Universidade Federal do Rio Grande do Sul
Escola de Engenharia
Programa de Pós-Graduação em Engenharia Civil

Python aplicado à análise de estruturas

Daniel Barbosa Mapurunga Matos e Eduardo Pagnussat Titello

Porto Alegre
2020

LISTA DE FIGURAS

Figura 1 – Instaladores do Anaconda 5
Figura 2 – Interface do Anaconda Navigator 6
Figura 3 – Tela inicial do Spyder 6
Figura 4 – Tela inicial do Jupyter notebook 7
Figura 5 – Plotagem de figura utilizando o matplotlib 46
Figura 6 – Exemplo de subplot 47
Figura 7 – Exemplo de scatter plot 48
Figura 8 – Exemplo de integração Pandas-Matplotlib 49
Figura 9 – Plotagem de um *color map* utilizando o matplotlib 51
Figura 10 – Plotagem de superfície utilizando o matplotlib 52
Figura 11 – Barra de treliça no sistema local de coordenadas 53
Figura 12 – Barra de treliça no sistema global de coordenadas 54
Figura 13 – Exemplo de estrutura com barras de treliça plana 56
Figura 14 – Estrutura em estudo 60
Figura 15 – Plotagem da estrutura em estudo 64
Figura 16 – Plotagem da treliça deformada 70
Figura 17 – Plotagem das reações de apoio 72
Figura 18 – Plotagem dos esforços atuantes 74

SUMÁRIO

1	INTRODUÇÃO	5
1.1	Por que Python?	5
1.2	Instalação do Anaconda	5
1.3	Spyder: Ambiente semelhante ao MATLAB	6
1.4	Jupyter notebook: Caderno interativo	7
1.4.1	Células de código Python	7
1.4.2	Células de Markdown	7
2	A LINGUAGEM PYTHON	8
2.1	Entrada e Saída de dados	8
2.2	Tipos de variáveis	8
2.2.1	Numéricas	8
2.2.2	Textual	9
2.2.3	Booleanas	9
2.2.4	Listas, tuplas e dicionários	9
2.2.5	Extra: Formatando strings	13
2.3	Operadores	15
2.3.1	Aritiméticos	15
2.3.2	Atualização	15
2.3.3	Comparativos	16
2.3.4	Participação	17
2.3.5	Lógicos	17
2.4	Blocos	18
2.4.1	Condicionais	18
2.4.2	Repetição	19
2.5	Funções	21
3	MÓDULOS BÁSICOS- PARTE 1	24
3.1	Trabalhando com módulos	24
3.2	NumPy	25
3.2.1	Criação de arrays	25
3.2.1.1	np.array	25
3.2.1.2	np.zeros	27
3.2.1.3	np.ones	27
3.2.1.4	np.eye	28

3.2.1.5 np.diag . . . 28
3.2.1.6 np.linspace . . . 29
3.2.1.7 Outras formas: . . . 30
3.2.2 Manipulação de arrays . . . 30
3.2.2.1 Alteração de elemento . . . 30
3.2.2.2 Determinação do tamanho e forma . . . 30
3.2.2.3 Cópia de arrays . . . 32
3.2.2.4 Transposição de arrays . . . 33
3.2.3 Operações e funções matemáticas . . . 33
3.2.4 Algebra linear com arrays . . . 34
3.3 SciPy . . . 36
3.3.1 Determinação das raízes de uma função. . . . 37

4 MÓDULOS BÁSICOS- PARTE 2 . . . 38
4.1 O módulo Pandas . . . 38
4.1.1 Importação de dados . . . 38
4.1.2 Manipulação de dados . . . 41
4.1.3 Integração Pandas e Numpy . . . 42
4.1.4 Criação e exportação de um DataFrame . . . 43
4.2 O módulo matplotlib.pyplot . . . 45
4.2.1 Integração matplotlib e Pandas . . . 49
4.2.2 O que é possível fazer com o matplotlib? . . . 50

