribeiro Gallo, diretor e diretor associado do curso de Engenharia Mecânica.

Thales Arantes de Castelo Branco e Souza pela sua ajuda na revisão deste trabalho.

# Sumário

# Resumo

KOREEDA, Gabriela Kishida, Análise Estrutural e Dimensionamento à Tensão emTreliças Planas e Pórticos em Python, Faculdade de Engenharia Mecânica, Universidade Estadual de Campinas, Trabalho de Conclusão de Curso, Relatório de Estágio, (2018), 106 pp.

Busca-se apresentar uma introdução à aplicação da linguagem de programação *Python* na área de Engenharia Mecânica. Apresentam-se três programas computacionais: o primeiro compreende a análise estrutural de uma treliça utilizando o método dos elementos finitos. No segundo, realiza-se o redimensionamento da mesma a fim de atender aos limites das propriedades do material sob as condições apresentadas. O terceiro, executa o método em um pórtico, uma estrutura sob efeito de força distribuída e momento. Exemplos são empregados para ilustrar as aplicações. Detalha-se a composição e o funcionamento de cada programa, as especificações dos arquivos de entrada e saída e os gráficos a serem gerados, além de uma breve explicação da teoria utilizada. Objetiva-se fomentar o emprego de *Python* na área de Engenharia Mecânica.

Palavras Chave: Python, Método dos Elementos Finitos, Treliça, Pórtico

# Lista de Figuras



# Lista de Tabelas

# Nomenclatura

## *Abreviações*

**GUI** Graphical User Interface

**MEF** Método dos Elementos Finitos

**MOOC** Massive Open Online Course

**IDE** Integrated Development Environment

**GL** Grau de Liberdade

# 1 Introdução

Python é a quarta linguagem de programação mais utilizada no mundo, de acordo com TIOBE Index, no entanto sua aplicação na área de Engenharia Mecânica ainda é escassa. A maioria dos trabalhos científicos são realizados em Matlab, que é uma alternativa cara, uma vez que se deve comprar uma licença de uso, e menos versátil, dado que Python é *open source* os usuários podem adicionar facilmente novas funcionalidades ao programa, por meio de bibliotecas.

| Apr 2018 | Apr 2017 | Change | Programming Language | Ratings | Change |
|---|---|---|---|---|---|
| 1 | 1 | | Java | 15.777% | +0.21% |
| 2 | 2 | | C | 13.589% | +6.62% |
| 3 | 3 | | C++ | 7.218% | +2.66% |
| 4 | 5 | ↑ | Python | 5.803% | +2.35% |
| 5 | 4 | ↓ | C# | 5.265% | +1.69% |
| 6 | 7 | ↑ | Visual Basic .NET | 4.947% | +1.70% |
| 7 | 6 | ↓ | PHP | 4.218% | +0.84% |
| 8 | 8 | | JavaScript | 3.492% | +0.64% |
| 9 | - | ⇈ | SQL | 2.650% | +2.65% |
| 10 | 11 | ↑ | Ruby | 2.018% | -0.29% |
| 11 | 9 | ↓ | Delphi/Object Pascal | 1.961% | -0.86% |
| 12 | 15 | ↑ | R | 1.806% | -0.33% |
| 13 | 16 | ↑ | Visual Basic | 1.798% | -0.26% |
| 14 | 13 | ↓ | Assembly language | 1.655% | -0.51% |
| 15 | 12 | ↓ | Swift | 1.534% | -0.75% |
| 16 | 10 | ⇊ | Perl | 1.527% | -0.89% |
| 17 | 17 | | MATLAB | 1.457% | -0.59% |

Figura 1: TIOBE index

Este trabalho de conclusão de curso tem como objetivo estudar a utilização da linguagem Python no campo da Engenharia Mecânica, assim como a sua aplicação na análise estrutural de uma treliça 2D e de uma estrutura de pórtico, por meio do método dos elementos finitos. Primeiramente, será apresentada uma revisão bibliográfica, inserindo a importância do problema estudado no âmbito acadêmico. Posteriormente três programas computacionais serão explicados em detalhes, abordando brevemente a teoria por trás do seu funcionamento.

O primeiro programa trata-se de uma análise estrutural de treliças planas, o segundo o dimensi-

onamento da mesma e o terceiro, o cálculo estrutural de um pórtico. Para a demonstração dos códigos desenvolvidos, foram utilizados alguns exemplos, porém qualquer estrutura pode ser analisada, contanto que atenda corretamente às especificações dos arquivos de entrada correspondentes.

Por meio de um arquivo de entrada, o programa saberá as especificações da estrutura, como o número de nós, coordenadas dos nós, material, carregamento, etc... A partir desses dados, serão calculados os deslocamentos, deformações, tensões e forças de reação da treliça que serão armazenados em um arquivo de saída. Os resultados poderão visualizados por gráficos gerados pela a biblioteca matplotlib.

# 2 Revisão Bibliográfica

## 2.1 Python

Python é uma linguagem de programação criada em 1989 por Guido Van Rossum. Ela surgiu pela necessidade de uma linguagem parecida com *ABC* mas que tivesse acesso ao programa *Amoeba* com o qual ele trabalhava. Primeiramente ela foi chamada de *Modula-3* e foi posteriormente renomeada para *Python*, em homenagem ao programa de TV dos anos 70 *Monty Python*. Sua primeira versão, *Python 1.0*, foi lançada em 1991 e atualmente encontra-se na versão 3.7.1.

É uma linguagem de alto nível, ou seja, é simples o suficiente para humanos lerem e escreverem, e também é orientada a objetos, baseada em objetos ao invés de ações, ou dados ao invés de lógica, como definido por THOMPSON (2018).

Python tem como outras principais características:

- Alta legibilidade: devido a simples sintaxe empregada, o que reduz o custo de manutenção e facilita o desenvolvimento colaborativo de programas.
- Gratuita: disponível para download no seu site.
- Extensível: existe uma enorme variedade de bibliotecas disponíveis para complementar as funções básicas da linguagem para qualquer tipo de finalidade.
- Multiplataforma: pode ser utilizada nos sistemas operacionais Microsoft Windows, Linux e Mac OS X.

Algumas aplicações da linguagem incluem: desenvolvimento web, computação numérica e científica, educação (no ensino da lógica de programação), interface gráfica do utilizador, desenvolvimento de softwares e sistemas de e-commerce.

## 2.2 Python em Engenharia Mecânica

Alguns estudos a respeito da utilização da linguagem *Python* na área de engenharia mecânica devem ser mencionados, como o trabalho de JACOBS (2015) que, assim como este trabalho de conclusão, tem como objetivo incentivar o uso da linguagem na implementação de análise de engenharia. Jacobs introduz conceitos básicos de *Python3* como os tipos de dados, estruturas de dados e funções além de abordar as condições necessárias para a construção de um programa bem escrito e eficiente. O conteúdo é

voltado especificamente para estudantes de Engenharia Mecânica e pode ser visto como uma diretriz para o presente estudo, já que este representa uma aplicação específica na área de Mecânica Computacional.

O trabalho de ALKMIN (2016) também pode ser citado, sendo um estudo abrangente de aplicações da linguagem na resolução de problemas térmicos e estruturais por meio do Método dos Elementos Finitos. Nele, Alkmin detalha extensivamente a teoria por trás dos cálculos realizados com alguns trechos de códigos em *Python*. Na realidade, o estudo foi feito no contexto da Engenharia Civil mas pode-se fazer várias intersecções com os conteúdos estudados em Engenharia Mecância. Diferentemente da monografia de Nasser, este trabalho irá focar principalmente no programa desenvolvido em *Python* e aplicado apenas à análise estrutural de treliças e pórticos.

## 2.3 MATLAB

*MATLAB* é uma plataforma de programação desenvolvida especialmente para cientistas e engenheiros. A parte central desse software é a linguagem *MATLAB* que é baseada em matrizes e assim permite uma forma mais natural de escrever expressões matemáticas computacionais.

Esta linguagem é usada tanto na indústria como no meio acadêmico, em aplicações como *deep learning* e *machine learning*, processamento de sinais, imagens e vídeos, controle de sistemas, finança computacional e biologia computacional.

Como já foi ressaltado, o uso por engenheiros é muito comum de forma que se pode destacar alguns recursos na área de engenharia mecânica como: *MATLAB for Mechanical Engineers* (DUKKIPATI, 2009) que é especialmente voltado para estudantes e apresenta uma introdução à linguagem, assim como a aplicação na análise de problemas em controle de sistemas, estática, dinâmica, vibração mecânica, circuitos elétricos e métodos numéricos; *Solving Mechanical Engineering Problems with MATLAB* (NASSERI, 2015), aborda problemas em mecânica dos sólidos e termodinâmicos; *Finite Element Method Using MATLAB* (Kwon e Bang, 1996) apresenta o Método dos Elementos Finitos utilizando a linguagem *MATLAB*.

Assim, ao contrário de *Python*, são muitos os recursos disponíveis sobre o estudo de aplicações em Engenharia Mecânica em *MATLAB*. Porém esta linguagem apresenta algumas desvantagens como: o pagamento de uma licença anual; ser *closed source* e não ser tão usado fora do meio acadêmico ou industrial.

Os programas em *Python* apresentados neste trabalho, são baseados nos programas em *MATLAB* utilizados na matéria EM503 - Introdução aos Métodos Numéricos Aplicados à Engenharia lecionada pelo Prof. Dr. Marco Lúcio Bittencourt.

## 2.4 Método dos Elementos Finitos

Em 1909, Walter Ritz (1878-1909) desenvolveu o Método Ritz, que determinava uma solução aproximada para problemas de valores de contorno na mecânica dos sólidos deformáveis. Essa foi a base para a criação do Método dos Elementos Finitos (MEF) que, posteriormente, recebeu contribuições de Richard Courant (1888-1972). Este introduziu funções lineares especiais definidas sobre regiões triângulares e aplicou o método na solução de problemas de torção. O termo Método dos Elementos Finitos foi apenas futuramente consolidado , em 1960, por mérito de Ray William Clough Jr (1920-2016), em seu artigo *The finite element method in plane stress analysis*, onde propôs uma abordagem semelhante ao Método de Ritz, porém com as modificações de Courant (CAMPOS, 2006).

O primeiro livro à respeito do Método dos Elementos Finitos foi publicado em 1967 por Zienkiewicz e Chung (1967). Nessa época, MEF já era aplicado em uma grande gama de problemas de engenharia, de forma que ao final da década de 60, vários softwares comerciais para realizar este tipo de análise foram criados, como Abaqus, Adina, Ansys, Nastran, etc. Este último foi desenvolvido para a NASA com recursos do governo Norte Americano.

De acordo com BRENNER (2008), o método dos elementos finitos fornece um formalismo para a geração discreta de algoritmos para a aproximação de soluções de equações diferenciais e deve ser vista como uma caixa preta, em que a entrada seria uma equação diferencial e a saída, um algoritmo para a solução correspondente. Uma definição mais genérica é fornecida por PAVANELLO (1997), como sendo um procedimento numérico para análise de estruturas e meios contínuos.

O MEF separa a estrutura em vários elementos e depois os reconecta em nós, garantindo os graus de liberdade apropriados para cada um. A partir do problema analisado, sabe-se as condições de contorno e as equações governantes de forma que o método cria um conjunto de equações algébricas simultâneas a serem aproximadas. Assim, resolvem-se as equações obtendo as variáveis desconhecidas nos nós.

Algumas vantagens na utilização do método são: capacidade de tratar geometrias, restrições e carregamentos complexos e aplicação em uma grande variedade de problemas de engenharia. Desvantagens que podem ser citadas são que obtêm apenas soluções aproximadas e possui erros inerentes, uma vez que depende das informações iniciais no modelamento do problema.

Para a realização deste trabalho de graduação é importante ressaltar o livro *Análise Computacional de Estruturas* (BITTENCOURT, 2014) o qual foi um guia para a realização dos programas computacionais apresentados, uma vez que aborda especificamente o Método dos Elementos Finitos para treliças e pórticos.

# 3 Programa truss2d.py

O primeiro programa a ser apresentado tem como objetivo analisar uma treliça plana. Uma estrutura composta por barras retas, solicitadas por tração ou compressão, interligadas entre si através dos nós (pinos, soldas, rebites ou parafusos) formando uma geometria triangular, capaz de suportar apenas forças normais. É definida como plana pois todos seus elementos encontram-se em apenas um plano espacial. Este tipo de estrutura é comumente utilizado no projeto de pontes, viadutos, telhados, etc... Dessa forma, observa-se a importância da análise estrutural deste tipo de construção, uma vez que deve suportar grandes quantidades de cargas ao longo do dia e garantir a segurança daqueles que a utilizam.

Baseado em HIBELLER (2011), deve-se fazer duas hipóteses na elaboração do projeto de uma treliça, a fim de garantir que cada membro da treliça estará sob ação de duas forças, uma cada extremidade, direcionadas ao longo do eixo do componente. Caso essa força tenda a alongar o elemento, ela é considerada de tração, se ela tender a encurtar, compressão. As hipóteses são apresentadas a seguir:

- Todas as cargas são aplicadas nos nós.
- Os membros são conectados entre si por pinos lisos.

Esta seção está organizada de forma que, primeiramente, será apresentado o arquivo de entrada do exemplo a ser demonstrado. Ele caracteriza a treliça e suas propriedades como coordenadas dos nós, nós de apoio, área e módulo de elasticidade do material de cada elemento e forças externas atuando sobre a estrutura. Posteriormente, uma breve explicação do procedimento seguido na realização dos códigos, baseado no Método dos Elementos Finitos. Após isso, o programa será explicado. Finalmente, o arquivo de saída e os gráficos esperados serão apresentados.

Os códigos apresentados nesta seção e nas seções 4 (*truss2d_design.py*) e 5 (*portico.py*) serão estruturados de acordo com o procedimento comum realizado pelos softwares de Análise de Elementos Finitos. A primeira parte, o Pré-Processamento, consiste no esforço do usuário de inserir as informações necessárias para a construção do modelo (arquivo de entrada), tais como:

1. Tipo de análise: neste caso análise estrutural estática;
2. Tipo de elemento: treliça 2-D linear;
3. Propriedade dos materiais;
4. Identificação das coordenadas dos nós;
5. Definição dos elementos de acordo com as conexões dos nós;

6. Condições de contorno e carregamentos.

A segunda parte, o Processamento, compreende os cálculos realizados pelo computador para a resolução das equações. A última parte, o Pós-Processamento, abrange a análise dos resultados e resume-se ao arquivo de saída e os gráficos gerados.

## 3.1 Arquivo de Entrada

Para ilustrar a utilização e resultados do programa, um exemplo de arquivo de entrada, apresentado no apêndice A, será empregado. O mesmo, deve ter extensão *.fem* e deve ser composto por sete seções, definidas a seguir:

1. Coordenadas de cada nó da treliça no formato:
   ***COORDINATES**
   <número de nós>
   <índice do nó> <coordenadas X e Y>

2. Grupos de elementos que possuem mesmo material e propriedades geométricas:
   ***ELEMENT _GROUPS**
   <número de grupos>
   <número do grupo> <número de elementos no grupo>

3. Incidências que definem os nós de cada elemento finito:
   ***INCIDENCES**
   <índice do elemento> <nós do elemento>

4. Propriedades dos materiais como o módulo de Young e tensão de tração e compressão admissíveis.
   ***MATERIALS**
   <número de materiais>
   Para cada material:
   <Módulo de Young> <Tensão de tração admissível> <Tensão de compressão admissível>

5. Propriedades geométricas como a área longitudinal de cada elemento. O número de propriedades é de acordo com o número de grupos de elementos.
   ***GEOMETRIC _PROPERTIES**
   <número de grupos>
   <Área transversal>

6. Nós que possuem graus de liberdade com valor zero.

   ***BCNODES**

   <número de nós>

   <número do nó> <grau de liberdade eliminado, 1 = UX e 2 = UY>

7. Carregamentos presentes em cada nó.

   ***LOADS**

   <número de nós carregados>

   <número do nó> <grau de liberdade, 1 = UX e 2 = UY> <carregamento>

## 3.2 Desenvolvimento Procedural

A estrutura treliçada apresentada na subseção anterior é representada pela figura 2, abaixo:

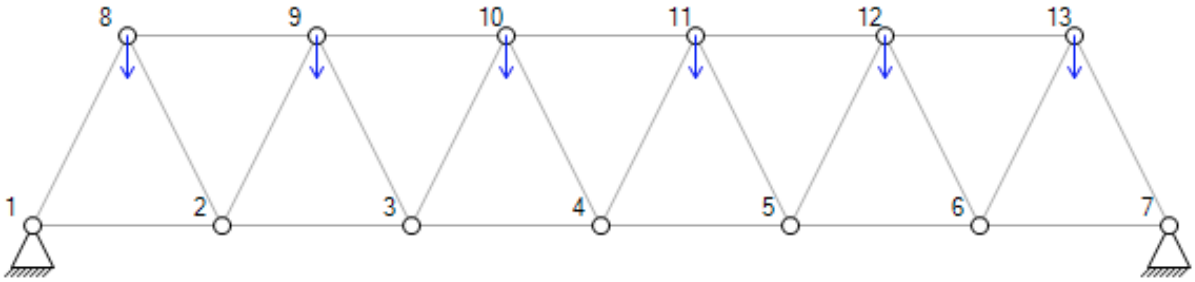


Figura 2: Treliça do programa *truss2d.py*.

Aplicando o Método dos Elementos Finitos para analisar a estrutura proposta, os seguintes passos foram empregados:

1. Adotou-se um sistema de referência global, nesse caso, o arquivo de entrada especificou-o como sendo a origem no nó 1. Além disso, os nós foram prontamente enumerados pela seção **COORDINATES*. Após isso, cada nó teve seus graus de liberdades numerados com auxílio da seção **BCNODES* que caracterizou os nós 1 e 7 como não tendo nenhum grau de liberdade, porém os outros nós possuem 2, na direção de X e de Y.

2. Numerou-se os elementos caracterizados pela incidência nodal da seção **INCIDENCES*. E partir disso, para cada um foram calculados seus comprimentos e cossenos para determinar a sua orientação.

3. Com base nos itens calculados anteriormente, comprimento, cosseno e seno, e a área de cada elemento, pode-se calcular a matriz de rigidez para elementos de barra plana em tracão de acordo

com a equação 1.

$$[K_e] = \frac{EA}{L}\begin{bmatrix} c^2 & cs & -c^2 & -cs \\ cs & s^2 & -cs & -s^2 \\ -c^2 & -cs & c^2 & cs \\ -cs & -s^2 & cs & s^2 \end{bmatrix} \tag{1}$$

4. Após ter calculado as matrizes de rigidez para cada elemento, determinou-se a matriz global de rigidez *K* sobrepondo-as de acordo com a incidência dos nós.

5. A partir das forças de reação a serem calculadas nos nós 1 e 7 e as forças externas sobre os nós 8, 9, 10, 11, 12 e 13, construiu-se o vetor global de forças concentradas a ser utilizado na equação 2, a fim de determinar os deslocamentos em cada grau de liberdade.

$$[K]\{U\} = \{F\} \tag{2}$$

6. Antes de calcular a equação 2, aplicou-se as condições de contorno nos nós 1 e 2, uma vez que não permitem deslocamentos em nenhuma das direções.

7. Finalmente, a equação 2 é resolvida e são obtidos os valores para as forças de reação e deslocamentos, os quais são utilizados para calcular as tensão internas de cada elemento.

$$\varepsilon = \frac{1}{L}\begin{bmatrix} -c & s & -s & c & s \end{bmatrix}\begin{bmatrix} u_1 \\ v_1 \\ u_2 \\ v_2 \end{bmatrix} \tag{3}$$

$$\sigma = \frac{\varepsilon}{L}\begin{bmatrix} -c & s & -s & c & s \end{bmatrix}\begin{bmatrix} u_1 \\ v_1 \\ u_2 \\ v_2 \end{bmatrix} \tag{4}$$

## 3.3 Programa

A partir do arquivo de entrada, o programa, escrito em *Python*, realizará todos os cálculos que serão armazenados em um arquivo de saída.

Primeiramente, deve-se importar as bibliotecas necessárias para o processamento do programa. A biblioteca *numpy* será utilizada para manipular as matrizes, pandas auxiliará na leitura do arquivo de entrada e as bibliotecas do *matplotlib* na visualização gráfica dos resultados.

```
# Importing libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.pylab as pl
from matplotlib.colors import LinearSegmentedColormap
from matplotlib import cm
```

Com as bibliotecas importadas, o próximo passo é ler o arquivo de entrada.

```
# Reading the input filename
br = 60 * '='
print(br + '\n' + 'TRUSS 2D' + '\n' + br + '\n' + '\n' + br + '\n' + br)
inpFileName = 'example1_truss'
fileName = inpFileName + '.fem'
with open(fileName, 'r') as myFile:
    file = myFile.read().splitlines()
myFile.close()
```

A variável inpFileName deve ser modificada para ler o arquivo desejado e a extensão '.fem' é adicionada dentro do código. O arquivo é então processado de forma que cada linha é armazenada como um elemento de um objeto tipo lista.

```
# Reading function
def readnum(keyword, document, type_num='int'):
    if keyword in document:
        totalnum = int(document[document.index(keyword) + 1])
        if totalnum > 0:
            num = pd.DataFrame(list(map(lambda x: x.split(),
                                        document[document.index(keyword) + 2:
                                                 document.index(keyword) + 2 + totalnum])
                                    )
                               ).astype(type_num).as_matrix()
        else:
            num = pd.DataFrame()
        return totalnum, num
    else:
```

```
        raise ValueError("{} Not Found.".format(keyword))


def removefirstcol(df):
    return df[:, 1:]
```

Foram definidas duas funções para ler os parâmetros da treliça. A função readnum lê as informações referentes a keyword (“*COORDINATES”, “*GROUP _ELEMENTS”, etc...) e retorna uma matriz e um número correspondente ao número de nós, número de grupos de elementos, etc..., pode-se também definir o tipo dos elementos da matriz, sendo que o tipo padrão é definido como inteiro. A função também levanta um erro no caso de não encontrar a keyword especificada.

A função removefirstcol remove a primeira coluna da matriz pois refere-se ao índice na matriz, sendo assim um valor irrelevante.