5 PROGRAMA PARA RESOLUÇÃO DE TRELIÇAS PLANAS . . . 53
5.1 Fundamentação teórica. . . . 53
5.1.1 Matriz de rigidez de uma barra de treliça plana. . . . 53
5.1.2 Montagem da matriz de rigidez global da estrutura . . . 56
5.1.3 Montagem do vetor de forças externas . . . 58
5.1.4 Aplicação das condições de contorno . . . 58
5.1.5 Cálculo dos deslocamentos . . . 59
5.1.6 Determinação das reações de apoio . . . 59
5.1.7 Determinação dos esforços em cada barra . . . 59
5.2 Introdução . . . 59
5.2.1 Esturutra em estudo . . . 60
5.3 Entrada de dados . . . 61
5.4 Plotagem da estrutura . . . 62
5.5 Comprimento das barras e cossenos diretores . . . 64
5.6 Montagem da matriz de rigidez . . . 66
5.6.1 Aplicação das condições de contorno . . . 67

5.7 Montagem do vetor de forças 68
5.8 Resolução da equação de equilíbrio 69
5.9 Plotagem da treliça deformada 69
5.10 Plotagem das reações de apoio 70
5.11 Determinação dos esforços 72

1 INTRODUÇÃO

1.1 POR QUE PYTHON?

Python é uma linguagem de código aberto, de fácil interpretação que exige menos o programador e mais da máquina, sendo, portanto, uma linguagem facilmente legível, que não prioriza a velocidade de execução. Hoje, a linguagem é amplamente utilizada devido ao vasto número de bibliotecas gratuitas disponíveis, que possibilitam uma infinidade realizações.

1.2 INSTALAÇÃO DO ANACONDA

Anaconda é uma distribuição gratuita da linguagem Python que possui diversas ferramentas, bibliotecas e ambientes.Sua instalação é aconselhável por já possuir as principais bibliotecas necessárias e um gerenciador de pacotes. Para baixar clique aqui e escolha a versão adequada para o seu sistema operacional.

Figura 1 – Instaladores do Anaconda

Ao abrir o instalador, basta clicar avançar em todas as janelas.

Após instalado, o seguinte ambiente poderá ser visualizado:

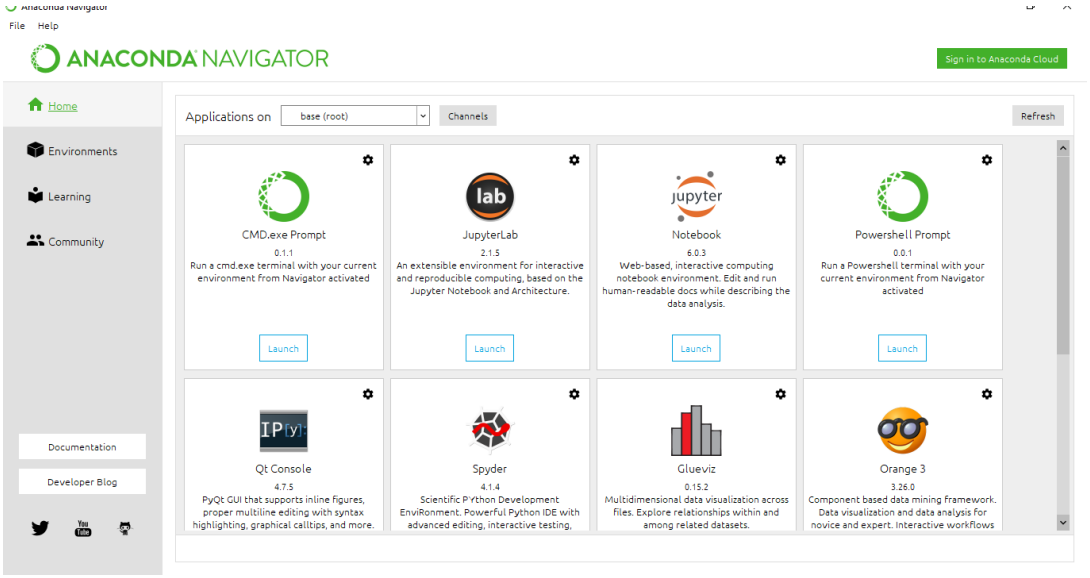


Figura 2 – Interface do Anaconda Navigator

Dos ambientes que podem ser visualizados na figura acima, os que serão abordados neste minicurso são Spyder e Jupyter notebook.