Assim, como foi definido no primeiro passo na seção MEF, podemos apresentar as coordenadas nodais da treliça de forma mais visual na tabela abaixo:

Tabela 1: Coordenadas nodais da treliça.

| Número do nó | X[m] | Y[m] |
|---|---|---|
| 1 | 0 | 0 |
| 2 | 500 | 0 |
| 3 | 1000 | 0 |
| 4 | 1500 | 0 |
| 5 | 2000 | 0 |
| 6 | 2500 | 0 |
| 7 | 3000 | 0 |
| 8 | 250 | 500 |
| 9 | 750 | 500 |
| 10 | 1250 | 500 |
| 11 | 1750 | 500 |
| 12 | 2250 | 500 |
| 13 | 2750 | 500 |

Após a leitura das variáveis de entrada, a matriz de graus de liberdade de cada nó é inicializada em 1, com número de linhas igual a *numNodalDOFs* e colunas igual a quantidade de nós e, posterior-

mente, os referentes aos apoios são igualados a 0. Obtêm-se,então, o total de graus de liberdade livres e restritos que são utilizados na numeração dos GL de cada nó. Como a treliça analisada é 2D o valor de *numNodalDOFs* é igual a 2.

```
# DOFs matrix initialized with ones
nodalDOFNumbers = np.ones((numNodalDOFs, totalNumNodes), dtype=int)

# Boundary conditions on dofs with zero values
nodalDOFNumbers[hdbcNodes[:numBCNodes, 1] - 1, hdbcNodes[:numBCNodes, 0] - 1] = 0

# Total number of DOFs for the model
totalNumDOFs = totalNumNodes*numNodalDOFs

# Number of free and restricted dofs
totalNumFreeDOFs = np.sum(nodalDOFNumbers == 1)
totalNumRestrDOFs = np.sum(nodalDOFNumbers == 0)

# DOFs numbering
ukeqnum = 0
keqnum = totalNumFreeDOFs
for i in range(totalNumNodes):
    for j in range(numNodalDOFs):
        if nodalDOFNumbers[j, i] == 1:
            ukeqnum = ukeqnum + 1
            nodalDOFNumbers[j, i] = ukeqnum
        elif nodalDOFNumbers[j, i] == 0:
            keqnum = keqnum + 1
            nodalDOFNumbers[j, i] = keqnum
```

Cria-se uma matriz auxiliar (*prop*) que apresenta o comprimento de cada elemento, assim como o valor de seno e cosseno correspondente, como explicado no segundo passo do Desenvolvimento Procedural. Utiliza-se a função *np.sqrt* para calcular a raiz quadrada e *np.absolute* para obter o valor absoluto.

```
# element lengths
lengths = np.sqrt(np.absolute((coords[incid[:, 0] - 1, 0] - coords[incid[:, 1] - 1, 0])**2 +
                              (coords[incid[:, 0] - 1, 1] - coords[incid[:, 1] - 1, 1])**2))
```

```
# lx = sin
lx = (coords[incid[:, 1] - 1, 1] - coords[incid[:, 0] - 1, 1]) / lengths

# ly = cos
ly = (coords[incid[:, 1] - 1, 0] - coords[incid[:, 0] - 1, 0]) / lengths

prop = np.matrix([lengths, lx, ly]).transpose()
```

A partir dos valores de x e y para dois nós, é possível calcular o comprimento de um elemento. Deve-se atentar ao fato que subtrai-se 1 do índice uma vez que a linguagem Python começa a contagem em 0. Os valores obtidos são apresentados na tabela 2.

Tabela 2: Incidência, comprimento, cosseno e seno diretores dos elementos de barra.

| Elemento | Incidência | l[cm] | c | s |
|---|---|---|---|---|
| 1 | 1-2 | 500.000 | 0.000 | 1.000 |
| 2 | 2-3 | 500.000 | 0.000 | 1.000 |
| 3 | 3-4 | 500.000 | 0.000 | 1.000 |
| 4 | 4-5 | 500.000 | 0.000 | 1.000 |
| 5 | 5-6 | 500.000 | 0.000 | 1.000 |
| 6 | 6-7 | 500.000 | 0.000 | 1.000 |
| 7 | 1-8 | 559.000 | 0.894 | 0.447 |
| 8 | 8-2 | 559.000 | -0.894 | 0.447 |
| 9 | 2-9 | 559.000 | 0.894 | 0.447 |
| 10 | 9-3 | 559.000 | -0.894 | 0.447 |
| 11 | 3-10 | 559.000 | 0.894 | 0.447 |
| 12 | 10-4 | 559.000 | -0.894 | 0.447 |
| 13 | 4-11 | 559.000 | 0.894 | 0.447 |
| 14 | 11-5 | 559.000 | -0.894 | 0.447 |
| 15 | 5-12 | 559.000 | 0.894 | 0.447 |
| 16 | 12-6 | 559.000 | -0.894 | 0.447 |
| 17 | 6-13 | 559.000 | 0.894 | 0.447 |
| 18 | 13-7 | 559.000 | 0.894 | -0.447 |
| 19 | 8-9 | 500.000 | 0.000 | 1.000 |
| 20 | 9-10 | 500.000 | 0.000 | 1.000 |
| 21 | 10-11 | 500.000 | 0.000 | 1.000 |
| 22 | 11-12 | 500.000 | 0.000 | 1.000 |
| 23 | 12-13 | 500.000 | 0.000 | 1.000 |

A partir dos valores da tabela 2, pode-se montar a matriz global de rigidez. Por meio de um *loop*, as matrizes de rigidez para cada elemento foram facilmente calculadas e já acrescentadas à matriz global. Agregando assim as etapas 3 e 4 da seção 3.2.

Primeiramente, a matriz de rigidez global e o número do elemento são inicializados em 0. No *loop*, determina-se o grupo de elementos e assim as propriedades como o módulo de elasticidade (E) e a área transversal, que são usados no cálculo da matriz de rigidez do elemento como apresentado na equação (1). A partir da numeração dos graus de liberdade do elemento, acrescenta-se a matriz *ke* à matriz global

*kg*.

```
# Assembling of the global stiffness matrix and load vector
# Creates zero global matrix
kg = np.zeros((totalNumDOFs, totalNumDOFs))
elemNum = 0
for grp in range(numGroups):
    e = maters[grp, 0]
    a = gps[grp, 0]
    for tne in range(groups[grp - 1, 0]):
        cc = prop[elemNum, 2]**2
        ss = prop[elemNum, 1]**2
        cs = prop[elemNum, 1]*prop[elemNum, 2]

        # Element stiffness matrix
        kedf = [[cc, cs, -cc, -cs],
                [cs, ss, -cs, -ss],
                [-cc, -cs, cc, cs],
                [-cs, -ss, cs, ss]]
        ke = np.dot((e * a / prop[elemNum, 0]), kedf)

        # Element DOFs
        elemEqs = nodalDOFNumbers[:, incid[elemNum, :] - 1]

        # Assembling into the global matrix
        index = np.concatenate(np.column_stack(elemEqs)) - 1
        kg[index, index.reshape((-1, 1))] = kg[index, index.reshape((-1, 1))] + ke
        elemNum = elemNum + 1
```

A matriz global de rigidez da treliça é então calculada e posteriormente a matriz dos carregamentos fg.

```
# Assembling global matrix
fg = np.zeros((totalNumDOFs, 1))
for i in range(numLoadedNodes):
    # element dofs
    elemEqs = nodalDOFNumbers[loads[i, 1] - 1, loads[i, 0] - 1]
```

```
    # assembling into the global vector
    fg[elemEqs - 1, 0] = loads[i, 2]
```

A partir de fg e kg, é possível resolver as equações de deslocamento (u), deformação (strain), tensão (stress) e forças de reação (rf) para cada elemento da treliça. Esses cálculos são realizados de forma direta pelo programa, que dispõe de métodos rápidos para a resolução de sistemas de equações matriciais. A matriz *u* de deslocamentos é primeiramente inicializada e calculada a partir da equação (2). A função *np.dot* realiza a multiplicação entre duas matrizes, *np.linalg.inv*, computa a inversa da matriz, *np.concatenate*, concatena duas matrizes em apenas uma, *np.row _stack*, empilha duas matrizes verticalmente e *np.array* cria uma matriz.

A partir dos resultados para os deslocamentos e as coordenadas de cada nó, pode-se calcula-se a posição dos nós da treliça deformada. Com a finalidade de tornar essa deformação mais visual, utiliza-se um fator de escala de 100. Por meio de um *loop*, determina-se a deformação e a tensão pelas equações (3) e (4).

```
# Solving systems of equations
u = np.zeros((totalNumDOFs, 1))
u[:totalNumFreeDOFs, 0] = np.dot(np.linalg.inv(kg[:totalNumFreeDOFs, :totalNumFreeDOFs]),
                                 fg[:totalNumFreeDOFs, 0])

# Calculates reaction forces, element strains and stresses
# reaction forces
rf = np.dot(kg[totalNumFreeDOFs:, :totalNumFreeDOFs], u[:totalNumFreeDOFs])

# Deformed coordinates
coordsd = coords + u[np.concatenate(np.row_stack(nodalDOFNumbers)) - 1].reshape(
    (nodalDOFNumbers.shape[0], nodalDOFNumbers.shape[1])).transpose()
scaleFactor = 100
coordsdd = coords + scaleFactor*u[np.concatenate(np.row_stack(nodalDOFNumbers)) - 1].reshape(
    (nodalDOFNumbers.shape[0], nodalDOFNumbers.shape[1])).transpose()

# Element strain and stress vectors
strain = np.zeros((totalNumElements, 1))
stress = np.zeros((totalNumElements, 1))
```

```
# Calculates strain and stress for the elements
elemNum = 0
for grp in range(numGroups):
    e = maters[grp, 0]
    for elNum in range(groups[grp - 1, 0]):
        # element dofs and displacements
        elemEqs = nodalDOFNumbers[:, incid[elemNum, :] - 1]
        index = np.concatenate(np.column_stack(elemEqs)) - 1
        elemU = u[index, 0]
        # strain
        strain[elemNum] = 1 / prop[elemNum, 0]*np.dot(
            np.array([[-prop[elemNum, 2], -prop[elemNum, 1],
                       prop[elemNum, 2], prop[elemNum, 1]]]), elemU)

        # stress
        stress[elemNum] = strain[elemNum]*e

        elemNum = elemNum + 1
```

A partir dos resultados obtidos são gerados quatro gráficos. O primeiro representa a treliça com os seus carregamentos e apoios. O segundo, as forças de reação dos apoios. O terceiro, a treliça deformada e colorida de acordo com a deformação ao longo de cada elemento. O quarto, a treliça deformada e colorida de acordo com a tensão em cada elemento.

Para auxiliar na formação dos gráficos, são criadas cinco funções, sendo que duas delas (*plot _gradient _hack* e *plot_gradient_rgb_pair*) estão disponível no repositório do github (IVANOV, 2012) A função *originalMesh* cria a treliça original; *nodes_mesh*, os nós da treliça; *plot_gradient_hack*, cria um gradiente entre duas cores para cada elemento; *plot_gradient_rgb_pair*, define as cores para cada valor do elemento e *rankmin* define esses valores entre dois pontos.

```
    # Original Mesh Function
def originalMesh(ax, num_elements=totalNumElements):
    for elemNum in range(num_elements):
        ax.plot(coords[incid[elemNum, :] - 1, 0], coords[incid[elemNum, :] - 1, 1],
                color='white', linewidth=1, alpha=0.5)
```

```
# Nodes of the Underformed Mesh Function
def nodes_mesh(ax, coords=coords, order=1):
    ax.scatter(x=coords[:, 0], y=coords[:, 1], c='yellow',
               s=10, alpha=0.5, zorder=order)


# Color gradient function
def plot_gradient_hack(p0, p1, npts=20, cmap=None, **kw):
    """
    Draw a gradient between p0 and p1 using a colormap
    The **kw dictionary gets passed to plt.plot, so things like linestyle,
    linewidth, labels, etc can be modified directly.
    """
    x_1, y_1 = p0
    x_2, y_2 = p1

    X = np.linspace(x_1, x_2, npts)
    Xs = X[:-1]
    Xf = X[1:]
    Xpairs = zip(Xs, Xf)

    Y = np.linspace(y_1, y_2, npts)
    Ys = Y[:-1]
    Yf = Y[1:]
    Ypairs = zip(Ys, Yf)

    C = np.linspace(0, 1, npts)
    cmap = plt.get_cmap(cmap)
    # the simplest way of doing this is to just do the following:
    for x, y, c in zip(Xpairs, Ypairs, C):
        plt.plot(x, y, '-', c=cmap(c), **kw)


def plot_gradient_rbg_pairs(p0, p1, rgb0, rgb1, **kw):
    """Form the gradient from RGB values at each point
    The **kw dictionary gets passed to plt.plot, so things like linestyle,
```

```
    linewidth, labels, etc can be modified directly.
    """
    cmap = LinearSegmentedColormap.from_list('jet', (rgb0, rgb1))
    plot_gradient_hack(p0, p1, cmap=cmap, **kw)


# Color definition Function
def rankmin(x):
    x = np.round(x, decimals=4)
    u, inv, counts = np.unique(x, return_inverse=True, return_counts=True)
    csum = np.zeros_like(counts)
    csum[1:] = counts[:-1].cumsum()
    return csum[inv]
```

Para cada gráfico, primeiro é definido a posição da imagem, no caso, duas por janela, posicionadas uma em cima da outra. São indicados os limites da imagem, assim como, o título e o nome de cada eixo. Posteriormente, traça-se a treliça original (*originalMesh*) e os nós (*nodes_mesh*).

```
# Mesh and boundary conditions
plt.style.use('dark_background')
fig1 = plt.figure()
plt.subplots_adjust(hspace=0.7, wspace=0.7)

ax1 = fig1.add_subplot(211)
ax1.set_title('Truss', fontsize=17)
plt.xlabel('X', fontsize=10)
plt.ylabel('Y', fontsize=10)
plt.xlim(min(coords[:, 0]) - 0.5*prop[0, 0], max(coords[:, 0]) + 0.5*prop[0, 0])
plt.ylim(min(coords[:, 1]) - 0.5*prop[0, 0], max(coords[:, 1]) + 0.5*prop[0, 0])

# Plots Original Mesh
originalMesh(ax1)

# Plots Nodes of the Undeformed Mesh
nodes_mesh(ax1)
```

Para plotar as setas correspondentes aos carregamentos e forças de reação, utiliza-se a função

*quiver*.

```
# Plot loads
fFe = 0.25 * loads[:, 2] * np.asscalar(abs(max(prop[:, 0]))/abs(max(loads[:, 2])))
for i in range(len(loads)):
    if loads[i, 1] == 1:
        ax1.quiver(coords[loads[i, 0] - 1, 0] - fFe[i], coords[loads[i, 0] - 1, 1],
                   fFe[i], 0, color='red', scale=1, units='xy', scale_units='xy',
                   headlength=10, headwidth=5)
    elif loads[i, 1] == 2:
        ax1.quiver(coords[loads[i, 0] - 1, 0], coords[loads[i, 0] - 1, 1] - fFe[i],
                   0, fFe[i], color='red', scale=1, units='xy', scale_units='xy',
                   headlength=10, headwidth=5)
```

Para gerar os gráficos de deformação e tensão, plota-se a treliça deformada e os elementos coloridos de acordo com o valor da deformação ou da tensão, pelas funções explicadas anteriormente.

```
### Plot Element Strains
fig2 = plt.figure()
plt.subplots_adjust(hspace=0.7, wspace=0.7)
ax3 = fig2.add_subplot(211)
ax3.set_title('Element Strains', fontsize=17)
plt.xlabel('X', fontsize=10)
plt.ylabel('Y', fontsize=10)
plt.xlim(min(coords[:, 0]) - 0.5*prop[0, 0], max(coords[:, 0]) + 0.5*prop[0, 0])
plt.ylim(min(coords[:, 1]) - 0.5*prop[0, 0], max(coords[:, 1]) + 0.5*prop[0, 0])
originalMesh(ax3)

# Plot deformed mesh
# plot deformed mesh and element strains
d_ef = np.abs(coords - coordsd)
d_ef = np.sum(d_ef, axis=1)
z = rankmin(d_ef)
colors = pl.cm.jet(np.linspace(0, 1, len(set(z))))
Z = np.unique(z, return_index=False)
for elemNum in range(totalNumElements):
    xy = coordsdd[incid[elemNum, :] - 1, :]
```

```
    plot_gradient_rbg_pairs(xy[0, :], xy[1, :],
                            colors[np.where(z[incid[elemNum, 0] - 1] == Z)[0][0]],
                            colors[np.where(z[incid[elemNum, 1] - 1] == Z)[0][0]])

sm = plt.cm.ScalarMappable(cmap=plt.cm.jet,
                           norm=plt.Normalize(vmin=d_ef.min(), vmax=d_ef.max()))
sm._A = []
plt.colorbar(sm, ax=ax3, ticks=np.linspace(d_ef.min(), d_ef.max(), num=10))

# plot nodes of deformed mesh
nodes_mesh(ax3, coordsdd, order=totalNumElements + 1)
```

A seguir o arquivo de saída é criado com o mesmo nome do arquivo de entrada, porém com extensão *.out*.

```
# Output File
outFileName = inpFileName + '.out2'
with open(outFileName, 'w') as f:
    f.write('*DISPLACEMENTS\n')
    for i in range(totalNumNodes):
        f.write('{:d} {:.4f} {:.4f}\n'.format(i + 1, u[nodalDOFNumbers[0, i] - 1, 0],
                                              u[nodalDOFNumbers[1, i] - 1, 0]))
    f.write('\n*ELEMENT_STRAINS\n')
    for i in range(totalNumElements):
        f.write('{:d} {:.6e}\n'.format(i + 1, strain[i, 0]))
    f.write('\n*ELEMENT_STRESSES\n')
    for i in range(totalNumElements):
        f.write('{:d} {:.6e}\n'.format(i + 1, stress[i, 0]))
    f.write('\n*REACTION_FORCES\n')
    for i in range(totalNumRestrDOFs):
        if hdbcNodes[i, 1] == 1:
            f.write('{:d} FX = {:.6e}\n'.format(hdbcNodes[i, 0], rf[i, 0]))
        else:
            f.write('{:d} FY = {:.6e}\n'.format(hdbcNodes[i, 0], rf[i, 0]))
    f.close()
```

O código inteiro encontra-se no anexo B.

## 3.4 Arquivo de Saída

O arquivo de saída gerado para o exemplo possui extensão *.out* e apresenta os cinco resultados no seguinte formato:

1. Deslocamento de cada nó, na direção X e Y.

   ***DISPLACEMENTS**

   <índice do nó> <deslocamento em X e Y>

2. Deformação de cada elemento.

   ***ELEMENT _STRAINS**

   <índice do nó> <deformação>

3. Tensão em cada elemento.

   ***ELEMENT _STRESS**

   <Índice do nó> <tensão>

4. Forças de reação nos apoios.

   ***REACTION _FORCES**

   Para cada apoio:

   <Índice do nó> <FX = ou FY = > <valor da força>

O arquivo completo encontra-se no apêndice C. Os resultados da tensão e deformação para cada elemento são apresentados na tabela 3.

Tabela 3: Deformação específica e tensão normal para cada elemento da treliça.

| Elemento | Deformação ($\varepsilon \times 10^{-}5$) | Tensão ($\sigma[kgf/cm^2]$) |
|---|---|---|
| 1 | -5.05 | -106.11 |
| 2 | 1.01 | 21.22 |
| 3 | 4.04 | 84.89 |
| 4 | 4.04 | 84.89 |
| 5 | 1.01 | 21.22 |
| 6 | -5.05 | -106.11 |
| 7 | -10.17 | -213.54 |
| 8 | 6.78 | 142.36 |
| 9 | -6.78 | 142.36 |
| 10 | 3.39 | 71.18 |
| 11 | -3.39 | -71.18 |
| 12 | 0.00 | 0.00 |
| 13 | 0.00 | 0.00 |
| 14 | -3.39 | -71.18 |
| 15 | 3.39 | 71.18 |
| 16 | -6.78 | -142.36 |
| 17 | 6.78 | 142.36 |
| 18 | -10.17 | -213.54 |
| 19 | -7.58 | -159.16 |
| 20 | -12.13 | -254.66 |
| 21 | -13.64 | -286.49 |
| 22 | -12.13 | -254.66 |
| 23 | -7.58 | -159.16 |

Os gráficos gerados pelo programa apresentando a treliça, as forças de reação, a deformação e a tensão calculadas são apresentados a seguir:

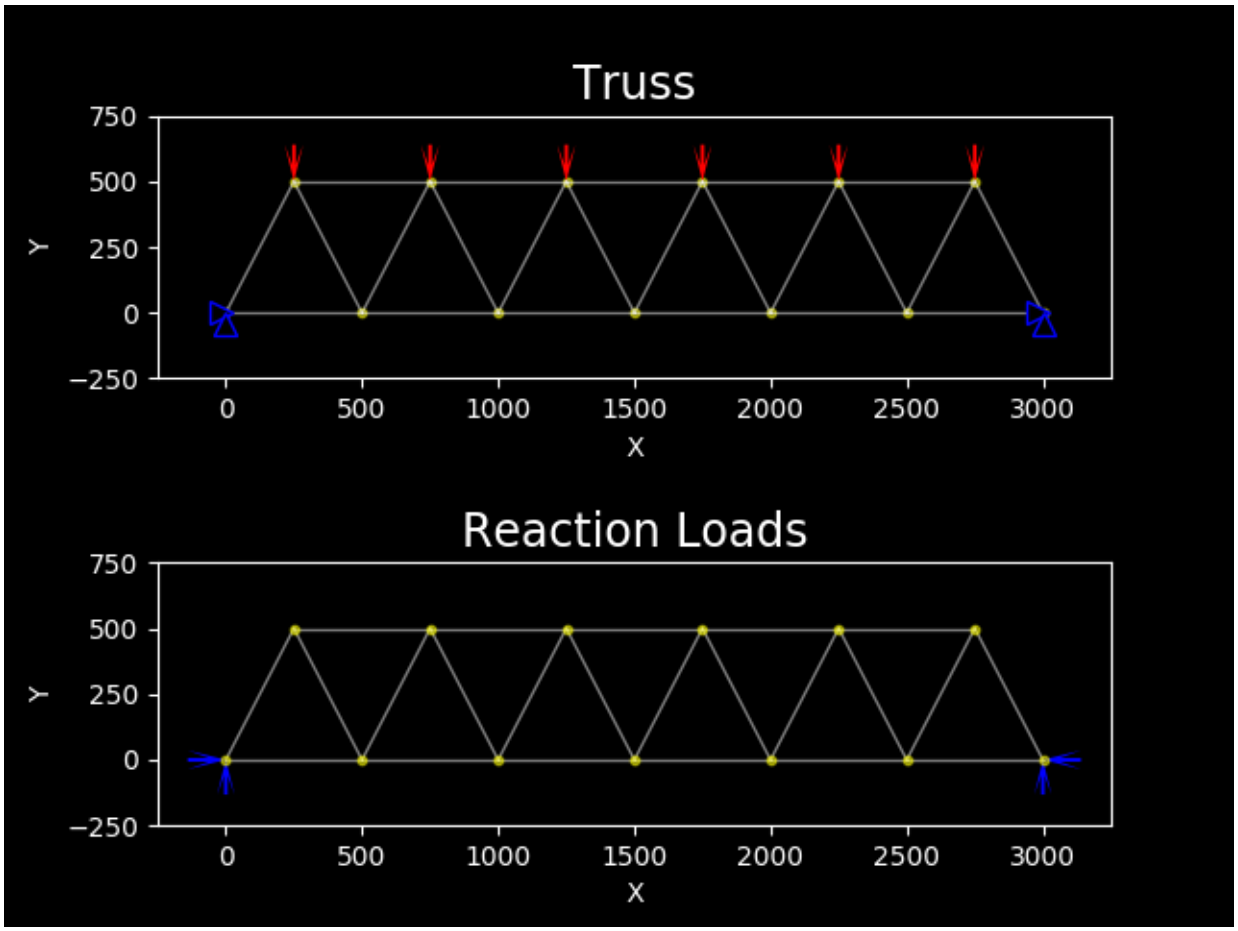


Figura 3: Treliça e forças de reação.