1.3 SPYDER: AMBIENTE SEMELHANTE AO MATLAB

O ambiente Spyder é bastante semelhante ao MATLAB, tendo, assim como este, um editor de código, um explorador de variáveis, figuras e arquivos e um console, como pode ser visto na figura abaixo.Este ambiente também possibilita a depuração do código, sendo possível o rastreamento de erros.

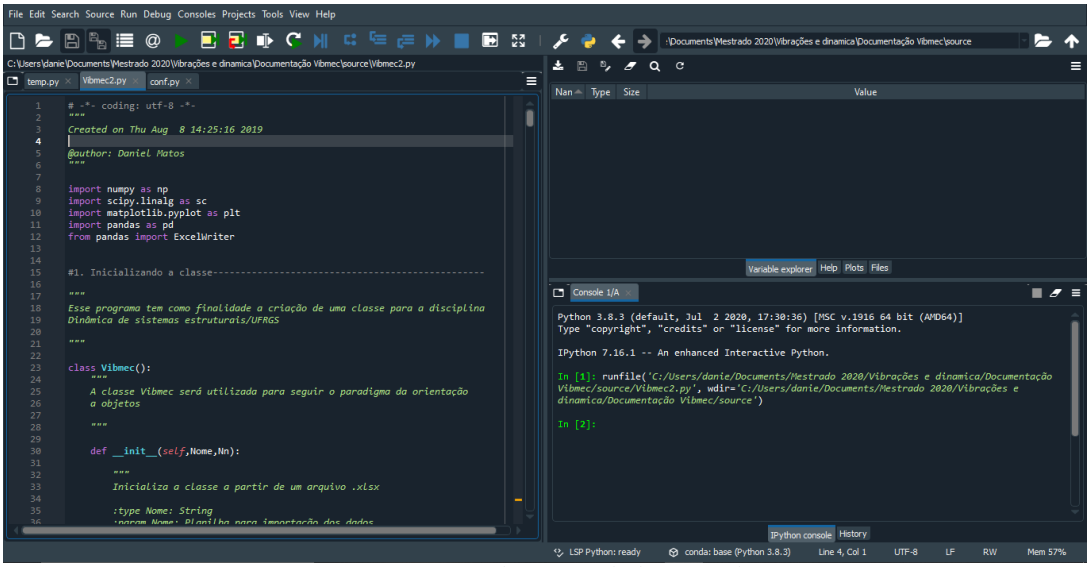


Figura 3 – Tela inicial do Spyder

1.4 JUPYTER NOTEBOOK: CADERNO INTERATIVO

O Jupyter notebook consiste numa aplicação que permite criar e compartilhar documentos com código ativo. Este permite que o código permite que o código seja construído por células, não havendo a necessidade rodar todo o programa a cada alteração feita no código. Neste ambiente são encontradas dois tipos principais de células, sendo elas Python e Markdown. A célula Python é utilizada para o processamento do código, enquanto a markdown é utilizada com o intuito de documentar o código. O ambiente Jupyter pode ser visualizado abaixo.

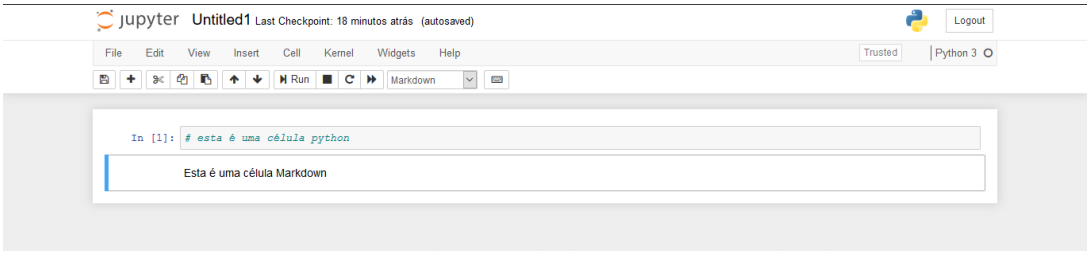


Figura 4 – Tela inicial do Jupyter notebook

1.4.1 Células de código Python

Estas células contém parte do código podendo ser processadas individualmente, obtendo resultados parciais durante o processo, que ficam gravados no arquivo. É importante verificar no canto superior esquerdo a sequência em que as células estão sendo processadas. Para processar uma célula, basta clicar no botão `Run` ou apertar as teclas `shift+enter`.