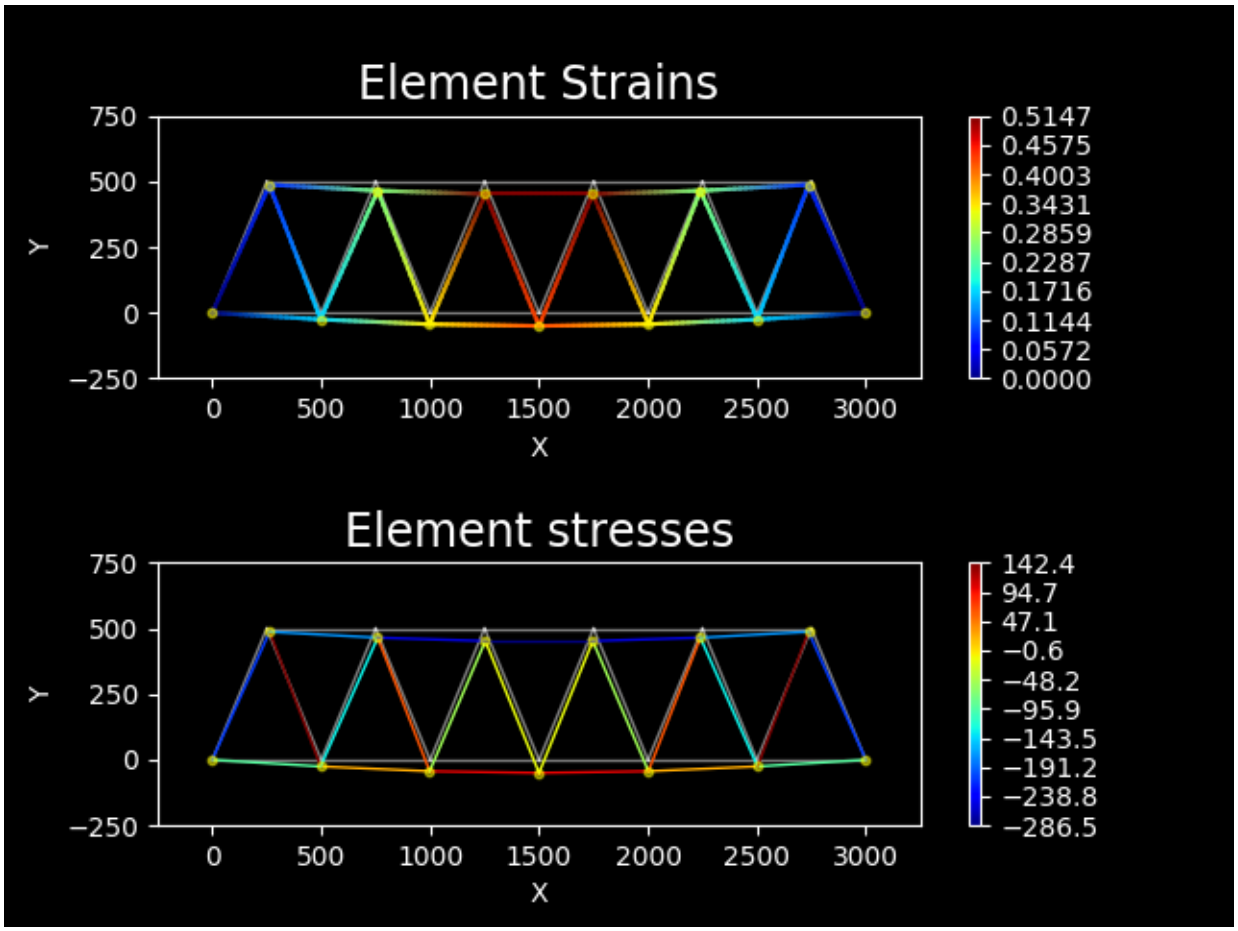


Figura 4: Treliça deformada representando a deformação e tensão.

# 4 Programa truss2d_design.py

O programa truss2d_design.py foi escrito a partir do truss2d.py com pequenas modificações de forma que aceitasse uma variável de entrada nova, *Design Iterations*, que define o número máximo de iterações que podem ser feitas a fim de calcular a melhor área dos elementos de barra que possam suportar as cargas sem ultrapassar a tensão ou compressão máxima.

O arquivo de saída terá dois novos resultados. A área e o volume de cada elemento de barra para cada iteração.

A partir dos resultados apresentados na tabela 3 e os valores de tensão máxima (120 $kgf/cm^2$) e compressão mínima (80 $kgf/cm^2$), observa-se que os elementos 6, 7, 8, 9, 16, 17, 18, 19, 20, 21, 22, e 23 ultrapassam esses limites, de forma que o redimensionamento das barras permitiria o projeto de uma estrutura mais adequada para suportar as forças externas propostas para o exemplo. Esse dimensionamento é feito dividindo o módulo das tensões que ultrapassam os limites pela tensão admissível, isso resulta em uma nova área do elemento.

## 4.1 Arquivo de Entrada

O arquivo de entrada desse novo programa tem apenas uma variável a mais, apresentado da seguinte maneira:

8. Número máximo de iterações para o design:
   ***DESIGN_ITERATIONS**

## 4.2 Programa

A primeira modificação feita foi adicionar um código para a leitura da variável de iterações máximas.

```
# Reading maximum design iterations
print('READING DESIGN ITERATIONS...')
if '*DESIGN_ITERATIONS' in file:
    maxNumDesignIter = int(file[file.index('*DESIGN_ITERATIONS') + 1])
else:
    raise ValueError('{} Not Found.'.format('*DESIGN_ITERATIONS'))
```

Para poder realizar os cálculos de modo iterativo, foi inserido um *loop* de forma que, enquanto o número máximo de iterações não for atingido ou a tensão máxima não for ultrapassada, os cálculos serão refeitos com uma nova área. Dessa forma, encontram-se as propriedades geométricas ideais para a treliça suportar as cargas definidas.

```
# Areas and Materials Properties for each element
areas = np.zeros((totalNumElements, maxNumDesignIter))
matProp = np.zeros((totalNumElements, 3))
elemNum = 0
for grp in range(numGroups):
    for el in range(int(groups[grp])):
        areas[elemNum, 0] = gps[grp, 0]
        matProp[elemNum, :] = maters[grp, :3]
        elemNum = elemNum + 1

# Element Strain and Stress Vectors
strain = np.zeros((totalNumElements, maxNumDesignIter))
stress = np.zeros((totalNumElements, maxNumDesignIter))

# Assembling global load vector
fg = np.zeros((totalNumDOFs, 1))
for i in range(numLoadedNodes):
    # element dofs
    elemEqs = nodalDOFNumbers[loads[i, 1] - 1, loads[i, 0] - 1]

    # assembling into the global vector
    fg[elemEqs - 1, 0] = loads[i, 2]

# Global displacement vector
u = np.zeros((totalNumDOFs, 1))

# Sets number of design iterations and design flag (=1, new design; 0= final design)
nDIt = 1
designFlag = 1

while designFlag:
```

```
# Creates zero global matrix and load vector
kg = np.zeros((totalNumDOFs, totalNumDOFs))
for elemNum in range(totalNumElements):
    # Youngs modulus and cross section area
    e = matProp[elemNum, 0]
    a = areas[elemNum, nDIt - 1]

    # director co-sine
    cc = prop[elemNum, 2]**2
    ss = prop[elemNum, 1]**2
    cs = prop[elemNum, 1]*prop[elemNum, 2]

    # element stiffness matrix
    kedf = [[cc, cs, -cc, -cs],
            [cs, ss, -cs, -ss],
            [-cc, -cs, cc, cs],
            [-cs, -ss, cs, ss]]
    ke = np.dot((e * a / prop[elemNum, 0]), kedf)

    # element dofs
    elemEqs = nodalDOFNumbers[:, incid[elemNum, :] - 1]

    # assembling into the global matrix
    index = np.concatenate(np.column_stack(elemEqs)) - 1
    kg[index, index.reshape((-1, 1))] = kg[index, index.reshape((-1, 1))] + ke

# Solving systems of equations
u[:totalNumFreeDOFs, 0] = np.dot(np.linalg.inv(kg[:totalNumFreeDOFs,
                                                  :totalNumFreeDOFs]),
                                 fg[:totalNumFreeDOFs, 0])

# Calculates strain and stress or the elements
for elemNum in range(totalNumElements):

    # Youngs modulus
    e = matProp[elemNum, 0]
```

```
    # Elements DOFs and displacements
    elemEqs = nodalDOFNumbers[:, incid[elemNum, :] - 1]
    index = np.concatenate(np.column_stack(elemEqs)) - 1
    elemU = u[index, 0]

    # Strain
    strain[elemNum, nDIt - 1] = 1 / prop[elemNum, 0]*np.dot(np.array([[-prop[elemNum, 2],
                                                                      -prop[elemNum, 1],
                                                                      prop[elemNum, 2],
                                                                      prop[elemNum, 1]]]),
                                                            elemU)

    # Stress
    stress[elemNum, nDIt - 1] = strain[elemNum, nDIt - 1]*e

# if the maximum number of design iterations is reached, finish the design process
# checks if the stresses for all bars are below the admissable limits or
# if the maximum number of iterations has been reached
tractionBars = stress[:, nDIt - 1] >= 0
compressionBars = stress[:, nDIt - 1] < 0
if (sum(stress[tractionBars, 0] <= matProp[tractionBars, 1]) +
    sum(stress[compressionBars, 0] >= -matProp[compressionBars, 2]) == totalNumElements) \
        or (nDIt + 1 >= maxNumDesignIter):
        designFlag = 0
# otherwise calculate the new areas
else:
    # increments number of design iterations
    nDIt += 1

    # copies areas from the previous iteration
    areas[:, nDIt - 1] = areas[:, nDIt - 2]

    # bars under compression with stress less than admissable
    # compression stress. They will be redesign.
    newBars = stress[:, nDIt - 2] < - matProp[:, 2]
```

```
# new areas for bars under compression
areas[newBars, nDIt - 1] = (- stress[newBars, nDIt - 2] /
                            matProp[newBars, 2] *
                            areas[newBars, nDIt - 2])

# bars under traction with stress larger than admissable
# tensile stress. They will be redesigned.
newBars = stress[:, nDIt - 2] > matProp[:, 1]

# new areas for bars under traction
areas[newBars, nDIt - 1] = (stress[newBars, nDIt - 2] /
                            matProp[newBars, 1] *
                            areas[newBars, nDIt - 2])
```

O programa segue a mesma metodologia apresentada na seção 3.2 Porém, no final do *loop*, é verificado se o número de iterações máximas ou os limites da tensão normal admitida são atingidos. Caso isso seja negativo, as novas áreas para os elementos que não atendam esses limites são calculadas utilizando a tensão obtida, a área antiga e a tensão permitida.

Os códigos utilizados para a parte do pós-processamento correspondem à mesma lógica apresentada na final da seção 3.3. O programa inteiro encontra-se no apêndice D.

## 4.3 Arquivo de Saída

O arquivo de saída apresenta duas informações adicionais em relação ao programa anterior, a área e o volume de cada elemento, além dos valores da tensão e deformação para cada iteração.

5. Área transversal de cada elemento:
   ***AREAS**
   <número de iterações realizadas>
   Para cada iteração:
   <número do elemento> <área>
6. Volume da treliça por inteiro:
   <número de iterações realizadas>
   ***VOLUMES**

<número de iterações>

Para cada iteração:

<volume>

O arquivo completo encontra-se no apêndice E. As tensões calculadas para cada uma das iterações são apresentadas na tabela 4.

Tabela 4: Tensões nos elementos a cada iteração.

| Elemento | $\sigma[kgf/cm^2]$ (iter. 1) | $\sigma[kgf/cm^2]$ (iter. 2) | $\sigma[kgf/cm^2]$ (iter. 3) | $\sigma[kgf/cm^2]$ (iter. 4) |
|---|---|---|---|---|
| 1 | -1.061064e+02 | -8.714724e+01 | -8.183750e+01 | -8.046448e+01 |
| 2 | 2.122128e+01 | 1.174169e+01 | 9.086823e+00 | 8.400314e+00 |
| 3 | 8.488514e+01 | 7.540555e+01 | 7.275068e+01 | 7.206417e+01 |
| 4 | 8.488514e+01 | 7.540555e+01 | 7.275068e+01 | 7.206417e+01 |
| 5 | 2.122128e+01 | 1.174169e+01 | 9.086823e+00 | 8.400314e+00 |
| 6 | -1.061064e+02 | -8.714724e+01 | -8.183750e+01 | -8.046448e+01 |
| 7 | -2.135351e+02 | -8.000000e+01 | -8.000000e+01 | -8.000000e+01 |
| 8 | 1.423567e+02 | 1.200000e+02 | 1.200000e+02 | 1.200000e+02 |
| 9 | -1.423567e+02 | -8.000000e+01 | -8.000000e+01 | -8.000000e+01 |
| 10 | 7.117835e+01 | 7.117835e+01 | 7.117835e+01 | 7.117835e+01 |
| 11 | -7.117835e+01 | -7.117835e+01 | -7.117835e+01 | -7.117835e+01 |
| 12 | 0.000000e+00 | 1.138412e-13 | 3.415237e-13 | 2.276825e-13 |
| 13 | 4.553649e-13 | 1.138412e-13 | 1.138412e-13 | 1.138412e-13 |
| 14 | -7.117835e+01 | -7.117835e+01 | -7.117835e+01 | -7.117835e+01 |
| 15 | 7.117835e+01 | 7.117835e+01 | 7.117835e+01 | 7.117835e+01 |
| 16 | -1.423567e+02 | -8.000000e+01 | -8.000000e+01 | -8.000000e+01 |
| 17 | 1.423567e+02 | 1.200000e+02 | 1.200000e+02 | 1.200000e+02 |
| 18 | -2.135351e+02 | -8.000000e+01 | -8.000000e+01 | -8.000000e+01 |
| 19 | -1.591596e+02 | -8.000000e+01 | -8.000000e+01 | -8.000000e+01 |
| 20 | -2.546554e+02 | -8.000000e+01 | -8.000000e+01 | -8.000000e+01 |
| 21 | -2.864873e+02 | -8.000000e+01 | -8.000000e+01 | -8.000000e+01 |
| 22 | -2.546554e+02 | -8.000000e+01 | -8.000000e+01 | -8.000000e+01 |
| 23 | -1.591596e+02 | -8.000000e+01 | -8.000000e+01 | -8.000000e+01 |

É possível perceber que, após as 5 iterações, alguns elementos ainda se encontram fora dos

limites de tensão máxima e compressão mínima, sendo necessárias iterações adicionais para atingir o resultado desejado.

Os primeiros dois gráficos gerados não mudaram em relação ao programa anterior, uma vez que não teve nenhuma modificação nas coordenadas ou nas forças sob a treliça. Na figura 5, são apresentadas as novas deformações e tensões sobre a treliça após as 5 iterações.

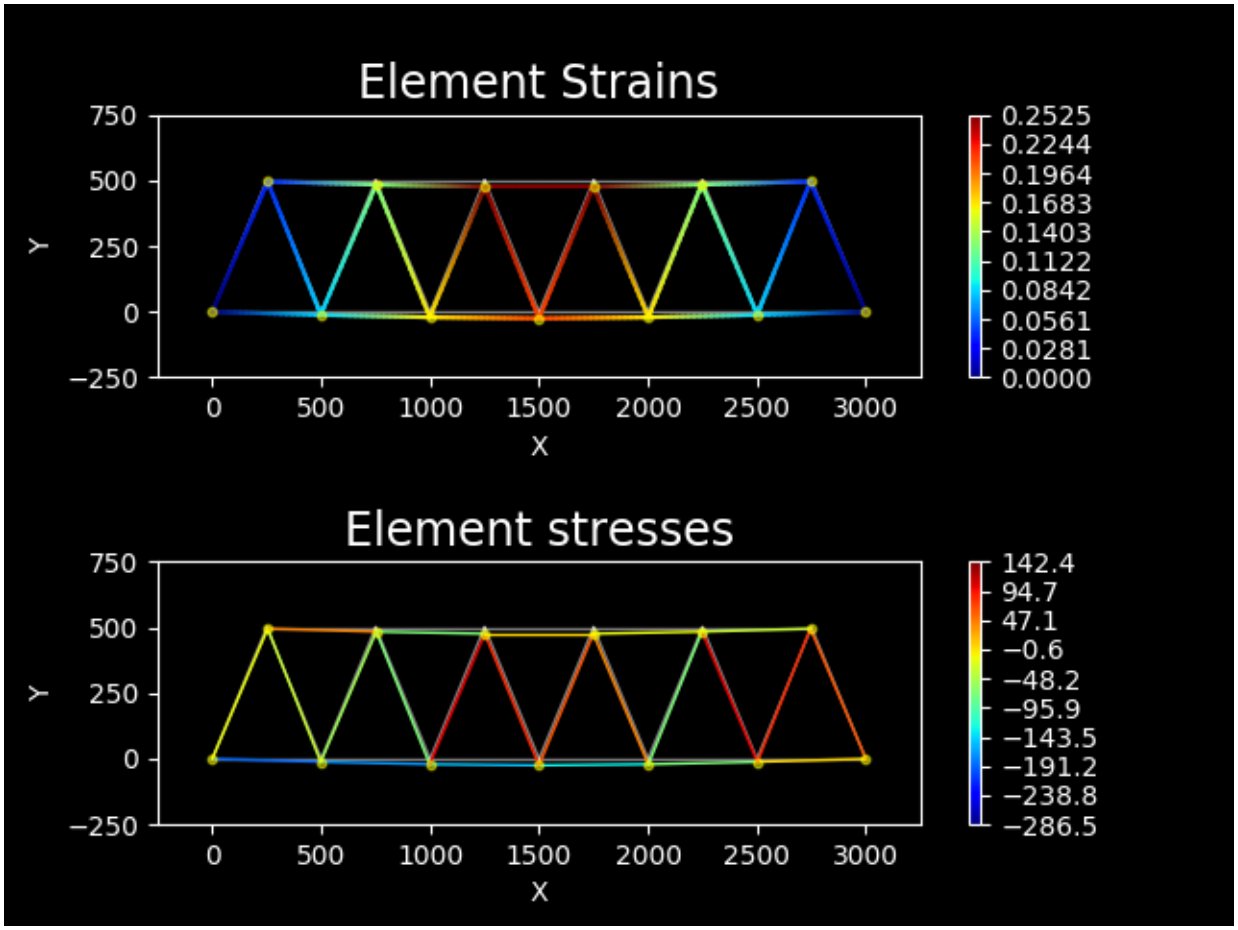


Figura 5: Deformação e tensão da treliça.

# 5 Programa portico.py

O programa portico.py tem como principal objetivo estudar estruturas compostas por elementos lineares que possuem conexões rígidas nos nós, de forma que duas barras conectadas por um nó tem deslocamentos e rotações compatíveis na ligação. Assim, esse tipo de estrutura pode suportar esforços normais, cortantes e, principalmente, esforços de flexão.

No dia a dia, essas estruturas podem ser observadas na entrada de casas e de municípios, além de ser uma construção comum na arquitetura grega e romana, como o Parthenon. Mas também são utilizadas internamente em composições para prover rigidez às construções, uma vez que resulta em estruturas hiperestáticas. Sendo assim, importante o seu estudo, principalmente, para a área de engenharia civil.

(a) Parthenon.

(b) Guindaste de pórtico.

Figura 6: Exemplos de pórticos.

## 5.1 Arquivo de Entrada

Baseado no arquivo de entrada do programa truss2d.py, modifica-se os itens 5 e 7, no primeiro, adiciona-se duas novas propriedades geométricas para cada grupo de elementos (deflexão máxima e mínima) e, no segundo, um terceiro grau de liberdade (rotação no eixo z). Além disso, adiciona-se um item a mais que caracteriza as cargas distribuídas sobre a estrutura.

Para a demonstração do programa *portico.py*, será empregado um arquivo de entrada (apêndice F) diferente dos dois programas anteriores.

5. Propriedades geométricas como a área transversal, momento de inércia em relação ao eixo z e as deflexões mínima e máxima de cada elemento. Cada linha corresponde ao número de grupos de elementos.
   ***GEOMETRIC_PROPERTIES**
   <número de grupos>
   <área da seção transversal> <Momento de inércia> <ymin> <ymax>

7. Carregamentos pontuais nas direções X e Y ou momento nos nós.

   ***LOADS**

   <número de nós carregados>

   <número do nó> <grau de liberdade, 1 = UX, 2 = UY e 3 = ROTZ> <carregamento>

8. Carregamentos distribuídos

   ***DISTRIBUTED_LOADS**

   <número de elementos carregados>

   <número do elemento> <força distribuída em X> <força distribuída em Y>

## 5.2 Desenvolvimento Procedural

O pórtico definido pelo arquivo de entrada apresentado na seção anterior é exibido na figura 6.

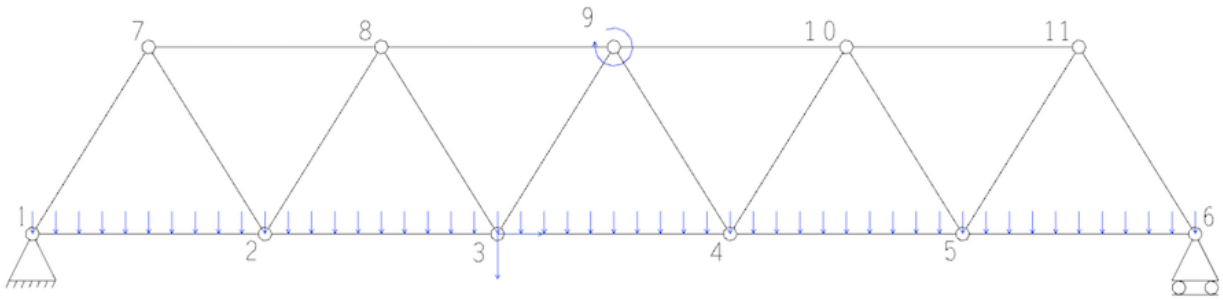


Figura 7: Pórtico do programa *portico.py*.