1.4.2 Células de Markdown

São células de documentação, onde podem ser inseridas equações em LateX, links, figuras e tabelas. Clique aqui para um guia rápido de Markdown. Para entender como funcionam as equações em LateX, clique aqui.

2 A LINGUAGEM PYTHON

2.1 ENTRADA E SAÍDA DE DADOS

Em Python, assim como todas as linguagens de programação, é possível solicitar dados e exibir informações na tela. Abaixo, será demonstrado a utiização dos comandos `print` e `input`.

[1]:
```
a = input('Digite a sua idade')                                    #␣
↪Criação de uma variável a partir da entrada do usuário
print( "A idade do usuário é:", a)                                  #␣
↪Exibição da variável na tela
```

```
Digite a sua idade
A idade do usuário é:
```

Mais a frente, após a explicação dos tipos de variáveis, será ensinado como formatar as informações que serão printadas na tela.

2.2 TIPOS DE VARIÁVEIS

As variáveis em Python possuem tipagem dinâmica, portanto, não há a necessidade de declara-las no ínicio do programa e, é possível, também, mudar o seu tipo ao longo do desenvolvimento do código.

2.2.1 Numéricas

A linguagem Python possui três tipos de variáveis para o armazenamento de números : os `int`, os `float` e os `complex`. As variáveis do tipo `int` armazenam apenas números inteiros, os `floats` armazenam números reais e os `complex` armazenam números complexos. **O divisor de décimais é marcado por um "." e o número complexo é indicado por "j"**. Abaixo são declaradas três variáveis, e seus tipos são exibidos na tela a patir do comando `type`.

[2]:
```
a = 1
b = 1.
c = 1+1j
```

```
print(type(a))
print(type(b))
print(type(c))
```

```
<class 'int'>
<class 'float'>
<class 'complex'>
```

2.2.2 Textual

Para o armazenamento de uma sequência de caracteres, utiliza-se uma variável do tipo `string`, em que esta sequência deve ser declarada entre aspas simples ou duplas.

[3]:
```
a = 'oi, mundo'
print(type(a))
```

```
<class 'str'>
```

Perceba que a variável a foi declarada novamente, mudando o seu tipo.

2.2.3 Booleanas

A linguagem Python também é dotada de operadores booleanos, ou seja `True` e `False`.

[4]:
```
a = 10
y = a > 10
print(y, type(y))
```

```
False <class 'bool'>
```

2.2.4 Listas, tuplas e dicionários

É possível, também, armazenar diversos valores dentro de uma só variável. As variáveis que possuem essa característica são as `lists`, as `tuples` e os `dictionarys`. As lists são sequências de variáveis que podem ser alteradas ao longo do código e são declaradas com o uso de colchetes.

[5]:
```
a = [1, 2, 'oi', 7.15 ,1+2j]
print(a, type(a))
```

```
[1, 2, 'oi', 7.15, (1+2j)] <class 'list'>
```

As `tuples`,diferentemente das `lists`, são imutáveis, ou seja, não podem ter seus valores alterados ao longo da execução.

[6]:
```
b = (1, 2, 'oi')
print(b, type(b))
```

```
(1, 2, 'oi') <class 'tuple'>
```

Para acessar um termo específico de uma lista ou de uma tupla, basta utilizar o seus nome seguido do seu "indexing" entre colchetes. Por exemplo, vamos acessar o **terceiro** termo da lista `a`. O último termo de uma sequência pode ser acessada utilizando `-1` como índice.

[7]:
```
print(a[2])
print(a[-1])
```

```
oi
(1+2j)
```

Em python, a contagem do indexing começa de zero, portanto, se você quer utilizar o terceiro termo de uma lista ou tupla, deve-se utilizar o indexing = 2.

É possível acessar os termos da lista e alterar seus valores específicos, como se mostra abaixo.

[8]:
```
a[2] = 'tchau'
print(a)
```

```
[1, 2, 'tchau', 7.15, (1+2j)]
```

Perceba que não é possível fazer o mesmo com uma tupla.

[9]:
```
b[2] = 'tchau'
```

```
---------------------------------------
TypeError                                 Traceback (most␣
↪recent call last)
<ipython-input-9-0a2d922de335> in <module>
----> 1 b[2] = 'tchau'

TypeError: 'tuple' object does not support item assignment
```

Além de poder chamar um valor específico da lista, também é possível realizar a chamada de um intervalo, da seguinte forma:

[10]:
```
print(a[0:3])
```

```
[1, 2, 'tchau']
```

Perceba que foram extraídos apenas os 3 primeiros números da lista.

É possível, ainda, escolher variáveis em um intervalo específico. Por exemplo, vamos escolher as variáveis de 2 em 2.

[11]:
```
print(a[::2])
```

```
[1, 'tchau', (1+2j)]
```

Para inverter a sequência de uma lista, pode-se utilizar o comando reversed, ou simplesmente...

[12]:
```
print(a[::-1])
```

```
[(1+2j), 7.15, 'tchau', 2, 1]
```

As variáveis do tipo `dictionary` são como listas onde os índices podem ser quaisquer tipos de valores. Dicionários são criados fazendo o uso de chaves (`{}`), onde é fornecido

o indíce (key) e seu valor seguido de `:`. Os valores dos dicionários são mutáveis e esses são expansíveis, conforme os exemplos a seguir:

[13]:
```
d = {'a': 1,
     'b': 2,
      30: 0,
     (2,3): 77}

print(d)
print(d['a'])
print(d[2,3])
print(d[30])
```

```
{'a': 1, 'b': 2, 30: 0, (2, 3): 77}
1
77
0
```

A mutação e a adiçao de valores ao dicionário são realizadas na forma:

[14]:
```
d['Python'] = 'Olá PPGEC!'
print(d)
```

```
{'a': 1, 'b': 2, 30: 0, (2, 3): 77, 'Python': 'Olá PPGEC!'}
```

Para remover valores é usado o comando `d.pop(key)` fornecendo a key a ser removida, esse retornará o valor removido. Por exemplo removendo o elemento `b` a função retorna 2.

[15]:
```
d.pop('b')
```

[15]: 2

[16]:
```
d
```

[16]: `{'a': 1, 30: 0, (2, 3): 77, 'Python': 'Olá PPGEC!'}`

Outro tipo de variável importante no Python é o `None`, que significa nenhum. Esse tipo de variável é bastante útil para representar a ausência de valores, visto que `0` tem

significado em alguns casos. Vamos declarar uma variável como `None` e verificar seu tipo:

[17]:
```
a = None              # a é None
print(a, type(a))     # printa seu valor e seu tipo
print(a == 0)         # None e 0 não são a mesma coisa!
```

```
None <class 'NoneType'>
False
```

2.2.5 Extra: Formatando strings

Strings são muito úteis na comunicação do programa com o usuário, através delas um conteúdo pré-determinado pode ser combinado com os valores de algumas variáveis resultando em uma saída formata como:

```
'A raiz da função é x=1.234 com um erro absoluto de 1.2E-4.'
```

Definindo os valores da raiz e do erro manualmente podemos obter o resultado anterior através de:

[18]:
```
raiz = 1.23456789123
erro = 1.2154656E-4
msg = 'A raiz da função é x={:.3f} com um erro absoluto de {:
 ↪.1E}.'.format(raiz, erro)
print(msg)
```

```
A raiz da função é x=1.235 com um erro absoluto de 1.2E-04.
```

Enquanto com a impressão sem formatação adotada até então temos:

[19]:
```
print('A raiz da função é x=', raiz,' com um erro absoluto␣
 ↪de ', erro, '.')
```

```
A raiz da função é x= 1.23456789123  com um erro absoluto de ␣
 ↪0.00012154656 .
```

Onde observamos que todas as casas decimais dos valores são impressas, mesmo não havendo necessidade de tamanha precisão.

A inserção de valores de variáveis em strings é realizada através da inserção de `{}` na posição do resultado no interior da string e da adição do comando `.format(variavel1, variavel2, ..., variaveln)`. No interior das chaves podem ser adicionadas informações como posição da variável à ser impressa e formato de impressão, no formato `posicao:formato`, onde ambas são opcionais.