No caso da estrutura do pórtico, considera-se que cada elemento é uma viga, pois além de estar sob efeito de tração também tem flexão agindo simultaneamente. No caso apresentado, há duas cargas pontuais atuando sobre o nó 3, uma em cada direção (X e Y), uma carga distribuída ao longo das vigas inferiores horizontais e um momento agindo no sentido horário no nó 9.

Como se trata de uma viga, uma abordagem ligeiramente diferente é adotada utilizando o Método dos Elementos Finitos. Podemos seguir os mesmos passos sugeridos na seção 3.2 com algumas modificações.

Deve-se atentar para o fato de agora haver 3 graus de liberdade para cada nó, deslocamento na direção X e Y e rotação na direção Z. Além disso, a matriz de rigidez de cada elemento é modificada de forma que abranja tanto a tração quanto a flexão, isso é feito sobrepondo a equação (1), que considera

apenas a tração, e a equação (5), que considera apenas a flexão, resultando na equação (6).

$$[\overline{K}_e] = \frac{EI_z}{L^3}\begin{bmatrix} 12 & 6L & -12 & 6L \\ 6L & 4L^2 & -6L & 2L^2 \\ -12 & -6L & 12 & -6L \\ 6L & 2L^2 & -6L & 4L^2 \end{bmatrix} \tag{5}$$

$$[\overline{K}_e] = \begin{bmatrix} k_b & 0 & 0 & -k_b & 0 & 0 \\ 0 & 12k_f & 6Lk_f & 0 & -12k_f & 6Lk_f \\ 0 & 6Lk_f & 4L^2k_f & 0 & -6Lk_f & 2L^2k_f \\ -k_b & 0 & 0 & k_b & 0 & 0 \\ 0 & -12k_f & -6Lk_f & 0 & 12k_f & -6Lk_f \\ 0 & 6Lk_f & 2L^2k_f & 0 & -6Lk_f & 4L^2k_f \end{bmatrix} \tag{6}$$

Sendo que as constantes de rigidez elástica à tração e à flexão são dadas pelas equações 5 e 6, respectivamente.

$$k_f = \frac{EI_z}{L^3} \tag{7}$$

$$k_b = \frac{EA}{L} \tag{8}$$

Apesar da equação (1) já representar a matriz de rigidez do sistema global, a equação (6) não o faz, de forma que é necessário realizar a transformação de coordenadas, como apresentado a seguir:

$$\begin{bmatrix} \overline{u}_1 \\ \overline{v}_1 \\ \overline{\theta}_1 \\ \overline{u}_2 \\ \overline{v}_2 \\ \overline{\theta}_2 \end{bmatrix} = \begin{bmatrix} cos\theta & sin\theta & 0 & 0 & 0 & 0 \\ -sin\theta & cos\theta & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & cos\theta & sin\theta & 0 \\ 0 & 0 & 0 & 0 & -sin\theta & cos\theta \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} u_1 \\ v_1 \\ \theta_1 \\ u_2 \\ v_2 \\ \theta_2 \end{bmatrix} \tag{9}$$

Ou, simplificadamente, $[\overline{u}_e] = [T][u_e]$

Finalmente, podemos resolver $[T]^T[K_e][T][u_e] = [T]^T[\overline{F}_e]$ que também pode ser escrita como:

$$[K_e][u_e] = [F_e] \tag{10}$$

Resolvendo a equação (10), obtemos os deslocamentos em X e em Y e a rotação em Z, assim como a força axial, força cortante e momento fletor para cada elemento uma vez que $[F_e]$ corresponde ao

vetor da equação (11).

$$[F_e] = \begin{bmatrix} P_1 \\ F_1 \\ M_1 \\ P_2 \\ F_2 \\ M_2 \end{bmatrix} \tag{11}$$

## 5.3 Programa

Baseado no programa truss2d.py, a primeira modificação que deve ser feita é executar a função readnum para obter o valor das variáveis de carga distribuída, além das mesmas variáveis apresentadas anteriormente.

```
# Reading Distributed Loads
print('READING DISTRIBUTED LOADS...')
numLoadedElements, distLoads = readnum('*DISTRIBUTED_LOADS', file)
```

O cálculo para o comprimento dos elementos continua o mesmo. Porém, adiciona-se a inicialização da matrix global de cargas distribuídas.

```
# Distributed Load Global Matrix
globalDistLoads = np.zeros((totalNumElements, 2))

# Organize distributed loads in a matrix with all elements
globalDistLoads[distLoads[:, 0] - 1, :] = distLoads[:, 1:3]
```

Após isso, é feita a caracterização dos nós correspondentes ao apoios do pórtico, o cálculo dos graus de liberdade e a numeração deles de acordo com cada nó, assim como nos dois programas anteriores.

A seguir é apresentado o algoritmo para o cálculo da matrix global de cargas distribuídas e posteriormente das cargas pontuais. Como foi apresentado na seção anterior, a matriz de transformação de coordenadas é utlizada no cálculo da matriz de ridigez para que seja representada no sistema global de coordenadas.

```
# Assembling of the Global Stiffness Matrix and Global Distributed Loads Vector
```

```
# Creates Zero Global Matrix
kg = np.zeros((totalNumDOFs, totalNumDOFs))

# Distributed Loads
fgDistLoads = np.zeros((totalNumDOFs, 1))

elemNum = 0
for grp in range(0, numGroups):
    E = maters[grp, 0]
    rho = maters[grp, 3]
    A = gps[grp, 0]
    Iz = gps[grp, 1]

    for tne in range(0, groups[grp, 0]):
        c = prop[elemNum, 2]
        s = prop[elemNum, 1]
        l = prop[elemNum, 0]

        # Rotation Matrix
        T = np.array(([ c, s, 0,  0, 0, 0],
                      [-s, c, 0,  0, 0, 0],
                      [ 0, 0, 1,  0, 0, 0],
                      [ 0, 0, 0,  c, s, 0],
                      [ 0, 0, 0, -s, c, 0],
                      [ 0, 0, 0,  0, 0, 1]))

        # Element Stiffness Matrix
        # Axial Forces
        kb = E * A / l
        # Transversal Forces
        kf = E * Iz / l ** 3
        ke = np.array(([ kb,          0,              0, -kb,          0,              0],
                       [  0,    12 * kf,      6 * l * kf,   0,    -12 * kf,      6 * l * kf],
                       [  0, 6 * l * kf, 4 * l ** 2 * kf,   0, -6 * l * kf, 2 * l ** 2 * kf],
                       [-kb,          0,              0,  kb,          0,              0],
                       [  0,   -12 * kf,     -6 * l * kf,   0,     12 * kf,      -6 * l *kf],
```

```
                [  0, 6 * l * kf, 2 * l ** 2 * kf,   0, -6 * l * kf, 4 * l ** 2 * kf]))
        ke = np.dot(np.dot((T.transpose()), ke), T)

        # Element DOFs
        elemEqs = nodalDOFNumbers[:, incid[elemNum, :] - 1]
        elemEqs = elemEqs.flatten('F').reshape(elemEqs.shape[0] * elemEqs.shape[1], 1)

        # Assembling into the Global Matrix
        kg[elemEqs - 1, elemEqs.reshape((-1)) - 1] = kg[elemEqs - 1,
                                                        elemEqs.reshape((-1)) - 1] + ke

        # Distributed Loads
        # Applied Distributed Load Rotation Matrix
        Tq = np.array(([ c, s],
                       [-s, c]))

        # Element Distributed Load Vector
        q = np.dot(Tq, np.array((globalDistLoads[elemNum, 0],
                                 globalDistLoads[elemNum, 1] - rho * A * 9.81)))

        # Element Nodal Distributed Load Vector
        fe = l / 2 * np.array(([q[0], q[1], q[1] * l / 6, q[0], q[1], -q[1] * l / 6]))
        fe = np.dot(T.transpose(), fe)

        # Element DOFs
        elemEqs = nodalDOFNumbers[:, incid[elemNum, :] - 1]
        elemEqs = elemEqs.flatten('F')

        # Assembling into the Global Matrix
        fgDistLoads[elemEqs - 1, 0] = fgDistLoads[elemEqs - 1, 0] + fe

        elemNum += 1

# Assembling Global Load Vector
fgnLoads = np.zeros((totalNumDOFs, 1))
```

```
# Assembling Nodal Loads
for i in range(0, numLoadedNodes):
    # Element DOFs
    elemEqs = nodalDOFNumbers[loads[i, 1] - 1, loads[i, 0] - 1]

    # Assebmbling into the Global Vector
    fgnLoads[elemEqs - 1] = loads[i, 2]

fg = fgnLoads + fgDistLoads
```

Com a matriz global de rigidez calculada, pode-se resolver o sistema da equação (10) para obter os deslocamentos e rotações em cada grau de liberdade como apresentado a seguir:

```
# Solving system of equations
u = np.zeros((totalNumDOFs, 1))

u[0:totalNumFreeDOFs, :] = np.dot(np.linalg.inv(kg[:totalNumFreeDOFs,
                                                   :totalNumFreeDOFs]),
                                  fg[:totalNumFreeDOFs, 0].reshape(totalNumFreeDOFs, 1))
```

As forças de reação são então calculadas assim como as posições dos nós da estrutura deformada.

```
# Calculates Reaction Forces, Element Internal Loads
# Reaction Forces
rf = (np.dot(kg[totalNumFreeDOFs:, 0:totalNumFreeDOFs], u[0:totalNumFreeDOFs]) -
      fg[totalNumFreeDOFs:])

# Deformed Coordinates
coordsd = coords + u[np.concatenate(nodalDOFNumbers[0:2, :]) - 1].reshape(coords.shape[1],
                                                                          coords.shape[0]).transpose()
scaleFactor = 8e2
coordsdd = coords + scaleFactor * u[np.concatenate(nodalDOFNumbers[0:2, :]
                                                   ) - 1].reshape(coords.shape[1],
                                                                  coords.shape[0]).transpose()
```

No passo final, são calculados a força normal e força cortante para cada elemento, assim como

o momento fletor para cada nó do elemento e a tensão correspondente ao y máximo e mínimo da seção tranversal das vigas.

```
# Element Normal Force, Shear Force, Internal Moment and Stresses for y-max and for y-min
nx = np.zeros((totalNumElements, 1))
vy = np.zeros((totalNumElements, 1))
mz = np.zeros((totalNumElements, 2))
stress_ymax = np.zeros((totalNumElements, 2))
stress_ymin = np.zeros((totalNumElements, 2))

# Calculates Strain and Stress for the Elements
elemNum = 0
for grp in range(0, numGroups):
    E = maters[grp, 0]
    A = gps[grp, 0]
    Iz = gps[grp, 1]
    ymax = gps[grp, 2]
    ymin = gps[grp, 3]

    for tne in range(0, groups[grp, 0]):
        c = prop[elemNum, 2]
        s = prop[elemNum, 1]
        l = prop[elemNum, 0]

        # Rotation Matrix
        T = np.array(([ c, s, 0,  0, 0, 0],
                      [-s, c, 0,  0, 0, 0],
                      [ 0, 0, 1,  0, 0, 0],
                      [ 0, 0, 0,  c, s, 0],
                      [ 0, 0, 0, -s, c, 0],
                      [ 0, 0, 0,  0, 0, 1]))

        # Element DOFs and Displacements
        elemEqs = nodalDOFNumbers[:, incid[elemNum, :] - 1]
        elemU = u[elemEqs[:] - 1].flatten('F')
        elemU = np.dot(T, elemU)
```

```
# Normal Force
nx[elemNum] = E * A * (-1 / l * elemU[0] + 1 / l * elemU[3])

# Shear Force
vy[elemNum] = E * Iz * 6 / l ** 2 * (2 / l * elemU[1] + elemU[2] -
                                     2 / l * elemU[4] + elemU[5])

# Moments in Nodes 1 and 2
mz[elemNum, 0] = E * Iz * (-6 / l ** 2 * elemU[1] - 4 / l * elemU[2] +
                           6 / l ** 2 * elemU[4] - 2 / l * elemU[5])
mz[elemNum, 1] = E * Iz * ( 6 / l ** 2 * elemU[1] + 2 / l * elemU[2] +
                           -6 / l ** 2 * elemU[4] + 4 / l * elemU[5])

# Stress
stress_ymax[elemNum, 0] = nx[elemNum] / A - mz[elemNum, 0] / Iz * ymax
stress_ymax[elemNum, 1] = nx[elemNum] / A - mz[elemNum, 1] / Iz * ymax
stress_ymin[elemNum, 0] = nx[elemNum] / A - mz[elemNum, 0] / Iz * ymin
stress_ymin[elemNum, 1] = nx[elemNum] / A - mz[elemNum, 1] / Iz * ymin

elemNum += 1
```

Após de finalizado o processamento dos cálculos, é gerado o arquivo de saída com os resultados e, posteriormente, os gráficos para visualização do pórtico. Os códigos utilizados para a parte do pós-processamento correspondem à mesma lógica apresentada na final da seção 3.3. O programa inteiro encontra-se no apêndice G.

## 5.4 Arquivo de Saída

O arquivo de saída gerado pelo programa resulta em um arquivo de extensão .out (apêndice H) no seguinte formato:

1. Deslocamentos dos elementos nos eixos x e y e rotação no eixo z (positivo no sentido anti-horário).
   ***DISPLACEMENTS**
   Para cada elemento:
   <índice do nó> <deslocamento em X> <deslocamento em Y> <rotação em Z>

2. Forças de reação nos respectivos nós, FX, FY ou MZ.

   ***REACTION _FORCES**

   Para cada nó:

   <índice do nó> <FX, FY ou MZ = > <valor da força>

3. Tensão normal em cada elemento.

   ***ELEMENT _NORMAL _FORCE**

   Para cada elemento:

   <índice do elemento> <força normal>

4. Tensão de cisalhamento em cada elemento.

   ***ELEMENT _SHEAR _FORCE**

   Para cada elemento:

   <índice do elemento> <força de cisalhamento>

5. Momento fletor em cada nó de cada elemento.

   ***ELEMENT _MOMENT _LIMITS**

   Para cada elemento:

   <índice do elemento> <momento fletor no nó i> <momento fletor no nó j>

6. Deflexão máxima em UY para cada elemento.

   ***ELEMENT _STRESSES _YMAX**

   Para cada elemento:

   <índice do elemento> <deflexão máxima>

7. Deflexão mínima em UY para cada elemento.

   ***ELEMENT _STRESSES _YMIN**

   Para cada elemento:

   <índice do elemento> <deflexão mínima>

Após calcular as tensões normais, cortantes, forças de reação e deformações de cada elemento, foram gerados os seguintes gráficos:

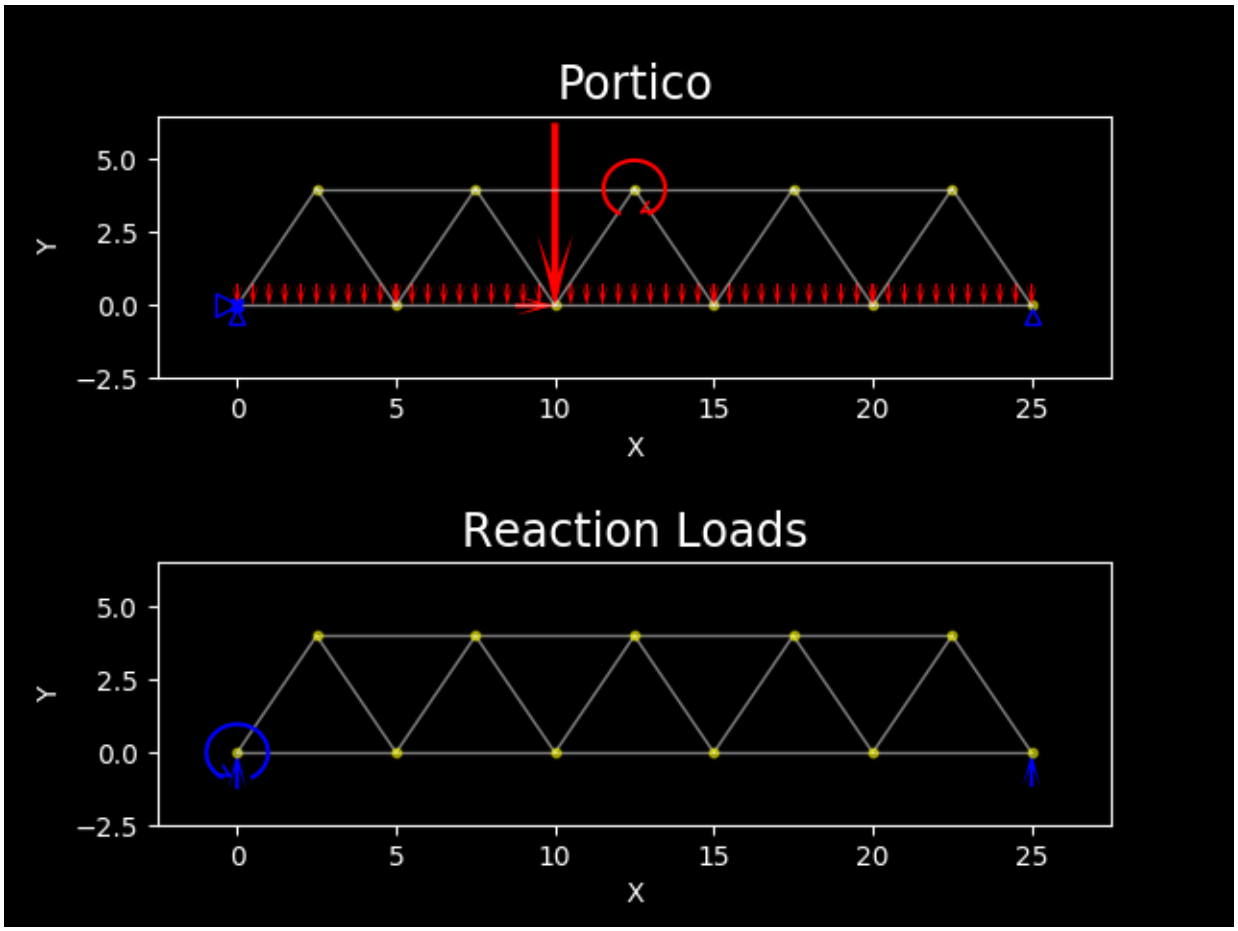


Figura 8: Pórtico e forças de reação.

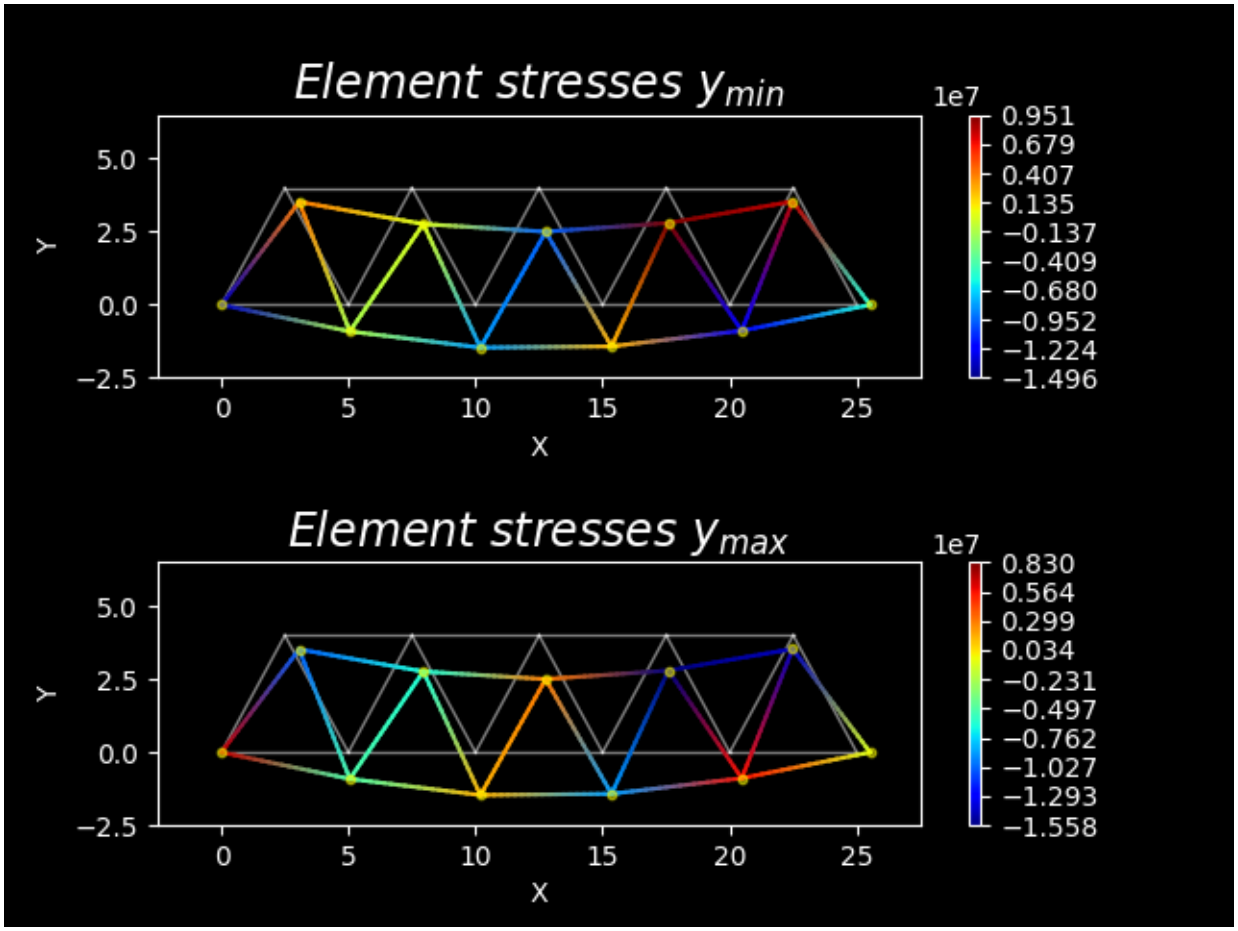


Figura 9: Deformação baseada na deflexão mínima e deflexão máxima.