A posição informada na formatação é a posição em que o variável a ser impressa consta em `.format()`, usualmente as variáveis são impressas na ordem em que são informadas, nesse caso a posição pdoe ser mantida em branco, como no exemplo anterior onde foi utilizado apenas `{:fomato}`. Vamos então executar o exemplo anterior alterando a ordem das variáveis e informando ou não a posição dos resultados à serem printados:

[20]:
```
msg1 = 'A raiz da função é x={:.3f} com um erro absoluto de␣
 ↪{:1.1E}.'.format(erro, raiz)
print(msg1)

msg2 = 'A raiz da função é x={1:.3f} com um erro absoluto de␣
 ↪{0:1.1E}.'.format(erro, raiz)
print(msg2)
```

```
A raiz da função é x=0.000 com um erro absoluto de 1.2E+00.
A raiz da função é x=1.235 com um erro absoluto de 1.2E-04.
```

No primeiro caso a string obtida apresentou os valores trocados, conforme esperado, no segundo a ordem de impressão foi fornecida e os resultados voltaram a ser condizentes.

Outra forma de organização dos valores para formatação é a denominação de nomes para as variáveis durante sua chamada em `.format()`, essas podem então ser posicionadas conforme seus nomes:

[21]:
```
print('A raiz da função é x={x:.3f} com um erro absoluto de␣
 ↪{e:.1E}.'.format(e=erro, x=raiz))
```

```
A raiz da função é x=1.235 com um erro absoluto de 1.2E-04.
```

Os principais formatos para impressão de números são: * `Xd` para impressão de números inteiros, onde X é o número mínimo de digitados reservados no espaço; * `.Xf` para impressão de números reais, onde X é o número de casas decimais; * `.Xe` e `.XE` para impressão de números reais na potencia 10, onde X é o número de casas decimais.

Vamos definir alguns números e printar esses formatados:

[22]:
```
a = 3
b = 3.14159
c = 123456789123.456789

print('a: {a}, {a:d}, {a:8d}, {a:.2f}'.format(a=a)) #␣
↪terceiro caso reserva 8 digitos na tela e deixa em branco!
print('b: {b}, {b:f}, {b:.2f}, {b:.8f}'.format(b=b))
print('c: {c}, {c:f}, {c:.2f}, {c:.1E}, {c:.5e}'.format(c=c))
```

```
a: 3, 3,        3, 3.00
b: 3.14159, 3.141590, 3.14, 3.14159000
c: 123456789123.45679, 123456789123.456787, 123456789123.46,␣
↪1.2E+11,
1.23457e+11
```

Os formatos não se limitam à impressão de números, esses permitem ainda a configuração de simbolos, alinhamentos e etc, e podem ser combinados em alguns casos. Maiores informações sobre formatos podem encontradas aqui e aqui.

2.3 OPERADORES

2.3.1 Aritiméticos

Os operadores aritiméticos do Python são: * `+`, `-`, `*`, `/` para adição, subtração, multiplicação e divisão; * `**` para exponenciação (`3^2` em Python é `3**2`); * `//` e `%` para divisão de inteiros e cálculo do seu resto. Por exemplo, `14/5 = 2.8`, já `14//5=2` (inteiro) e `14%5=4` (resto), logo `14=2*5+4`.

2.3.2 Atualização

Todos os operadores aritiméticos anteriores podem ser combinados com o símbolo de atribuição (=), obtendo operadores que atualizam o valores das variáveis. São exemplos `+=`, `*=`, `**=`, exemplíficados a seguir:

[23]:
```
a = 2          # Define a como 2
a += 5         # Atribuí mais 5 a `a`, 2+5=7.
print('a={}'.format(a))

b = 3          # Define b como 3
b *= 9         # Multiplica valor de b por 9, 3*9=27
print('b={}'.format(b))

c = 2          # Define c como 2
c **= 3        # Eleva c ao cubo 2**3=8
print('c={}'.format(c))
```

```
a=7
b=27
c=8
```

2.3.3 Comparativos

O conteúdo de variáveis pode ser comparado através dos operadores: * >, >=, <, <= para verificar se uma é maior, maior igual, menor ou menor igual; * == para verificar são iguais; * != para verificar são diferentes.

Vale ressaltar que = significa atribuição, e não comparação.

Por exemplo, usando as variáveis `a`, `b` e `c` anteriores:

[24]:
```
print('a<b? ', a<b)
print('c<a? ', c<a)
print('a=b? ', a==b)
print('a<>b? ', a!=b)
print('b=b? ', b==b)
```

```
a<b?  True
c<a?  False
a=b?  False
a<>b?  True
b=b?  True
```

2.3.4 Participação

A participação de um valor em um elemento (lista, dicionário, tupla, array) pode ser verificada através do comando `in`, conforme exemplo:

[25]:
```
a = [1, 3, 5, 7, 9]
print('Temos 1 em a?', 1 in a)
print('Temos 8 em a?', 8 in a)
```

```
Temos 1 em a? True
Temos 8 em a? False
```

2.3.5 Lógicos

Os operadores lógicos trabalham em conjunto com as operações booleanas, eles são `and`, `or` e `not` que tem como significado seus próprios nomes. Através desses podemos comparar operações para avaliar condições (vistas na próxima seção).

Como exemplo vamos adotar a lista `a` anterior, três escalares `b=0`, `c=5`, `d=88`, e fazer alguams verificações:

[26]:
```
b = 0
c = 5
d = 88

# Verificações
V1 = b <= c # b é menor ou igual à c? Sim
V2 = d in a # d está em a? Não
V3 = c in a # c está em a? Sim
print(V1, V2, V3)

# Agora vamos imaginar que precisamos saber se: `b<=c` (V1)␣
↪e `c` (V3) participa de `a`
V4 = V1 and V3 # Equivalente à `(b<=c) and (c in a)`
print(V4) # True pq ambas condições são verdadeias

# Agora supondo que queremos saber se `d` NÃO está em `a`
V5 = d not in a # Mesmo que `not V2`
print(V5) # True pq `d` NÃO está mesmo em `a`
```

```
True False True
True
True
```

2.4 BLOCOS

2.4.1 Condicionais

Se imaginarmos um fluxograma, um bloco condicional é um separador de fluxo definido por uma condição imposta pelo usuário. Por exemplo, vamos imaginar que você precisa validar uma senha de 8 dígitos, a primeira condição a ser verificada é o tamanho da senha, **se** essa condição for verdadeira a senha deve ser confirmada, **se** as senhas forem iguais e tiverem mais de 8 dígitos é informado ao usuário que a senha foi aceita.

[27]:
```
senha1 = input("Digite uma senha no mínimo 8 dígitos:")
if len(senha1) < 8:
  print("Senha inválida, a senha precisa ter no mínimo 8␣
↪digitos.")
else:
  senha2 = input("Por favor, confirme sua senha:")
  if senha1 == senha2:
    print("Senha aceita.")
  elif len(senha2) < 8:
    print("Senha inválida, a senha precisa ter no mínimo 8␣
↪digitos.")
  else:
    print("As senhas inseridas são diferentes!")
```

```
Digite uma senha no mínimo 8 dígitos:12345678
Por favor, confirme sua senha:12345678
Senha aceita.
```

No código anterior aplicamos as estruturas `if`, `elif` e `else`. A estrutura `if` inicia a verificação de uma condição, caso essa não seja verdadeira o código passa para a próxima condição contida em `elif`. O comando `else` é ativado quando nenhuma das condições anteriores é satisfeita.

2.4.2 Repetição

Observando o exemplo acima, veja que ele não faz sentido visto que se a senha digitada não for válida o código apenas termina sem a possibilidade de inserção de uma nova senha. A forma correta de executar o código acima é dentro de um loop em que esse loop só acabará quando a condição for satisfeita. Vamos então usar o artífício `while True:` onde o código é repetido infinitamente até que o loop seja quebrado.

[28]:
```
while True:
  senha1 = input("Digite uma senha no mínimo 8 dígitos:")
  if len(senha1) < 8:
    print("Senha inválida, a senha precisa ter no mínimo 8␣
↪digitos.")
  else:
    senha2 = input("Por favor, confirme sua senha:")
    if senha1 == senha2:
      print("Senha aceita.")
      break # Quebra o loop
    elif len(senha2) < 8:
      print("Senha inválida, a senha precisa ter no mínimo 8␣
↪digitos.")
    else:
      print("As senhas inseridas são diferentes!")
```

```
Digite uma senha no mínimo 8 dígitos:12345679
Por favor, confirme sua senha:12345679
Senha aceita.
```

Outras condições, como as utilizadas em `if`, podem ser consideradas pelo comando `while`. Como por exemplo `while x1!=x2` irá repetir o bloco de código até que `x1` seja igual à `x2`, sem a necessidade de uso do `break`.