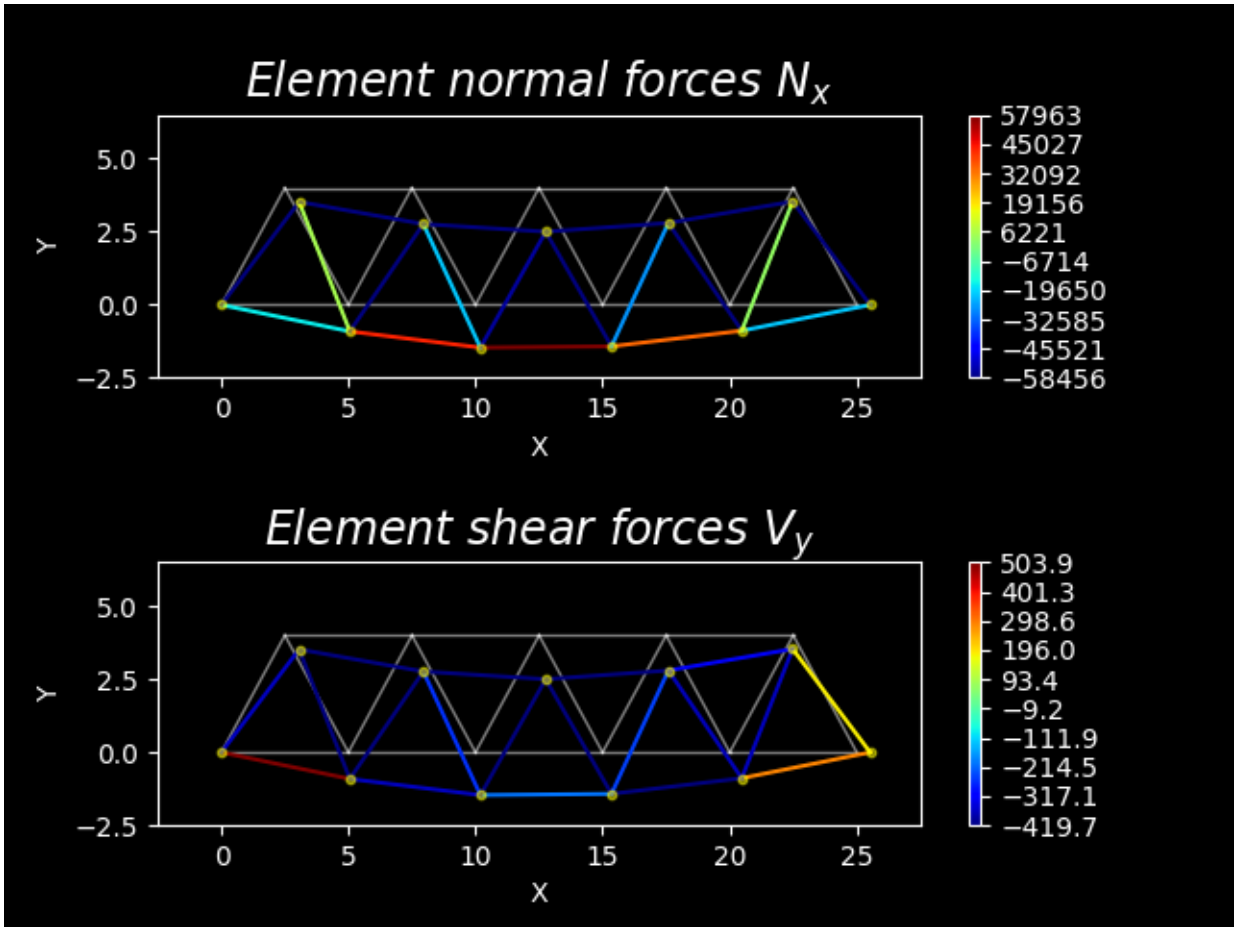


Figura 10: Força normal e força cortante de cada elemento.

# 6 Conclusão

A realização do presente estudo, possibilitou a utilização da linguagem *Python* na análise estrutural de uma treliça e um pórtico. De forma introdutória, apresentou passo a passo os algoritmos necessários para a realização dos cálculos, além dos arquivos de entrada e saída esperados, para assim servir-se como material didático para outras aplicações.

Comparando os programas base em *MATLAB* com os demonstrados em *Python*, observa-se que estes últimos rodaram em menos tempo, como apresentado na tabela 5. Nestes exemplos, a velocidade do programa não foi um fator de grande influência porém, no caso do processamento de alto volume de dados, pode-se concluir que *Python* torna-se uma alternativa melhor.

Tabela 5: Tempo de compilação dos programas em *Python* e *MATLAB*

| Programa | Python | Matlab |
|---|---|---|
| truss2d | 2.956s | 3.415s |
| truss2d_design | 2.940s | 4.085s |
| portico | 4.738s | 7.016s |

Deve-se levar em consideração que uma pequena diferença entre os tempos marcados seja devido ao uso de diferentes ferramentas. Os tempos no *MATLAB* foram cronometrados pela ferramenta de *profiling* disponível no software, já os em *Python* pelo seguinte comando inserido no *bash shell*:

```
$ time python truss2d.py
```

Além disso, apesar de em ambas as linguagens os programas terem a mesma finalidade, analisarem os mesmos arquivos de entrada e gerarem os mesmo gráficos e arquivos de saída, deve-se ressaltar a diferença de número de linhas de cada um dos programas.

Tabela 6: Quantidade de linhas dos programas em *Python* e *MATLAB*

| Programa | Python | Matlab |
|---|---|---|
| truss2d | 411 | 502 |
| truss2d_design | 489 | 593 |
| portico | 615 | 791 |

Outro ponto a ser levantado na comparação entre as duas linguagens é o preço. Enquanto *Python*

é uma linguagem totalmente gratuita e que disponibiliza de várias IDEs, também gratuitas, o software *MATLAB* requer o pagamento de uma licença anual de USD 940.

No desenvolvimento dos códigos, foram utilizadas as bibliotecas *Numpy, Pandas* e *Matplotlib* que são comuns para manipulação e visualização de dados. Porém, outras poderiam ter sido usadas para complementar as funções, a fim de facilitar os cálculos realizados. Por exemplo, a biblioteca *mechpy* que foi desenvolvida justamente para ser um conjunto de funções para Engenheiros Mecânicos e apresenta um módulo de MEF, ainda não apresenta resultados tão completos quanto os propostos neste trabalho. À vista disso, este estudo apresenta-se como um complemento à essa biblioteca.

Na elaboração deste trabalho pôde-se desenvolver as habilidades em *Python*, que atualmente é um conhecimento altamente requisitado em várias áreas, principalmente, em Ciência de Dados, onde é empregado na extração, limpeza e análise de dados.

## 6.1 Trabalhos Futuros

Dado que os recursos em Python, disponíveis na área de Engenharia Mecânica, ainda são escassos, os programas desenvolvidos para este trabalho serão divulgados na plataforma gratuita online *github* <https://github.com/gabrielakoreeda> para assim auxiliar outros estudantes. Além disso, também estará disponível para aceitar contribuições de melhoria.

Baseado nos programas apresentados neste relatório, propõe-se a criação de uma interface gráfica, que também pode ser programada em *Python*, utilizando a biblioteca *Tkinter*. Através do GUI, o usuário poderia inserir os atributos da treliça ou do pórtico manualmente, de forma mais intuitiva. E então o programa realizaria os cálculos e apresentaria os resultados em tabelas mais visuais do que o arquivo de saída apresentado, além de exibir os gráficos gerados.

Outro projeto seria a inclusão desses códigos com os comentários em um *Jupyter Notebook* e dessa forma, poderia ser utilizado como material didático no ensino de MEF, como alternativa ao uso de *MATLAB*. Sendo possivelmente ampliado para se tornar um MOOC mais abrangente ao incluir aplicações em outras áreas como mecânica dos fluídos e transferência de calor.

# 7 Referências Bibliográficas


1. TIOBE Index for April 2018. Disponível em: https://www.tiobe.com/tiobe-index/. Acesso em: 14/04/2018.
2. ABOUT PYTHON. Disponível em: https://www.python.org/about/. Acesso em: 19/11/2018
3. THOMPSON, Maurice J.. Python Programming for Beginners: 1. ed. Ebook, 2018.
4. JACOBS, P.A.. Elements of Python3 Programming for Mechanical Engineers. 2015. Mechanical Engineering Report - The University of Queensland, Queensland, 2015.
5. ALKMIM, Nasser Samir. Implementação Computacional da Solução de Problemas Térmicos e Mecânicos pelo Métodos dos Elementos Finitos em Python. Monografia de Projeto Final - Universidade de Brasília, Brasília, 2016.
6. DUKKIPATI, Rao V.. MATLAB for Mechanical Engineers: 1.ed. Fairfield: New Age Science Limited, 2009.
7. NASSERI, Simin. Solving Mechanical Engineering Problems with MATLAB: 1.ed. New York: Linus Publications, 2015.
8. MOHITE, Dr. P.M.. The Origins of the Finite Element Method.
9. WECK, de Olivier; Kim, Il Yong. Finite Element Method Prototyping. Notas de Aula - Massachusetts Institute of Technology, 2004.
10. CAMPOS, Marco Donisete de. O Metodo de Elementos Finitos Aplicado á Simulação Numérica de Escoamentos de Fluidos. 2006. Universidade Federal de Mato Grosso, 2006.
11. BITTENCOURT, Marco Lúcio. Analise Computacional de Estruturas. Campinas: Editora da Unicamp, 2014.
12. PAVANELLO, Renato. Notas de Aula: Introdução ao Método dos Elementos Finitos. Campinas, 1997.
13. BRENNER, Susanne; SCOTT, Ridgway. The Mathematical Theory of Finite Elements Methods: 3. ed. New York: Springer, 2008.
14. ZIENKIEWICZ, O.C.; Taylor, R.L.. The Finite Element Method, Volume 1: 5. ed. Oxford: Butterworth-Heinemann, 2000.
15. HIBBELER, R.C.. Estática: mecânica para engenharia: 12. ed. São Paulo: Pearson Prentice Hall, 2011.
16. GRADIENT LINES IN MATPLOTLIB. IVANOV. Disponível em: <https://gist.github.com/ivanov/5439438>. Acesso em: 15/04/2018.

# A Arquivo de Entrada truss2d.py e truss2d _design.py

Este documento representa o arquivo de entrada aceito pelos programas *truss2d.py* e *truss2d_design.py*. O primeiro não considera o último item, *DESIGN ITERATIONS*, enquanto que o segundo sim. Sugere-se salvar o conteúdo deste apêndice como um documento de leitura extensão *.fem*, para poder ser lido pelos programas.

*COORDINATES
13
1 0 0
2 500 0
3 1000 0
4 1500 0
5 2000 0
6 2500 0
7 3000 0
8 250 500
9 750 500
10 1250 500
11 1750 500
12 2250 500
13 2750 500

*ELEMENT _GROUPS
1
1 23

*INCIDENCES
1 1 2
2 2 3
3 3 4
4 4 5
5 5 6
6 6 7
7 1 8
8 8 2

9 2 9
10 9 3
11 3 10
12 10 4
13 4 11
14 11 5
15 5 12
16 12 6
17 6 13
18 7 13
19 8 9
20 9 10
21 10 11
22 11 12
23 12 13

*MATERIALS
1
2100000 120 80

*GEOMETRIC _PROPERTIES
1
314.15

*BCNODES
4
1 1
1 2
7 1
7 2

*LOADS
6
8 2 -20000
9 2 -20000
10 2 -20000

11 2 -20000

12 2 -20000

13 2 -20000

*DESIGN_ITERATIONS

5

# B truss2d.py

```
# Importing libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.pylab as pl
from matplotlib.colors import LinearSegmentedColormap
from matplotlib import cm

# Reading the input filename
br = 60 * '='
print(br + '\n' + 'TRUSS 2D' + '\n' + br + '\n' + '\n' + br + '\n' + br)
inpFileName = 'example1_truss'
fileName = inpFileName + '.fem'
with open(fileName, 'r') as myFile:
    file = myFile.read().splitlines()
myFile.close()


# Reading function
def readnum(keyword, document, type_num='int'):
    if keyword in document:
        totalnum = int(document[document.index(keyword) + 1])
        if totalnum > 0:
            num = pd.DataFrame(list(map(lambda x: x.split(),
                                        document[document.index(keyword) + 2:
                                                 document.index(keyword) + 2 + totalnum])
                                    )
                               ).astype(type_num).as_matrix()
        else:
            num = pd.DataFrame()
        return totalnum, num
    else:
        raise ValueError("{} Not Found.".format(keyword))
```

```
def removefirstcol(df):
    return df[:, 1:]


# Reading nodal coordinates
print('READING NODAL COORDINATES...')
totalNumNodes, coords = readnum('*COORDINATES', file, type_num='float')
coords = removefirstcol(coords)

# Number of nodal dofs
NumNodalDOFs = 2

# Reading element groups
print('READING ELEMENT GROUPS...')
numGroups, groups = readnum('*ELEMENT_GROUPS', file)
groups = removefirstcol(groups)

# Total number of elements
totalNumElements = groups.sum()

# Reading incidences
print('READING ELEMENT INCIDENCES...')
if '*INCIDENCES' in file:
    incid = pd.DataFrame(list(map(lambda groups: groups.split(),
                                  file[file.index('*INCIDENCES') + 1:
                                       file.index('*INCIDENCES') + totalNumElements + 1])
                              )).astype('int').as_matrix()
else:
    raise ValueError('{} Not Found'.format('*INCIDENCES'))

# deletes element numbers
incid = removefirstcol(incid)

# Reading Materials
print('READING MATERIALS...')
```

```
numMaters, maters = readnum('*MATERIALS', file, type_num='float')

# Reading geometric properties
print('READING GEOMETRIC PROPERTIES...')
numGPs, gps = readnum('*GEOMETRIC_PROPERTIES', file, type_num='float')

# Reading boundary conditions
print('READING BOUNDARY CONDITIONS...')
numBCNodes, hdbcNodes = readnum('*BCNODES', file)

# Reading loads
print('READING LOADS...')
numLoadedNodes, loads = readnum('*LOADS', file)
print(br)

# Element orientation: length, lx = sin theta, ly = cos theta
print('SOLUTION...\n' + br + '\n' + br)

# element lengths
lengths = np.sqrt(np.absolute((coords[incid[:, 0] - 1, 0] - coords[incid[:, 1] - 1, 0])**2 +
                              (coords[incid[:, 0] - 1, 1] - coords[incid[:, 1] - 1, 1])**2))

# lx = sin
lx = (coords[incid[:, 1] - 1, 1] - coords[incid[:, 0] - 1, 1]) / lengths

# ly = cos
ly = (coords[incid[:, 1] - 1, 0] - coords[incid[:, 0] - 1, 0]) / lengths

prop = np.matrix([lengths, lx, ly]).transpose()

# Solver
# DOF Numbering
# DOFs matrix initialized with ones
nodalDOFNumbers = np.ones((numNodalDOFs, totalNumNodes), dtype=int)

# Boundary conditions on dofs with zero values
```

```
nodalDOFNumbers[hdbcNodes[:numBCNodes, 1] - 1, hdbcNodes[:numBCNodes, 0] - 1] = 0

# Total number of DOFs for the model
totalNumDOFs = totalNumNodes*numNodalDOFs

# Number of free and restricted dofs
totalNumFreeDOFs = np.sum(nodalDOFNumbers == 1)
totalNumRestrDOFs = np.sum(nodalDOFNumbers == 0)

# DOFs numbering
ukeqnum = 0
keqnum = totalNumFreeDOFs
for i in range(totalNumNodes):
    for j in range(numNodalDOFs):
        if nodalDOFNumbers[j, i] == 1:
            ukeqnum = ukeqnum + 1
            nodalDOFNumbers[j, i] = ukeqnum
        elif nodalDOFNumbers[j, i] == 0:
            keqnum = keqnum + 1
            nodalDOFNumbers[j, i] = keqnum

# Assembling of the global stiffness matrix and load vector
# Creates zero global matrix
kg = np.zeros((totalNumDOFs, totalNumDOFs))
elemNum = 0
for grp in range(numGroups):
    e = maters[grp, 0]
    a = gps[grp, 0]
    for tne in range(groups[grp - 1, 0]):
        cc = prop[elemNum, 2]**2
        ss = prop[elemNum, 1]**2
        cs = prop[elemNum, 1]*prop[elemNum, 2]

        # Element stiffness matrix
        kedf = [[cc, cs, -cc, -cs],
                [cs, ss, -cs, -ss],
```

```
                [-cc, -cs, cc, cs],
                [-cs, -ss, cs, ss]]
        ke = np.dot((e * a / prop[elemNum, 0]), kedf)

        # Element DOFs
        elemEqs = nodalDOFNumbers[:, incid[elemNum, :] - 1]

        # Assembling into the global matrix
        index = np.concatenate(np.column_stack(elemEqs)) - 1
        kg[index, index.reshape((-1, 1))] = kg[index, index.reshape((-1, 1))] + ke
        elemNum = elemNum + 1

# Assembling global matrix
fg = np.zeros((totalNumDOFs, 1))
for i in range(numLoadedNodes):
    # element dofs
    elemEqs = nodalDOFNumbers[loads[i, 1] - 1, loads[i, 0] - 1]

    # assembling into the global vector
    fg[elemEqs - 1, 0] = loads[i, 2]

# Solving systems of equations
u = np.zeros((totalNumDOFs, 1))
u[:totalNumFreeDOFs, 0] = np.dot(np.linalg.inv(kg[:totalNumFreeDOFs, :totalNumFreeDOFs]),
                                 fg[:totalNumFreeDOFs, 0])

# Calculates reaction forces, element strains and stresses
# reaction forces
rf = np.dot(kg[totalNumFreeDOFs:, :totalNumFreeDOFs], u[:totalNumFreeDOFs])

# Deformed coordinates
coordsd = coords + u[np.concatenate(np.row_stack(nodalDOFNumbers)) - 1].reshape(
    (nodalDOFNumbers.shape[0], nodalDOFNumbers.shape[1])).transpose()
scaleFactor = 100
coordsdd = coords + scaleFactor*u[np.concatenate(np.row_stack(nodalDOFNumbers)) - 1].reshape(
    (nodalDOFNumbers.shape[0], nodalDOFNumbers.shape[1])).transpose()
```

```
# Element strain and stress vectors
strain = np.zeros((totalNumElements, 1))
stress = np.zeros((totalNumElements, 1))

# Calculates strain and stress for the elements
elemNum = 0
for grp in range(numGroups):
    e = maters[grp, 0]
    for elNum in range(groups[grp - 1, 0]):
        # element dofs and displacements
        elemEqs = nodalDOFNumbers[:, incid[elemNum, :] - 1]
        index = np.concatenate(np.column_stack(elemEqs)) - 1
        elemU = u[index, 0]
        # strain
        strain[elemNum] = 1 / prop[elemNum, 0]*np.dot(
            np.array([[-prop[elemNum, 2], -prop[elemNum, 1],
                       prop[elemNum, 2], prop[elemNum, 1]]]), elemU)

        # stress
        stress[elemNum] = strain[elemNum]*e

        elemNum = elemNum + 1

# Post-Processing
print('POST-PROCESSING... \n' + br)

# Original Mesh Function
def originalMesh(ax, num_elements=totalNumElements):
    for elemNum in range(num_elements):
        ax.plot(coords[incid[elemNum, :] - 1, 0], coords[incid[elemNum, :] - 1, 1],
                color='white', linewidth=1, alpha=0.5)


# Nodes of the Underformed Mesh Function
def nodes_mesh(ax, coords=coords, order=1):
```

```
    ax.scatter(x=coords[:, 0], y=coords[:, 1], c='yellow',
               s=10, alpha=0.5, zorder=order)


# Color gradient function
def plot_gradient_hack(p0, p1, npts=20, cmap=None, **kw):
    """
    Draw a gradient between p0 and p1 using a colormap
    The **kw dictionary gets passed to plt.plot, so things like linestyle,
    linewidth, labels, etc can be modified directly.
    """
    x_1, y_1 = p0
    x_2, y_2 = p1

    X = np.linspace(x_1, x_2, npts)
    Xs = X[:-1]
    Xf = X[1:]
    Xpairs = zip(Xs, Xf)

    Y = np.linspace(y_1, y_2, npts)
    Ys = Y[:-1]
    Yf = Y[1:]
    Ypairs = zip(Ys, Yf)

    C = np.linspace(0, 1, npts)
    cmap = plt.get_cmap(cmap)
    # the simplest way of doing this is to just do the following:
    for x, y, c in zip(Xpairs, Ypairs, C):
        plt.plot(x, y, '-', c=cmap(c), **kw)


def plot_gradient_rbg_pairs(p0, p1, rgb0, rgb1, **kw):
    """Form the gradient from RGB values at each point
    The **kw dictionary gets passed to plt.plot, so things like linestyle,
    linewidth, labels, etc can be modified directly.
    """
```

```
    cmap = LinearSegmentedColormap.from_list('jet', (rgb0, rgb1))
    plot_gradient_hack(p0, p1, cmap=cmap, **kw)


# Color definition Function
def rankmin(x):
    x = np.round(x, decimals=4)
    u, inv, counts = np.unique(x, return_inverse=True, return_counts=True)
    csum = np.zeros_like(counts)
    csum[1:] = counts[:-1].cumsum()
    return csum[inv]

# Mesh and boundary conditions
plt.style.use('dark_background')
fig1 = plt.figure()
plt.subplots_adjust(hspace=0.7, wspace=0.7)

ax1 = fig1.add_subplot(211)
ax1.set_title('Truss', fontsize=17)
plt.xlabel('X', fontsize=10)
plt.ylabel('Y', fontsize=10)
plt.xlim(min(coords[:, 0]) - 0.5*prop[0, 0], max(coords[:, 0]) + 0.5*prop[0, 0])
plt.ylim(min(coords[:, 1]) - 0.5*prop[0, 0], max(coords[:, 1]) + 0.5*prop[0, 0])

# Plots Original Mesh
originalMesh(ax1)

# Plots Nodes of the Undeformed Mesh
nodes_mesh(ax1)

# Plot loads
fFe = 0.25 * loads[:, 2] * np.asscalar(abs(max(prop[:, 0]))/abs(max(loads[:, 2])))
for i in range(len(loads)):
    if loads[i, 1] == 1:
        ax1.quiver(coords[loads[i, 0] - 1, 0] - fFe[i], coords[loads[i, 0] - 1, 1],
                   fFe[i], 0, color='red', scale=1, units='xy', scale_units='xy',
```

```
                    headlength=10, headwidth=5)
    elif loads[i, 1] == 2:
        ax1.quiver(coords[loads[i, 0] - 1, 0], coords[loads[i, 0] - 1, 1] - fFe[i],
                    0, fFe[i], color='red', scale=1, units='xy', scale_units='xy',
                    headlength=10, headwidth=5)

# Plot bcs
apoio = np.asscalar(0.7e-1*abs(max(prop[:, 0])))
for i in range(len(hdbcNodes)):
    if hdbcNodes[i, 1] == 1:
        ax1.plot(coords[hdbcNodes[i, 0] - 1, 0] - 0.5*apoio,
                 coords[hdbcNodes[i, 0] - 1, 1], marker='>', color='black',
                 markersize=8, markeredgecolor='blue')
    elif hdbcNodes[i, 1] == 2:
        ax1.plot(coords[hdbcNodes[i, 0] - 1, 0],
                 coords[hdbcNodes[i, 0] - 1, 1] - apoio, marker='^', color='black',
                 markersize=8, markeredgecolor='blue')

# Plot reaction forces
ax2 = fig1.add_subplot(212)
ax2.set_title('Reaction Loads', fontsize = 17)
plt.xlabel('X', fontsize = 10)
plt.ylabel('Y', fontsize = 10)
plt.xlim(min(coords[:, 0]) - 0.5*prop[0, 0], max(coords[:, 0]) + 0.5*prop[0, 0])
plt.ylim(min(coords[:, 1]) - 0.5*prop[0, 0], max(coords[:, 1]) + 0.5*prop[0, 0])
originalMesh(ax2)

# plot reaction loads
fFR = 0.25 * (rf * abs(max(prop[:, 0]))) / abs(max(rf))
for i in range(len(hdbcNodes)):
    ffr = np.asscalar(fFR[i])
    if hdbcNodes[i, 1] == 1:
        ax2.quiver(coords[hdbcNodes[i, 0] - 1, 0] - fFR[i, 0],
                    coords[hdbcNodes[i, 0] - 1, 1], fFR[i, 0], 0,
                    color='blue', scale=1, units='xy',
                    scale_units='xy', headlength=10, headwidth=5)
```

```
    elif hdbcNodes[i, 1] == 2:
        ax2.quiver(coords[hdbcNodes[i, 0] - 1, 0],
                   coords[hdbcNodes[i, 0] - 1, 1] - fFR[i, 0],
                   0, fFR[i, 0], color='blue', scale=1, units='xy',
                   scale_units='xy', headlength=10, headwidth=5)

nodes_mesh(ax2, order=totalNumElements + 1)
fig1.show()

### Plot Element Strains
fig2 = plt.figure()
plt.subplots_adjust(hspace=0.7, wspace=0.7)
ax3 = fig2.add_subplot(211)
ax3.set_title('Element Strains', fontsize=17)
plt.xlabel('X', fontsize=10)
plt.ylabel('Y', fontsize=10)
plt.xlim(min(coords[:, 0]) - 0.5*prop[0, 0], max(coords[:, 0]) + 0.5*prop[0, 0])
plt.ylim(min(coords[:, 1]) - 0.5*prop[0, 0], max(coords[:, 1]) + 0.5*prop[0, 0])
originalMesh(ax3)

# Plot deformed mesh
# plot deformed mesh and element strains
d_ef = np.abs(coords - coordsd)
d_ef = np.sum(d_ef, axis=1)
z = rankmin(d_ef)
colors = pl.cm.jet(np.linspace(0, 1, len(set(z))))
Z = np.unique(z, return_index=False)
for elemNum in range(totalNumElements):
    xy = coordsdd[incid[elemNum, :] - 1, :]
    plot_gradient_rbg_pairs(xy[0, :], xy[1, :],
                            colors[np.where(z[incid[elemNum, 0] - 1] == Z)[0][0]],
                            colors[np.where(z[incid[elemNum, 1] - 1] == Z)[0][0]])

sm = plt.cm.ScalarMappable(cmap=plt.cm.jet,
                           norm=plt.Normalize(vmin=d_ef.min(), vmax=d_ef.max()))
sm._A = []
```

```
plt.colorbar(sm, ax=ax3, ticks=np.linspace(d_ef.min(), d_ef.max(), num=10))

# plot nodes of deformed mesh
nodes_mesh(ax3, coordsdd, order=totalNumElements + 1)

# Plot Element Stresses
ax4 = fig2.add_subplot(212)
ax4.set_title('Element stresses', fontsize=17)
plt.xlabel('X', fontsize=10)
plt.ylabel('Y', fontsize=10)
plt.xlim(min(coords[:, 0]) - 0.5*prop[0, 0], max(coords[:, 0]) + 0.5*prop[0, 0])
plt.ylim(min(coords[:, 1]) - 0.5*prop[0, 0], max(coords[:, 1]) + 0.5*prop[0, 0])
originalMesh(ax4)

# plot stresses on elements of the deformed mesh
z2 = rankmin(stress)
colors2 = pl.cm.jet(np.linspace(0, 1, len(set(z2))))
Z2 = np.unique(z2, return_index=False)
for elemNum in range(totalNumElements):
        ax4.plot(coordsdd[incid[elemNum, :] - 1, 0],
                 coordsdd[incid[elemNum, :] - 1, 1],
                 linewidth=1, alpha=1, c=colors2[np.where(z2[elemNum] == Z2)[0][0]])

sm = plt.cm.ScalarMappable(cmap=plt.cm.jet,
                           norm=plt.Normalize(vmin=stress.min(), vmax=stress.max()))
sm._A = []
plt.colorbar(sm, ax=ax4, ticks=np.linspace(stress.min(), stress.max(), num=10))

# plot nodes of deformed mesh
nodes_mesh(ax4, coordsdd, order=totalNumElements + 1)
plt.show()

# Output File
outFileName = inpFileName + '.out2'
with open(outFileName, 'w') as f:
    f.write('*DISPLACEMENTS\n')
```

```
for i in range(totalNumNodes):
    f.write('{:d} {:.4f} {:.4f}\n'.format(i + 1, u[nodalDOFNumbers[0, i] - 1, 0],
                                          u[nodalDOFNumbers[1, i] - 1, 0]))
f.write('\n*ELEMENT_STRAINS\n')
for i in range(totalNumElements):
    f.write('{:d} {:.6e}\n'.format(i + 1, strain[i, 0]))
f.write('\n*ELEMENT_STRESSES\n')
for i in range(totalNumElements):
    f.write('{:d} {:.6e}\n'.format(i + 1, stress[i, 0]))
f.write('\n*REACTION_FORCES\n')
for i in range(totalNumRestrDOFs):
    if hdbcNodes[i, 1] == 1:
        f.write('{:d} FX = {:.6e}\n'.format(hdbcNodes[i, 0], rf[i, 0]))
    else:
        f.write('{:d} FY = {:.6e}\n'.format(hdbcNodes[i, 0], rf[i, 0]))
f.close()
```

# C Arquivo de Saída truss2d.py

*DISPLACEMENTS
1 0.0000 0.0000
2 -0.0253 -0.2512
3 -0.0202 -0.4322
4 0.0000 -0.4976
5 0.0202 -0.4322
6 0.0253 -0.2512
7 0.0000 0.0000
8 0.1326 -0.1299
9 0.0947 -0.3536
10 0.0341 -0.4806
11 -0.0341 -0.4806
12 -0.0947 -0.3536
13 -0.1326 -0.1299

*ELEMENT _STRAINS
1 -5.052687e-05
2 1.010537e-05
3 4.042150e-05
4 4.042150e-05
5 1.010537e-05
6 -5.052687e-05
7 -1.016834e-04
8 6.778891e-05
9 -6.778891e-05
10 3.389445e-05
11 -3.389445e-05
12 9.930137e-20
13 1.986027e-19
14 -3.389445e-05
15 3.389445e-05
16 -6.778891e-05
17 6.778891e-05
18 -1.016834e-04

19 -7.579030e-05

20 -1.212645e-04

21 -1.364225e-04

22 -1.212645e-04

23 -7.579030e-05

*ELEMENT _STRESS

1 -1.061064e+02

2 2.122128e+01

3 8.488514e+01

4 8.488514e+01

5 2.122128e+01

6 -1.061064e+02

7 -2.135351e+02

8 1.423567e+02

9 -1.423567e+02

10 7.117835e+01

11 -7.117835e+01

12 2.085329e-13

13 4.170657e-13

14 -7.117835e+01

15 7.117835e+01

16 -1.423567e+02

17 1.423567e+02

18 -2.135351e+02

19 -1.591596e+02

20 -2.546554e+02

21 -2.864873e+02

22 -2.546554e+02

23 -1.591596e+02

*REACTION _FORCES

1 FX = 6.333333e+04

1 FY = 6.000000e+04

7 FX = -6.333333e+04

7 FY = 6.000000e+04

# D truss2d_design.py

```
# Importing libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.pylab as pl
from matplotlib.colors import LinearSegmentedColormap
from matplotlib import cm

# Reading the input filename
br = 60 * '='
print(br + '\n' + 'TRUSS 2D' + '\n' + br + '\n' + '\n' + br + '\n' + br)
inpFileName = 'example2_truss'
fileName = inpFileName + '.fem'
with open(fileName, 'r') as myFile:
    file = myFile.read().splitlines()
myFile.close()


# Reading function
def readnum(keyword, document, type_num='int'):
    if keyword in document:
        totalnum = int(document[document.index(keyword) + 1])
        if totalnum > 0:
            num = pd.DataFrame(list(map(lambda x: x.split(),
                                        document[document.index(keyword) + 2:
                                                 document.index(keyword) + 2 + totalnum])
                                    )
                               ).astype(type_num).as_matrix()
        else:
            num = pd.DataFrame()
        return totalnum, num
    else:
        raise ValueError("{} Not Found.".format(keyword))
```

```
def removefirstcol(df):
    return df[:, 1:]


# Reading nodal coordinates
print('READING NODAL COORDINATES...')
totalNumNodes, coords = readnum('*COORDINATES', file, type_num='float')
coords = removefirstcol(coords)

# Number of nodal dofs
numNodalDOFs = 2

# Reading element groups
print('READING ELEMENT GROUPS...')
numGroups, groups = readnum('*ELEMENT_GROUPS', file)
groups = removefirstcol(groups)

# Total number of elements
totalNumElements = groups.sum()

# Reading incidences
print('READING ELEMENT INCIDENCES...')
if '*INCIDENCES' in file:
    incid = pd.DataFrame(list(map(lambda groups: groups.split(),
                                  file[file.index('*INCIDENCES') + 1:
                                       file.index('*INCIDENCES') + totalNumElements + 1])
                              )
                         ).astype('int').as_matrix()
else:
    raise ValueError('{} Not Found'.format('*INCIDENCES'))

# deletes element numbers
incid = removefirstcol(incid)

# Reading Materials
```

```
print('READING MATERIALS...')
numMaters, maters = readnum('*MATERIALS', file, type_num='float')

# Reading geometric properties
print('READING GEOMETRIC PROPERTIES...')
numGPs, gps = readnum('*GEOMETRIC_PROPERTIES', file, type_num='float')

# Reading boundary conditions
print('READING BOUNDARY CONDITIONS...')
numBCNodes, hdbcNodes = readnum('*BCNODES', file)

# Reading loads
print('READING LOADS...')
numLoadedNodes, loads = readnum('*LOADS', file)
print(br)

# Reading maximum design iterations
print('READING DESIGN ITERATIONS...')
if '*DESIGN_ITERATIONS' in file:
    maxNumDesignIter = int(file[file.index('*DESIGN_ITERATIONS') + 1])
else:
    raise ValueError('{} Not Found.'.format('*DESIGN_ITERATIONS'))

# Element orientation: length, lx = sin theta, ly = cos theta
print('SOLUTION...\n' + br + '\n' + br)

# element lengths
lengths = np.sqrt(np.absolute((coords[incid[:, 0] - 1, 0] -
                               coords[incid[:, 1] - 1, 0])**2 +
                              (coords[incid[:, 0] - 1, 1] -
                               coords[incid[:, 1] - 1, 1])**2))

# lx = sin
lx = (coords[incid[:, 1] - 1, 1] - coords[incid[:, 0] - 1, 1]) / lengths

# ly = cos
```

```
ly = (coords[incid[:, 1] - 1, 0] - coords[incid[:, 0] - 1, 0]) / lengths

prop = np.matrix([lengths, lx, ly]).transpose()

# Solver
# DOF Numbering
# DOFs matrix initialized with ones
nodalDOFNumbers = np.ones((numNodalDOFs, totalNumNodes), dtype=int)

# Boundary conditions on dofs with zero values
nodalDOFNumbers[hdbcNodes[:numBCNodes, 1] - 1, hdbcNodes[:numBCNodes, 0] - 1] = 0

# Total number of DOFs for the model
totalNumDOFs = totalNumNodes*numNodalDOFs

# Number of free and restricted dofs
totalNumFreeDOFs = np.sum(nodalDOFNumbers == 1)
totalNumRestrDOFs = np.sum(nodalDOFNumbers == 0)

# DOFs numbering
ukeqnum = 0
keqnum = totalNumFreeDOFs
for i in range(totalNumNodes):
    for j in range(numNodalDOFs):
        if nodalDOFNumbers[j, i] == 1:
            ukeqnum = ukeqnum + 1
            nodalDOFNumbers[j, i] = ukeqnum
        elif nodalDOFNumbers[j, i] == 0:
            keqnum = keqnum + 1
            nodalDOFNumbers[j, i] = keqnum

# Assembling of the global stiffness matrix and load vector
# Areas and Materials Properties for each element
areas = np.zeros((totalNumElements, maxNumDesignIter))
matProp = np.zeros((totalNumElements, 3))
elemNum = 0
```

```
for grp in range(numGroups):
    for el in range(int(groups[grp])):
        areas[elemNum, 0] = gps[grp, 0]
        matProp[elemNum, :] = maters[grp, :3]
        elemNum = elemNum + 1

# Element Strain and Stress Vectors
strain = np.zeros((totalNumElements, maxNumDesignIter))
stress = np.zeros((totalNumElements, maxNumDesignIter))

# Assembling global load vector
fg = np.zeros((totalNumDOFs, 1))
for i in range(numLoadedNodes):
    # element dofs
    elemEqs = nodalDOFNumbers[loads[i, 1] - 1, loads[i, 0] - 1]

    # assembling into the global vector
    fg[elemEqs - 1, 0] = loads[i, 2]

# Global displacement vector
u = np.zeros((totalNumDOFs, 1))

# Sets number of design iterations and design flag (=1, new design; 0= final design)
nDIt = 1
designFlag = 1

while designFlag:
    # Creates zero global matrix and load vector
    kg = np.zeros((totalNumDOFs, totalNumDOFs))
    for elemNum in range(totalNumElements):
        # Youngs modulus and cross section area
        e = matProp[elemNum, 0]
        a = areas[elemNum, nDIt - 1]

        # director co-sine
        cc = prop[elemNum, 2]**2
```

```
        ss = prop[elemNum, 1]**2
        cs = prop[elemNum, 1]*prop[elemNum, 2]

        # element stiffness matrix
        kedf = [[cc, cs, -cc, -cs],
                [cs, ss, -cs, -ss],
                [-cc, -cs, cc, cs],
                [-cs, -ss, cs, ss]]
        ke = np.dot((e * a / prop[elemNum, 0]), kedf)

        # element dofs
        elemEqs = nodalDOFNumbers[:, incid[elemNum, :] - 1]

        # assembling into the global matrix
        index = np.concatenate(np.column_stack(elemEqs)) - 1
        kg[index, index.reshape((-1, 1))] = kg[index, index.reshape((-1, 1))] + ke

    # Solving systems of equations
    u[:totalNumFreeDOFs, 0] = np.dot(np.linalg.inv(kg[:totalNumFreeDOFs,
                                                      :totalNumFreeDOFs]),
                                     fg[:totalNumFreeDOFs, 0])

    # Calculates strain and stress or the elements
    for elemNum in range(totalNumElements):

        # Youngs modulus
        e = matProp[elemNum, 0]

        # Elements DOFs and displacements
        elemEqs = nodalDOFNumbers[:, incid[elemNum, :] - 1]
        index = np.concatenate(np.column_stack(elemEqs)) - 1
        elemU = u[index, 0]

        # Strain
        strain[elemNum, nDIt - 1] = 1 / prop[elemNum, 0]*np.dot(np.array([[-prop[elemNum, 2],
                                                                           -prop[elemNum, 1],
```

```
                                                                    prop[elemNum, 2],
                                                                    prop[elemNum, 1]]]),
                                                            elemU)

    # Stress
    stress[elemNum, nDIt - 1] = strain[elemNum, nDIt - 1]*e

# if the maximum number of design iterations is reached, finish the design process
# checks if the stresses for all bars are below the admissable limits or
# if the maximum number of iterations has been reached
tractionBars = stress[:, nDIt - 1] >= 0
compressionBars = stress[:, nDIt - 1] < 0
if (sum(stress[tractionBars, 0] <= matProp[tractionBars, 1]) +
    sum(stress[compressionBars, 0] >= -matProp[compressionBars, 2]) == totalNumElements) \
        or (nDIt + 1 >= maxNumDesignIter):
        designFlag = 0
# otherwise calculate the new areas
else:
    # increments number of design iterations
    nDIt += 1

    # copies areas from the previous iteration
    areas[:, nDIt - 1] = areas[:, nDIt - 2]

    # bars under compression with stress less than admissable
    # compression stress. They will be redesign.
    newBars = stress[:, nDIt - 2] < - matProp[:, 2]

    # new areas for bars under compression
    areas[newBars, nDIt - 1] = (- stress[newBars, nDIt - 2] /
                                matProp[newBars, 2] *
                                areas[newBars, nDIt - 2])

    # bars under traction with stress larger than admissable
    # tensile stress. They will be redesigned.
    newBars = stress[:, nDIt - 2] > matProp[:, 1]
```

```
        # new areas for bars under traction
        areas[newBars, nDIt - 1] = (stress[newBars, nDIt - 2] /
                                    matProp[newBars, 1] *
                                    areas[newBars, nDIt - 2])

# reaction forces
rf = np.dot(kg[totalNumFreeDOFs:, :totalNumFreeDOFs], u[:totalNumFreeDOFs])

# Deformed coordinates
coordsd = coords + u[np.concatenate(np.row_stack(nodalDOFNumbers)) - 1].reshape(
    (nodalDOFNumbers.shape[0], nodalDOFNumbers.shape[1])).transpose()
scaleFactor = 100
coordsdd = coords + scaleFactor*u[np.concatenate(np.row_stack(nodalDOFNumbers)) - 1].reshape(
    (nodalDOFNumbers.shape[0], nodalDOFNumbers.shape[1])).transpose()

# Post-Processing
print('POST-PROCESSING... \n' + br)


# Original Mesh Function
def originalMesh(ax, num_elements=totalNumElements):
    for elemNum in range(num_elements):
        ax.plot(coords[incid[elemNum, :] - 1, 0], coords[incid[elemNum, :] - 1, 1],
                color='white', linewidth=1, alpha=0.5)


# Nodes of the Underformed Mesh Function
def nodes_mesh(ax, coords=coords, order=1):
    ax.scatter(x=coords[:, 0], y=coords[:, 1], c='yellow',
               s=10, alpha=0.5, zorder=order)


# Color gradient function
def plot_gradient_hack(p0, p1, npts=20, cmap=None, **kw):
    """
```

```
    Draw a gradient between p0 and p1 using a colormap
    The **kw dictionary gets passed to plt.plot, so things like linestyle,
    linewidth, labels, etc can be modified directly.
    """
    x_1, y_1 = p0
    x_2, y_2 = p1

    X = np.linspace(x_1, x_2, npts)
    Xs = X[:-1]
    Xf = X[1:]
    Xpairs = zip(Xs, Xf)

    Y = np.linspace(y_1, y_2, npts)
    Ys = Y[:-1]
    Yf = Y[1:]
    Ypairs = zip(Ys, Yf)

    C = np.linspace(0, 1, npts)
    cmap = plt.get_cmap(cmap)
    # the simplest way of doing this is to just do the following:
    for x, y, c in zip(Xpairs, Ypairs, C):
        plt.plot(x, y, '-', c=cmap(c), **kw)


def plot_gradient_rbg_pairs(p0, p1, rgb0, rgb1, **kw):
    """Form the gradient from RGB values at each point
    The **kw dictionary gets passed to plt.plot, so things like linestyle,
    linewidth, labels, etc can be modified directly.
    """
    cmap = LinearSegmentedColormap.from_list('jet', (rgb0, rgb1))
    plot_gradient_hack(p0, p1, cmap=cmap, **kw)


# Color definition Function
def rankmin(x):
    x = np.round(x, decimals=4)
```

```
    u, inv, counts = np.unique(x, return_inverse=True, return_counts=True)
    csum = np.zeros_like(counts)
    csum[1:] = counts[:-1].cumsum()
    return csum[inv]


# Mesh and boundary conditions
plt.style.use('dark_background')
fig1 = plt.figure()
plt.subplots_adjust(hspace=0.7, wspace=0.7)

ax1 = fig1.add_subplot(211)
ax1.set_title('Truss', fontsize=17)
plt.xlabel('X', fontsize=10)
plt.ylabel('Y', fontsize=10)
plt.xlim(min(coords[:, 0]) - 0.5*prop[0, 0], max(coords[:, 0]) + 0.5*prop[0, 0])
plt.ylim(min(coords[:, 1]) - 0.5*prop[0, 0], max(coords[:, 1]) + 0.5*prop[0, 0])

# Plots Original Mesh
originalMesh(ax1)

# Plots Nodes of the Underformed Mesh
nodes_mesh(ax1)

# Plot Loads
fFe = 0.25 * loads[:, 2] * np.asscalar(abs(max(prop[:, 0]))/abs(max(loads[:, 2])))
for i in range(len(loads)):
    if loads[i, 1] == 1:
        ax1.quiver(coords[loads[i, 0] - 1, 0] - fFe[i], coords[loads[i, 0] - 1, 1],
                   fFe[i], 0, color='red', scale=1, units='xy', scale_units='xy',
                   headlength=10, headwidth=5)
    elif loads[i, 1] == 2:
        ax1.quiver(coords[loads[i, 0] - 1, 0], coords[loads[i, 0] - 1, 1] - fFe[i],
                   0, fFe[i], color='red', scale=1, units='xy', scale_units='xy',
                   headlength=10, headwidth=5)
```

```
# Plot BCs
apoio = np.asscalar(0.7e-1*abs(max(prop[:, 0])))
for i in range(len(hdbcNodes)):
    if hdbcNodes[i, 1] == 1:
        ax1.plot(coords[hdbcNodes[i, 0] - 1, 0] - apoio,
                 coords[hdbcNodes[i, 0] - 1, 1], marker='>', color='black',
                 markersize=8, markeredgecolor='blue')
    elif hdbcNodes[i, 1] == 2:
        ax1.plot(coords[hdbcNodes[i, 0] - 1, 0],
                 coords[hdbcNodes[i, 0] - 1, 1] - apoio,
                 marker='^', color='black', markersize=8, markeredgecolor='blue')

# Plot reaction forces
ax2 = fig1.add_subplot(212)
ax2.set_title('Reaction Loads', fontsize=17)
plt.xlabel('X', fontsize=10)
plt.ylabel('Y', fontsize=10)
plt.xlim(min(coords[:, 0]) - 0.5*prop[0, 0], max(coords[:, 0]) + 0.5*prop[0, 0])
plt.ylim(min(coords[:, 1]) - 0.5*prop[0, 0], max(coords[:, 1]) + 0.5*prop[0, 0])
originalMesh(ax2)

# plot reaction loads
fFR = 0.25 * (rf * abs(max(prop[:, 0]))) / abs(max(rf))
for i in range(len(hdbcNodes)):
    if hdbcNodes[i, 1] == 1:
        ax2.quiver(coords[hdbcNodes[i, 0] - 1, 0] - fFR[i, 0],
                   coords[hdbcNodes[i, 0] - 1, 1], fFR[i, 0], 0,
                   color='blue', scale=1, units='xy',
                   scale_units='xy', headlength=10, headwidth=5)
    elif hdbcNodes[i, 1] == 2:
        ax2.quiver(coords[hdbcNodes[i, 0] - 1, 0],
                   coords[hdbcNodes[i, 0] - 1, 1] - fFR[i, 0], 0,
                   fFR[i, 0], color='blue', scale=1, units='xy',
                   scale_units='xy', headlength=10, headwidth=5)

nodes_mesh(ax2, order=totalNumElements + 1)
```

```
fig1.show()

# Plot Element Strains
fig2 = plt.figure()
plt.subplots_adjust(hspace=0.7, wspace=0.7)
ax3 = fig2.add_subplot(211)
ax3.set_title('Element Strains', fontsize=17)
plt.xlabel('X', fontsize=10)
plt.ylabel('Y', fontsize=10)
plt.xlim(min(coords[:, 0]) - 0.5*prop[0, 0], max(coords[:, 0]) + 0.5*prop[0, 0])
plt.ylim(min(coords[:, 1]) - 0.5*prop[0, 0], max(coords[:, 1]) + 0.5*prop[0, 0])
originalMesh(ax3)

# Plot Deformed Mesh
d_ef = np.abs(coords - coordsd)
d_ef = np.sum(d_ef, axis=1)
z = rankmin(d_ef)
colors = pl.cm.jet(np.linspace(0, 1, len(set(z))))
Z = np.unique(z, return_index=False)
for elemNum in range(totalNumElements):
    xy = coordsdd[incid[elemNum, :] - 1, :]
    plot_gradient_rbg_pairs(xy[0, :], xy[1, :],
                            colors[np.where(z[incid[elemNum, 0] - 1] == Z)[0][0]],
                            colors[np.where(z[incid[elemNum, 1] - 1] == Z)[0][0]])

sm = plt.cm.ScalarMappable(cmap=plt.cm.jet,
                           norm=plt.Normalize(vmin=d_ef.min(), vmax=d_ef.max()))
sm._A = []
plt.colorbar(sm, ax=ax3, ticks=np.linspace(d_ef.min(), d_ef.max(), num=10))

# plot nodes of deformed mesh
nodes_mesh(ax3, coordsdd, order=totalNumElements + 1)

# Plot Element Stresses
ax4 = fig2.add_subplot(212)
ax4.set_title('Element stresses', fontsize=17)
```

```
plt.xlabel('X', fontsize=10)
plt.ylabel('Y', fontsize=10)
plt.xlim(min(coords[:, 0]) - 0.5*prop[0, 0], max(coords[:, 0]) + 0.5*prop[0, 0])
plt.ylim(min(coords[:, 1]) - 0.5*prop[0, 0], max(coords[:, 1]) + 0.5*prop[0, 0])
originalMesh(ax4)

# plot stresses on elements of the deformed mesh
z2 = rankmin(stress)
colors2 = pl.cm.jet(np.linspace(0, 1, len(set(z2))))
Z2 = np.unique(z2, return_index=False)
for elemNum in range(totalNumElements):
        ax4.plot(coordsdd[incid[elemNum, :] - 1, 0],
                 coordsdd[incid[elemNum, :] - 1, 1],
                 linewidth=1, alpha=1, c=colors2[np.where(z2[elemNum] == Z2)[0][0]])

sm = plt.cm.ScalarMappable(cmap=plt.cm.jet,
                           norm=plt.Normalize(vmin=stress.min(), vmax=stress.max()))
sm._A = []
plt.colorbar(sm, ax=ax4, ticks=np.linspace(stress.min(), stress.max(), num=10))

# plot nodes of deformed mesh
nodes_mesh(ax4, coordsdd, order=totalNumElements + 1)
plt.show()

# Output File
outFileName = inpFileName + 'py.out2'
with open(outFileName, 'w') as f:
    f.write('*DISPLACEMENTS\n')
    for i in range(totalNumNodes):
        f.write('{:d} {:.4f} {:.4f}\n'.format(i + 1, u[nodalDOFNumbers[0, i] - 1, 0],
                                              u[nodalDOFNumbers[1, i] - 1, 0]))
    f.write('\n*ELEMENT_STRAINS\n')
    for i in range(totalNumElements):
        f.write('{:d} {:.6e}\n'.format(i + 1, strain[i, 0]))
    f.write('\n*ELEMENT_STRESSES\n')
    for i in range(totalNumElements):
```

```
            f.write('{:d} {:.6e}\n'.format(i + 1, stress[i, 0]))
        f.write('\n*REACTION_FORCES\n')
        for i in range(totalNumRestrDOFs):
            if hdbcNodes[i, 1] == 1:
                f.write('{:d} FX = {:.6e}\n'.format(hdbcNodes[i, 0], rf[i, 0]))
            else:
                f.write('{:d} FY = {:.6e}\n'.format(hdbcNodes[i, 0], rf[i, 0]))
        f.write('\n*AREAS\n')
        f.write('{:d}\n'.format(nDIt))
        for i in range(totalNumElements):
            f.write('{:d} '.format(i + 1))
            for j in range(nDIt):
                f.write('{:e} '.format(areas[i, j]))
            f.write('\n')
        f.write('\n*VOLUMES\n')
        f.write('{:d}\n'.format(nDIt))
        for i in range(nDIt):
            f.write('{:e}\n'.format(np.asscalar(np.dot(areas[:, i], prop[:, 0]))))
        f.close()

if nDIt >= maxNumDesignIter:
    print('\nThe maximum number of design iterations has been reached.\n '
          'The stress for some bars may be above the given admissible stresses!!\n')
```

# E Arquivo de Saída truss2d _design.py

*DISPLACEMENTS
1 0.0000 0.0000
2 -0.0253 -0.2512
3 -0.0202 -0.4322
4 0.0000 -0.4976
5 0.0202 -0.4322
6 0.0253 -0.2512
7 0.0000 0.0000
8 0.1326 -0.1299
9 0.0947 -0.3536
10 0.0341 -0.4806
11 -0.0341 -0.4806
12 -0.0947 -0.3536
13 -0.1326 -0.1299

*ELEMENT _STRAINS
1 -5.052687e-05
2 1.010537e-05
3 4.042150e-05
4 4.042150e-05
5 1.010537e-05
6 -5.052687e-05
7 -1.016834e-04
8 6.778891e-05
9 -6.778891e-05
10 3.389445e-05
11 -3.389445e-05
12 9.930137e-20
13 1.986027e-19
14 -3.389445e-05
15 3.389445e-05
16 -6.778891e-05
17 6.778891e-05
18 -1.016834e-04

19 -7.579030e-05

20 -1.212645e-04

21 -1.364225e-04

22 -1.212645e-04

23 -7.579030e-05

*ELEMENT _STRESS

1 -1.061064e+02

2 2.122128e+01

3 8.488514e+01

4 8.488514e+01

5 2.122128e+01

6 -1.061064e+02

7 -2.135351e+02

8 1.423567e+02

9 -1.423567e+02

10 7.117835e+01

11 -7.117835e+01

12 2.085329e-13

13 4.170657e-13

14 -7.117835e+01

15 7.117835e+01

16 -1.423567e+02

17 1.423567e+02

18 -2.135351e+02

19 -1.591596e+02

20 -2.546554e+02

21 -2.864873e+02

22 -2.546554e+02

23 -1.591596e+02

*REACTION _FORCES

1 FX = 6.333333e+04

1 FY = 6.000000e+04

7 FX = -6.333333e+04

7 FY = 6.000000e+04

*AREAS

4

1 3.141500e+02 4.166667e+02 4.538919e+02 4.643172e+02

2 3.141500e+02 3.141500e+02 3.141500e+02 3.141500e+02

3 3.141500e+02 3.141500e+02 3.141500e+02 3.141500e+02

4 3.141500e+02 3.141500e+02 3.141500e+02 3.141500e+02

5 3.141500e+02 3.141500e+02 3.141500e+02 3.141500e+02

6 3.141500e+02 4.166667e+02 4.538919e+02 4.643172e+02

7 3.141500e+02 8.385255e+02 8.385255e+02 8.385255e+02

8 3.141500e+02 3.726780e+02 3.726780e+02 3.726780e+02

9 3.141500e+02 5.590170e+02 5.590170e+02 5.590170e+02

10 3.141500e+02 3.141500e+02 3.141500e+02 3.141500e+02

11 3.141500e+02 3.141500e+02 3.141500e+02 3.141500e+02

12 3.141500e+02 3.141500e+02 3.141500e+02 3.141500e+02

13 3.141500e+02 3.141500e+02 3.141500e+02 3.141500e+02

14 3.141500e+02 3.141500e+02 3.141500e+02 3.141500e+02

15 3.141500e+02 3.141500e+02 3.141500e+02 3.141500e+02

16 3.141500e+02 5.590170e+02 5.590170e+02 5.590170e+02

17 3.141500e+02 3.726780e+02 3.726780e+02 3.726780e+02

18 3.141500e+02 8.385255e+02 8.385255e+02 8.385255e+02

19 3.141500e+02 6.250000e+02 6.250000e+02 6.250000e+02

20 3.141500e+02 1.000000e+03 1.000000e+03 1.000000e+03

21 3.141500e+02 1.125000e+03 1.125000e+03 1.125000e+03

22 3.141500e+02 1.000000e+03 1.000000e+03 1.000000e+03

23 3.141500e+02 1.000000e+03 1.000000e+03 1.000000e+03

*VOLUMES

4

3.835207e+06 6.265324e+06 6.302550e+06 6.312975e+06

# F Arquivo de Entrada portico.py

```
*COORDINATES
11
1 0.0 0.0
2 5.0 0.0
3 10.0 0.0
4 15.0 0.0
5 20.0 0.0
6 25.0 0.0
7 2.5 4.0
8 7.5 4.0
9 12.5 4.0
10 17.5 4.0
11 22.5 4.0

*ELEMENT_GROUPS
2
1 5
2 14

*INCIDENCES
1 1 2
2 2 3
3 3 4
4 4 5
5 5 6
6 1 7
7 7 2
8 2 8
9 8 3
10 3 9
11 9 4
12 4 10
13 10 5
14 5 11
```

15 11 6
16 7 8
17 8 9
18 9 10
19 10 11

*MATERIALS
2
210E9 250E6 210E6 7850
210E9 250E6 210E6 7850

*GEOMETRIC_PROPERTIES
2
70.5E-4 4.47E-5 10.0E-2 -10.0E-2
60.0E-4 1.53E-5 7.5E-2 -7.5E-2
*BCNODES
4
1 1
1 2
1 3
6 2

*LOADS
3
3 1 2000
3 2 -10000
9 3 -5000

*DISTRIBUTED _LOADS
5
1 0 -500
2 0 -500
3 0 -500
4 0 -500
5 0 -500

# G portico.py

```
# Import libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.pylab as pl
from matplotlib import cm
from matplotlib.colors import LinearSegmentedColormap
import math

# Reading the input file name
br = 60 * "="
print(br + '\n' + 'PORTICO 2D' + '\n' + br + '\n' + '\n' + br + '\n' + br)
inputFileName = 'portico1'
fileName = inputFileName + '.fem'
with open(fileName, 'r') as myFile:
    file = myFile.read().splitlines()
myFile.close()


# Reading function
def readnum(keyword, file, type='int'):
    if keyword in file:
        totalnum = int(file[file.index(keyword) + 1])
        if totalnum > 0:
            num = pd.DataFrame(list(map(lambda x: x.split(),
                                        file[file.index(keyword) + 2:
                                             file.index(keyword) + 2 + totalnum])
                                    )
                               ).astype(type).as_matrix()
        else:
            num = pd.DataFrame()
        return totalnum, num
    else:
        raise ValueError("{} Not Found.".format(keyword))
```

```
def removefirstcol(df):
    return df[:, 1:]


# Reading nodal coordinates
print("READING NODAL COORDINATES...")
totalNumNodes, coords = readnum('*COORDINATES', file, type='float')
coords = removefirstcol(coords)

# Number of nodal degrees of freedom
numNodalDOFs = 3

# Reading element groups
print("READING ELEMENT GROUPS...")
numGroups, groups = readnum('*ELEMENT_GROUPS', file)
groups = removefirstcol(groups)

# Total number of elements
totalNumElements = groups.sum()

# Reading incidences
print("READING INCIDENCES...")
if '*INCIDENCES' in file:
    incid = pd.DataFrame(list(map(lambda groups: groups.split(),
                                  file[file.index('*INCIDENCES') + 1:
                                       file.index('*INCIDENCES') + totalNumElements + 1])
                              )
                         ).astype('int').as_matrix()
else:
    raise ValueError('{} Not Found.'.format('*INCIDENCES'))

# deletes element numbers
incid = removefirstcol(incid)
```

```
# Reading Materials
print('READING MATERIALS...')
numMaters, maters = readnum('*MATERIALS', file, type='float')

# Reading Geometric Properties
print('READING GEOMETRIC PROPERTIES...')
numGPs, gps = readnum('*GEOMETRIC_PROPERTIES', file, type='float')

# Reading Boundary Conditions
print('READING BOUNDARY CONDITIONS...')
numBCNodes, hdbcNodes = readnum('*BCNODES', file)

# Reading Nodal Loads
print('READING LOADS...')
numLoadedNodes, loads = readnum('*LOADS', file)
print(br)

# Reading Distributed Loads
print('READING DISTRIBUTED LOADS...')
numLoadedElements, distLoads = readnum('*DISTRIBUTED_LOADS', file)

# Element Orientation: length, lx = sin(theta), ly = cos(theta)
print('SOLUTION...\n' + br + '\n' + br)

# Element Lengths
lengths = np.sqrt(np.absolute((coords[incid[:, 0] - 1, 0] - coords[incid[:, 1] - 1, 0])**2 +
                              (coords[incid[:, 0] - 1, 1] - coords[incid[:, 1] - 1, 1])**2))

# lx = sin(theta)
lx = (coords[incid[:, 1] - 1, 1] - coords[incid[:, 0] - 1, 1]) / lengths

# ly = cos(theta)
ly = (coords[incid[:, 1] - 1, 0] - coords[incid[:, 0] - 1, 0]) / lengths

prop = np.matrix([lengths, lx, ly]).transpose()
```

```
# Distributed Load Global Matrix
globalDistLoads = np.zeros((totalNumElements, 2))

# Organize distributed loads in a matrix with all elements
globalDistLoads[distLoads[:, 0] - 1, :] = distLoads[:, 1:3]

# Solver
# DOF Numbering
# DOFs matriz initialized with ones
nodalDOFNumbers = np.ones((numNodalDOFs, totalNumNodes), dtype=int)

# Boundary Conditions on DOFs with Zero Values
for i in range(0, numBCNodes):
    nodalDOFNumbers[hdbcNodes[i, 1] - 1, hdbcNodes[i, 0] - 1] = 0

# Total Number of DOFs for the Model
totalNumDOFs = totalNumNodes * numNodalDOFs

# Numbers of Free and Restricted DOFs
totalNumFreeDOFs = np.sum(nodalDOFNumbers == 1)
totalNumRestrDOFs = np.sum(nodalDOFNumbers == 0)

# DOFs Numbering
ukeqnum = 0
keqnum = totalNumFreeDOFs
for i in range(totalNumNodes):
    for j in range(numNodalDOFs):
        if nodalDOFNumbers[j, i] == 1:
            ukeqnum = ukeqnum + 1
            nodalDOFNumbers[j, i] = ukeqnum
        elif nodalDOFNumbers[j, i] == 0:
            keqnum = keqnum + 1
            nodalDOFNumbers[j, i] = keqnum

# Assembling of the Global Stiffness Matrix and Global Distributed Loads Vector
# Creates Zero Global Matrix
```

```
kg = np.zeros((totalNumDOFs, totalNumDOFs))

# Distributed Loads
fgDistLoads = np.zeros((totalNumDOFs, 1))

elemNum = 0
for grp in range(0, numGroups):
    E = maters[grp, 0]
    rho = maters[grp, 3]
    A = gps[grp, 0]
    Iz = gps[grp, 1]

    for tne in range(0, groups[grp, 0]):
        c = prop[elemNum, 2]
        s = prop[elemNum, 1]
        l = prop[elemNum, 0]

        # Rotation Matrix
        T = np.array(([ c, s, 0,  0, 0, 0],
                      [-s, c, 0,  0, 0, 0],
                      [ 0, 0, 1,  0, 0, 0],
                      [ 0, 0, 0,  c, s, 0],
                      [ 0, 0, 0, -s, c, 0],
                      [ 0, 0, 0,  0, 0, 1]))

        # Element Stiffness Matrix
        # Axial Forces
        kb = E * A / l
        # Transversal Forces
        kf = E * Iz / l ** 3
        ke = np.array(([ kb,          0,              0, -kb,          0,              0],
                       [  0,    12 * kf,      6 * l * kf,   0,    -12 * kf,      6 * l * kf],
                       [  0, 6 * l * kf, 4 * l ** 2 * kf,   0, -6 * l * kf, 2 * l ** 2 * kf],
                       [-kb,          0,              0,  kb,          0,              0],
                       [  0,   -12 * kf,     -6 * l * kf,   0,     12 * kf,      -6 * l *kf],
                       [  0, 6 * l * kf, 2 * l ** 2 * kf,   0, -6 * l * kf, 4 * l ** 2 * kf]))
```

```
        ke = np.dot(np.dot((T.transpose()), ke), T)

        # Element DOFs
        elemEqs = nodalDOFNumbers[:, incid[elemNum, :] - 1]
        elemEqs = elemEqs.flatten('F').reshape(elemEqs.shape[0] * elemEqs.shape[1], 1)

        # Assembling into the Global Matrix
        kg[elemEqs - 1, elemEqs.reshape((-1)) - 1] = kg[elemEqs - 1, elemEqs.reshape((-1)) - 1] + ke

        # Distributed Loads
        # Applied Distributed Load Rotation Matrix
        Tq = np.array(([ c, s],
                       [-s, c]))

        # Element Distributed Load Vector
        q = np.dot(Tq, np.array((globalDistLoads[elemNum, 0],
                                 globalDistLoads[elemNum, 1] - rho * A * 9.81)))

        # Element Nodal Distributed Load Vector
        fe = l / 2 * np.array(([q[0], q[1], q[1] * l / 6, q[0], q[1], -q[1] * l / 6]))
        fe = np.dot(T.transpose(), fe)

        # Element DOFs
        elemEqs = nodalDOFNumbers[:, incid[elemNum, :] - 1]
        elemEqs = elemEqs.flatten('F')

        # Assembling into the Global Matrix
        fgDistLoads[elemEqs - 1, 0] = fgDistLoads[elemEqs - 1, 0] + fe

        elemNum += 1

# Assembling Global Load Vector
fgnLoads = np.zeros((totalNumDOFs, 1))

# Assembling Nodal Loads
for i in range(0, numLoadedNodes):
```

```
    # Element DOFs
    elemEqs = nodalDOFNumbers[loads[i, 1] - 1, loads[i, 0] - 1]

    # Assebmbling into the Global Vector
    fgnLoads[elemEqs - 1] = loads[i, 2]

fg = fgnLoads + fgDistLoads

# Solving system of equations
u = np.zeros((totalNumDOFs, 1))

u[0:totalNumFreeDOFs, :] = np.dot(np.linalg.inv(kg[:totalNumFreeDOFs,
                                                   :totalNumFreeDOFs]),
                                  fg[:totalNumFreeDOFs, 0].reshape(totalNumFreeDOFs, 1))

# Calculates Reaction Forces, Element Internal Loads
# Reaction Forces
rf = (np.dot(kg[totalNumFreeDOFs:, 0:totalNumFreeDOFs], u[0:totalNumFreeDOFs]) -
      fg[totalNumFreeDOFs:])

# Deformed Coordinates
coordsd = coords + u[np.concatenate(nodalDOFNumbers[0:2, :]) - 1].reshape(coords.shape[1],
                                                                          coords.shape[0]).transpose()
scaleFactor = 8e2
coordsdd = coords + scaleFactor * u[np.concatenate(nodalDOFNumbers[0:2, :]) -
                        1].reshape(coords.shape[1], coords.shape[0]).transpose()

# Element Normal Force, Shear Force, Internal Moment and Stresses for y-max and for y-min
nx = np.zeros((totalNumElements, 1))
vy = np.zeros((totalNumElements, 1))
mz = np.zeros((totalNumElements, 2))
stress_ymax = np.zeros((totalNumElements, 2))
stress_ymin = np.zeros((totalNumElements, 2))

# Calculates Strain and Stress for the Elements
elemNum = 0
```

```
for grp in range(0, numGroups):
    E = maters[grp, 0]
    A = gps[grp, 0]
    Iz = gps[grp, 1]
    ymax = gps[grp, 2]
    ymin = gps[grp, 3]

    for tne in range(0, groups[grp, 0]):
        c = prop[elemNum, 2]
        s = prop[elemNum, 1]
        l = prop[elemNum, 0]

        # Rotation Matrix
        T = np.array(([ c, s, 0,  0, 0, 0],
                      [-s, c, 0,  0, 0, 0],
                      [ 0, 0, 1,  0, 0, 0],
                      [ 0, 0, 0,  c, s, 0],
                      [ 0, 0, 0, -s, c, 0],
                      [ 0, 0, 0,  0, 0, 1]))

        # Element DOFs and Displacements
        elemEqs = nodalDOFNumbers[:, incid[elemNum, :] - 1]
        elemU = u[elemEqs[:] - 1].flatten('F')
        elemU = np.dot(T, elemU)

        # Normal Force
        nx[elemNum] = E * A * (-1 / l * elemU[0] + 1 / l * elemU[3])

        # Shear Force
        vy[elemNum] = E * Iz * 6 / l ** 2 * (2 / l * elemU[1] + elemU[2] -
                                             2 / l * elemU[4] + elemU[5])

        # Moments in Nodes 1 and 2
        mz[elemNum, 0] = E * Iz * (-6 / l ** 2 * elemU[1] - 4 / l * elemU[2] +
                                   6 / l ** 2 * elemU[4] - 2 / l * elemU[5])
        mz[elemNum, 1] = E * Iz * ( 6 / l ** 2 * elemU[1] + 2 / l * elemU[2] +
```

```
                                    -6 / l ** 2 * elemU[4] + 4 / l * elemU[5])

        # Stress
        stress_ymax[elemNum, 0] = nx[elemNum] / A - mz[elemNum, 0] / Iz * ymax
        stress_ymax[elemNum, 1] = nx[elemNum] / A - mz[elemNum, 1] / Iz * ymax
        stress_ymin[elemNum, 0] = nx[elemNum] / A - mz[elemNum, 0] / Iz * ymin
        stress_ymin[elemNum, 1] = nx[elemNum] / A - mz[elemNum, 1] / Iz * ymin

        elemNum += 1

# Output File
outFileName = inputFileName + 'py.out2'
with open(outFileName, 'w') as f:
    f.write('*DISPLACEMENTS\n')
    for i in range(totalNumNodes):
        f.write('{:d} {:.4f} {:.4f} {:.4f}\n'.format(i + 1, u[nodalDOFNumbers[0, i] - 1, 0],
              u[nodalDOFNumbers[1, i] - 1, 0], u[nodalDOFNumbers[2, i] - 1, 0]))

    f.write('\n*REACTION_FORCES\n')
    for i in range(totalNumRestrDOFs):
        if hdbcNodes[i, 1] == 1:
            f.write('{:d} FX = {:.6e}\n'.format(hdbcNodes[i, 0], rf[i, 0]))
        elif hdbcNodes[i, 1] == 2:
            f.write('{:d} FY = {:.6e}\n'.format(hdbcNodes[i, 0], rf[i, 0]))
        else:
            f.write('{:d} MZ = {:.6e}\n'.format(hdbcNodes[i, 0], rf[i, 0]))

    f.write('\n*ELEMENT_NORMAL_FORCE\n')
    for i in range(totalNumElements):
        f.write('{:d} {:.6e}\n'.format(i + 1, nx[i].item()))

    f.write('\n*ELEMENT_SHEAR_FORCE\n')
    for i in range(totalNumElements):
        f.write('{:d} {:.6e}\n'.format(i + 1, vy[i].item()))

    f.write('\n*ELEMENT_MOMENT_LIMITS\n')
```

```
    for i in range(totalNumElements):
        f.write('{:d} {:.6e} {:.6e}\n'.format(i + 1, mz[i, 0], mz[i, 1]))

    f.write('\n*ELEMENT_STRESSES_YMAX\n')
    for i in range(totalNumElements):
        f.write('{:d} {:.6e} {:.6e}\n'.format(i + 1, stress_ymax[i, 0], stress_ymax[i, 1]))

    f.write('\n*ELEMENT_STRESSES_YMIN\n')
    for i in range(totalNumElements):
        f.write('{:d} {:.6e} {:.6e}\n'.format(i + 1, stress_ymin[i, 0], stress_ymin[i, 1]))

    f.close()

# Post Processing
print('POST-PROCESSING...\n' + br)


# Original Mesh Function
def originalMesh(ax, num_elements=totalNumElements):
    for elemNum in range(num_elements):
        ax.plot(coords[incid[elemNum, :] - 1, 0], coords[incid[elemNum, :] - 1, 1],
                color='white', linewidth=1, alpha=0.5)


# Nodes of the Underformed Mesh Function
def nodes_mesh(ax, coords=coords, order=1):
    ax.scatter(x=coords[:, 0], y=coords[:, 1], c='yellow',
               s=10, alpha=0.5, zorder=order)


# Circular Quiver Function
def plot_circ_quiver(h, k, r, M, color):
    if M > 0:
        t = np.linspace(-math.pi/3, 4 * math.pi/3)
    else:
        t = np.linspace(4 * math.pi/3, -math.pi/3)
```

```
    x = r * np.cos(t) + h
    y = r * np.sin(t) + k
    plt.plot(x, y, color=color)

    ax = plt.gca()

    if M < 0:
        ax.annotate('', xy=(x[-1] - r/3, y[-1]), xytext=(x[-2] - r/3, y[-2]),
                    arrowprops=dict(color=color, arrowstyle="->"))
    else:
        ax.annotate('', xy=(x[-1] + r/3, y[-1]), xytext=(x[-2] + r/3, y[-2]),
                    arrowprops=dict(color=color, arrowstyle="->"))


# Color gradient function
def plot_gradient_hack(p0, p1, npts=20, cmap=None, **kw):
    """
    Draw a gradient between p0 and p1 using a colormap
    The **kw dictionary gets passed to plt.plot, so things like linestyle,
    linewidth, labels, etc can be modified directly.
    """
    x_1, y_1 = p0
    x_2, y_2 = p1

    X = np.linspace(x_1, x_2, npts)
    Xs = X[:-1]
    Xf = X[1:]
    Xpairs = zip(Xs, Xf)

    Y = np.linspace(y_1, y_2, npts)
    Ys = Y[:-1]
    Yf = Y[1:]
    Ypairs = zip(Ys, Yf)

    C = np.linspace(0, 1, npts)
```

```
    cmap = plt.get_cmap(cmap)
    # the simplest way of doing this is to just do the following:
    for x, y, c in zip(Xpairs, Ypairs, C):
        plt.plot(x, y, '-', c=cmap(c), **kw)


def plot_gradient_rbg_pairs(p0, p1, rgb0, rgb1, **kw):
    """Form the gradient from RGB values at each point
    The **kw dictionary gets passed to plt.plot, so things like linestyle,
    linewidth, labels, etc can be modified directly.
    """
    cmap = LinearSegmentedColormap.from_list('jet', (rgb0, rgb1))
    plot_gradient_hack(p0, p1, cmap=cmap, **kw)


# Color definition Function
def rankmin(x):
    x = np.round(x, decimals=4)
    u, inv, counts = np.unique(x, return_inverse=True, return_counts=True)
    csum = np.zeros_like(counts)
    csum[1:] = counts[:-1].cumsum()
    return csum[inv]


# Mesh and Boundary Conditions
plt.style.use('dark_background')
fig1 = plt.figure()
plt.subplots_adjust(hspace=0.7, wspace=0.7)

ax1 = fig1.add_subplot(211)
ax1.set_title('Portico', fontsize=17)
plt.xlabel('X', fontsize=10)
plt.ylabel('Y', fontsize=10)
plt.xlim(min(coords[:, 0]) - 0.5*prop[0, 0], max(coords[:, 0]) + 0.5*prop[0, 0])
plt.ylim(min(coords[:, 1]) - 0.5*prop[0, 0], max(coords[:, 1]) + 0.5*prop[0, 0])
```

```
# Plots Original Mesh
originalMesh(ax1)

# Plots Nodes of the Underformed Mesh
nodes_mesh(ax1)

# Plot Loads
r = np.mean(prop[:, 0])
fFe = 0.25 * loads[:, 2] * abs(max(prop[:, 0]).item()) / abs(max(loads[:, 2]).item())
for i in range(loads.shape[0]):
    if loads[i, 1] == 1:
        ax1.quiver(coords[loads[i, 0] - 1, 0] - fFe[i], coords[loads[i, 0] - 1, 1], fFe[i], 0,
                   color='red', scale=1, units='xy', scale_units='xy', headlength=10,
                   headwidth=5)
    elif loads[i, 1] == 2:
        ax1.quiver(coords[loads[i, 0] - 1, 0], coords[loads[i, 0] - 1, 1] - fFe[i], 0, fFe[i],
                   color='red', scale=1, units='xy', scale_units='xy', headlength=10,
                   headwidth=5)
    elif loads[i, 1] == 3:
        plot_circ_quiver(coords[loads[i, 0] - 1, 0], coords[loads[i, 0] - 1, 1],
                         r/5, loads[i, 2], 'red')

# Plot Distributed Loads
distL = (0.15 * (distLoads[:, 1:3]) * np.asscalar(abs(max(prop[:, 0])))) /
         max(np.sqrt(distLoads[:, 1]**2 + distLoads[:, 2]**2)))
num_arrows_per_element = 11
for i in range(numLoadedElements):
    dx = ((coords[incid[distLoads[i, 0] - 1, 1] - 1, 0] -
           coords[incid[distLoads[i, 0] - 1, 0] - 1, 0])
           / (num_arrows_per_element - 1))
    dy = ((coords[incid[distLoads[i, 0] - 1, 1] - 1, 1] -
           coords[incid[distLoads[i, 0] - 1, 0] - 1, 1])
           / (num_arrows_per_element - 1))
    for j in range(num_arrows_per_element):
        ax1.quiver(coords[incid[distLoads[i, 0] - 1, 0] - 1, 0] - distL[i, 0] + j * dx,
                   coords[incid[distLoads[i, 0] - 1, 0] - 1, 1] - distL[i, 1] + j * dy,
```

```
                    distL[i, 0], distL[i, 1],
                    color='red', scale=1, units='xy', scale_units='xy', headlength=10,
                    headwidth=5)

# Plot BCs
apoio = np.asscalar(0.7e-1*abs(max(prop[:, 0])))
for i in range(hdbcNodes.shape[0]):
    if hdbcNodes[i, 1] == 1:
        ax1.plot(coords[hdbcNodes[i, 0] - 1, 0] - apoio,
                 coords[hdbcNodes[i, 0] - 1, 1], marker='>', color='black',
                 markersize=8, markeredgecolor='blue')
    elif hdbcNodes[i, 1] == 2:
        ax1.plot(coords[hdbcNodes[i, 0] - 1, 0],
                 coords[hdbcNodes[i, 0] - 1, 1] - apoio,
                 marker='^', color='black', markersize=6, markeredgecolor='blue')
    elif hdbcNodes[i, 1] == 3:
        ax1.plot(coords[hdbcNodes[i, 0] - 1, 0], coords[hdbcNodes[i, 1] - 1, 1],
                 marker='X', color='blue', markersize=5)

# Plot Reaction Forces
ax2 = fig1.add_subplot(212)
ax2.set_title('Reaction Loads', fontsize=17)
plt.xlabel('X', fontsize=10)
plt.ylabel('Y', fontsize=10)
plt.xlim(min(coords[:, 0]) - 0.5*prop[0, 0], max(coords[:, 0]) + 0.5*prop[0, 0])
plt.ylim(min(coords[:, 1]) - 0.5*prop[0, 0], max(coords[:, 1]) + 0.5*prop[0, 0])
originalMesh(ax2)

# Plot Reaction Loads
fRF = 0.25 * (rf * abs(max(prop[:, 0]))) / abs(max(rf[:, 0]))
for i in range(hdbcNodes.shape[0]):
    if hdbcNodes[i, 1] == 1:
        ax2.quiver(coords[hdbcNodes[i, 0] - 1, 0] - fRF[i, 0],
                   coords[hdbcNodes[i, 0] - 1, 1], fRF[i, 0], 0,
                   color='blue', scale=1, units='xy',
                   scale_units='xy', headlength=10, headwidth=5)
```

```
    elif hdbcNodes[i, 1] == 2:
        ax2.quiver(coords[hdbcNodes[i, 0] - 1, 0],
                   coords[hdbcNodes[i, 0] - 1, 1] - fRF[i, 0],
                   0, fRF[i, 0], color='blue', scale=1, units='xy',
                   scale_units='xy', headlength=10, headwidth=5)
    elif hdbcNodes[i, 1] == 3:
        plot_circ_quiver(coords[hdbcNodes[i, 0] - 1, 0], coords[hdbcNodes[i, 0] - 1, 1],
                         r / 5, fRF[i, 0], 'blue')

nodes_mesh(ax2, order=totalNumElements + 1)
fig1.show()

# Plot Element Stresses Ymin
fig2 = plt.figure()
plt.subplots_adjust(hspace=0.7, wspace=0.7)
ax3 = fig2.add_subplot(211)
ax3.set_title(r'$Element\ stresses\ y_{min}$', fontsize=17)
plt.xlabel('X', fontsize=10)
plt.ylabel('Y', fontsize=10)
plt.xlim(min(coords[:, 0]) - 0.5*prop[0, 0], max(coords[:, 0]) + 0.5*prop[0, 0])
plt.ylim(min(coords[:, 1]) - 0.5*prop[0, 0], max(coords[:, 1]) + 0.5*prop[0, 0])
originalMesh(ax3)

# Plots Stresses on Elements of the Deformed Mesh
z = rankmin(stress_ymin[:, 0] - stress_ymin[:, 1])
colors = pl.cm.jet(np.linspace(0, 1, len(set(z))))
Z = np.unique(z, return_index=False)
for elemNum in range(totalNumElements):
    xy = coordsdd[incid[elemNum, :] - 1, :]
    plot_gradient_rbg_pairs(xy[0, :], xy[1, :], colors[np.where(z[incid[elemNum, 0] - 1]
                    == Z)[0][0]], colors[np.where(z[incid[elemNum, 1] - 1] == Z)[0][0]])

sm = plt.cm.ScalarMappable(cmap=plt.cm.jet,
                           norm=plt.Normalize(vmin=stress_ymin.min(), vmax=stress_ymin.max()))
sm._A = []
plt.colorbar(sm, ax=ax3, ticks=np.linspace(stress_ymin.min(), stress_ymin.max(), num=10))
```

```
nodes_mesh(ax3, coordsdd, order=totalNumElements + 1)

# Plot Element Stresses Ymax
ax4 = fig2.add_subplot(212)
ax4.set_title(r'$Element\ stresses\ y_{max}$', fontsize=17)
plt.xlabel('X', fontsize=10)
plt.ylabel('Y', fontsize=10)
plt.xlim(min(coords[:, 0]) - 0.5*prop[0, 0], max(coords[:, 0]) + 0.5*prop[0, 0])
plt.ylim(min(coords[:, 1]) - 0.5*prop[0, 0], max(coords[:, 1]) + 0.5*prop[0, 0])
originalMesh(ax4)

# Plots Stresses on Elements of the Deformed Mesh
z2 = rankmin(stress_ymax[:, 0] - stress_ymax[:, 1])
colors2 = pl.cm.jet(np.linspace(0, 1, len(set(z2))))
Z2 = np.unique(z2, return_index=False)
for elemNum in range(totalNumElements):
    xy = coordsdd[incid[elemNum, :] - 1, :]
    plot_gradient_rbg_pairs(xy[0, :], xy[1, :], colors2[np.where(z2[incid[elemNum, 0] - 1]
                    == Z)[0][0]], colors2[np.where(z2[incid[elemNum, 1] - 1] == Z2)[0][0]])

sm = plt.cm.ScalarMappable(cmap=plt.cm.jet,
                           norm=plt.Normalize(vmin=stress_ymax.min(), vmax=stress_ymax.max()))
sm._A = []
plt.colorbar(sm, ax=ax4, ticks=np.linspace(stress_ymax.min(), stress_ymax.max(), num=10))

nodes_mesh(ax4, coordsdd, order=totalNumElements + 1)
fig2.show()

# Plot Element Normal Force
fig3 = plt.figure()
plt.subplots_adjust(hspace=0.7, wspace=0.7)
ax5 = fig3.add_subplot(211)
ax5.set_title(r'$Element\ normal\ forces\ N_{x}$', fontsize=17)
plt.xlabel('X', fontsize=10)
plt.ylabel('Y', fontsize=10)
```

```
plt.xlim(min(coords[:, 0]) - 0.5*prop[0, 0], max(coords[:, 0]) + 0.5*prop[0, 0])
plt.ylim(min(coords[:, 1]) - 0.5*prop[0, 0], max(coords[:, 1]) + 0.5*prop[0, 0])
originalMesh(ax5)

# Plot Stresses on Elements of the Deformed Mesh
colors = plt.cm.jet(nx / float(nx.max()))
for elemNum in range(totalNumElements):
    ax5.plot(coordsdd[incid[elemNum, ] - 1, 0],
             coordsdd[incid[elemNum, ] - 1, 1],
             color=colors[elemNum].flatten())

sm = plt.cm.ScalarMappable(cmap=plt.cm.jet,
                           norm=plt.Normalize(vmin=nx.min(), vmax=nx.max()))
sm._A = []
plt.colorbar(sm, ax=ax5, ticks=np.linspace(nx.min(), nx.max(), num=10))
nodes_mesh(ax5, coordsdd, order=totalNumElements + 1)

# Plot Elements Shear Force
plt.subplots_adjust(hspace=0.7, wspace=0.7)
ax6 = fig3.add_subplot(212)
ax6.set_title(r'$Element\ shear\ forces\ V_{y}$', fontsize=17)
plt.xlabel('X', fontsize=10)
plt.ylabel('Y', fontsize=10)
plt.xlim(min(coords[:, 0]) - 0.5*prop[0, 0], max(coords[:, 0]) + 0.5*prop[0, 0])
plt.ylim(min(coords[:, 1]) - 0.5*prop[0, 0], max(coords[:, 1]) + 0.5*prop[0, 0])
originalMesh(ax6)

# Plot Stresses on Elements of the Deformed Mesh
colors = plt.cm.jet(vy / float(vy.max()))
for elemNum in range(totalNumElements):
    ax6.plot(coordsdd[incid[elemNum, ] - 1, 0],
             coordsdd[incid[elemNum, ] - 1, 1],
             color=colors[elemNum].flatten())

sm = plt.cm.ScalarMappable(cmap=plt.cm.jet,
                           norm=plt.Normalize(vmin=vy.min(), vmax=vy.max()))
```

```
sm._A = []
plt.colorbar(sm, ax=ax6, ticks=np.linspace(vy.min(), vy.max(), num=10))
nodes_mesh(ax6, coordsdd, order=totalNumElements + 1)

plt.show()
```

# H Arquivo de Saída portico.py

*DISPLACEMENTS
1 0.0000 0.0000 0.0000
2 0.0001 -0.0011 -0.0002
3 0.0002 -0.0018 -0.0000
4 0.0004 -0.0018 0.0001
5 0.0006 -0.0000 0.0005
6 0.0007 0.0000 0.0005
7 0.0007 -0.0006 -0.0004
8 0.0006 -0.0015 -0.0000
9 0.0003 -0.0019 -0.0005
10 0.0001 -0.0015 0.0002
11 -0.0000 -0.0006 0.0003

*REACTION _FORCES
1 FX = -2.000000e+03
1 FY = 3.453947e+04
1 MZ = 4.628388e+03
6 FY = 3.256920e+04

*ELEMENT _NORMAL _FORCE
1 2.092638e+04
2 5.061697e+04
3 5.796267e+04
4 4.628773e+04
5 1.867522e+04
6 -3.575801e+04
7 3.198599e+04
8 -2.370354e+04
9 1.870353e+04
10 1.602354e+03
11 -6.865460e+03
12 1.562586e+04
13 -2.136042e+04
14 3.065116e+04

15 -3.470902e+04
16 -3.605111e+04
17 -5.845598e+04
18 -5.464974e+04
19 -3.494404e+04

*ELEMENT _SHEAR _FORCE
1 2.092638e+04
2 5.061697e+04
3 5.796267e+04
4 4.628773e+04
5 1.867522e+04
6 -3.575801e+04
7 3.198599e+04
8 -2.370354e+04
9 1.870353e+04
10 1.602354e+03
11 -6.865460e+03
12 1.562586e+04
13 -2.136042e+04
14 3.065116e+04
15 -3.470902e+04
16 -3.605111e+04
17 -5.845598e+04
18 -5.464974e+04
19 -3.494404e+04

*ELEMENT _MOMENT _LIMITS
1 -1.691073e+03 8.283850e+02
2 2.971067e+02 4.351673e+02
3 -3.318106e+01 5.744862e+02
4 2.539214e+02 -2.335419e+02
5 -2.456406e+02 1.686062e+03
6 -3.105255e+02 -1.696408e+02
7 5.624957e+02 -3.953284e+02
8 1.359499e+02 1.192208e+02

9 -2.001063e+02 2.106178e+02
10 6.789661e+02 -1.300887e+03
11 1.315169e+03 -4.968529e+02
12 -1.762881e+02 2.704491e+02
13 -1.064959e+02 1.972894e+01
14 3.182758e+01 1.638978e+02
15 -6.138413e+02 9.407270e+02
16 2.304697e+02 1.679631e+02
17 4.872902e+02 -1.064093e+03
18 1.319850e+03 -4.590210e+02
19 -8.207592e+01 1.848671e+02

*ELEMENT _STRESSES _YMAX
1 6.751443e+06 1.115070e+06
2 6.515044e+06 6.206183e+06
3 8.295885e+06 6.936451e+06
4 5.997579e+06 7.088101e+06
5 3.198498e+06 -1.122985e+06
6 -4.437484e+06 -5.128095e+06
7 2.573666e+06 7.268882e+06
8 -4.617011e+06 -4.535006e+06
9 4.098169e+06 2.084815e+06
10 -3.061206e+06 6.643957e+06
11 -7.591152e+06 1.291310e+06
12 3.468468e+06 1.278579e+06
13 -3.038031e+06 -3.656781e+06
14 4.952509e+06 4.305106e+06
15 -2.775811e+06 -1.039624e+07
16 -7.138272e+06 -6.831867e+06
17 -1.213134e+07 -4.526522e+06
18 -1.557814e+07 -6.858187e+06
19 -5.421674e+06 -6.730218e+06

*ELEMENT _STRESSES _YMIN
1 -8.148813e+05 4.821491e+06
2 7.844380e+06 8.153241e+06

3 8.147424e+06 9.506859e+06

4 7.133693e+06 6.043171e+06

5 2.099436e+06 6.420919e+06

6 -7.481852e+06 -6.791240e+06

7 8.088330e+06 3.393114e+06

8 -3.284169e+06 -3.366174e+06

9 2.136342e+06 4.149696e+06

10 3.595324e+06 -6.109839e+06

11 5.302665e+06 -3.579797e+06

12 1.740153e+06 3.930042e+06

13 -4.082109e+06 -3.463360e+06

14 5.264544e+06 5.911947e+06

15 -8.793863e+06 -1.173430e+06

16 -4.878765e+06 -5.185170e+06

17 -7.353986e+06 -1.495881e+07

18 -2.638434e+06 -1.135839e+07

19 -6.226340e+06 -4.917796e+06