Para iterar uma lista ou sequência pode ser usado o comando `for`. Diferentemente de outras linguagens de programação, o comando `for` tem como entrada um objeto iterável (como uma lista, tupla, dicionário ou sequencias).

Por exemplo vamos criar uma lista de valores e percorrer esta, onde em cada loop um elemento da lista é acessado e tem seu valor passado para a variável `n`.

[29]:
```
lista = [3, 4, 1, 23, 42, 5]
for n in lista:
  print(n)
```

```
3
4
1
23
42
5
```

Esse comando é muito utilizado para acessar e percorrer arrays, realizar processo iterativos e muito mais. Para criar um `for` tradicional, ou seja, em que a variável `n` segue uma sequencia de números inteiros pode ser usado o comando `range(inicio, fim, passo)` ou `range(fim)`, conforme exemplos à seguir:

[30]:
```
for n in range(10,20): # Passo não informado, adota padrão =1
  print(n)
```

```
10
11
12
13
14
15
16
17
18
19
```

[31]:
```
for n in range(10,20,2): # O passo agora é 2
  print(n)
```

```
10
12
14
```

```
16
18
```

[32]:
```
for i in range(7):
  print(i)
```

```
0
1
2
3
4
5
6
```

Podemos observar que o iterador `range()` cria uma sequencia de números inteiros a partir de um intervalo definido, matemáticamente falando temos um invervalo fechado-aberto.

Para numerar os termos de uma lista podemos acrescentar a função `enumerate` na chamada do `for`, conforme o exemplo a seguir, utilizando a lista dos exemplos anteriores:

[33]:
```
for i, n in enumerate(lista):
  print('O termo {} é {}.'.format(i,n))
```

```
O termo 0 é 3.
O termo 1 é 4.
O termo 2 é 1.
O termo 3 é 23.
O termo 4 é 42.
O termo 5 é 5.
```

2.5 FUNÇÕES

Uma função é um bloco que realiza uma operação pré-definida que será realizada sempre que chamada pelo usuário. Podemos criar funções para representar expressões matemáticas, ou mesmo operações mais trabalhosas. Funções apresentar valores de retorno. Adotando equações de segundo grau no formato $y = ax^2 + bx + c$ vamos

definir uma função `quad(x,a,b,c)` e uma função `raizes(a,b,c)`, onde a primeira retorna o valor de `y` em um dado `x` e a segunda através da formula de Bhaskara retorna as raizes.

Funções são declaradas no formato `def nome(arg1, arg2, arg3, ...):` e quando possuem retorno a última linha do bloco de função deve conter `return saida1, saida2, ....`

[34]:
```
# Função quad que retorna valor de y
def quad(x, a, b, c):
  y = a*x**2 + b*x + c
  return y

def raizes(a, b, c):
  delta = b**2-4*a*c
  if delta >= 0:
    x1 = (-b+delta**0.5)/(2*a)
    x2 = (-b-delta**0.5)/(2*a)
  else:
    print('As raizes da expressão são complexas.')
    x1, x2 = None, None

  return x1, x2
```

Perceba que na definição da função nada é executado, agora chamando essas teremos alguns resultados.

[35]:
```
r1, r2 = raizes(1, -3, 2)
print(r1, r2)
```

```
2.0 1.0
```

Como resultado a função retornou nas variáveis `r1` e `r2` as raizes da equação. Podemos agora aplicar `r1` ou `r2` em `quad` e verificar se `y` é realmente zero nas raizes.

[36]:
```
y = quad(r1, 1, -3, 2)
print(y)
```

```
0.0
```

Os argumentos das funções podem ter valores pré-definidos, dessa forma seu preenchimento é opcional. Além disso, na chamada das funções deve-se respeitar a ordem dos argumentos ou deve ser especificado o argumento que recebe cada valor.

Vamos então redefinir a função `quad` com argumentos padrões (`a=1, b=c=0`) e chama-la embaralhando seus argumentos.

[37]:
```
def quad(x, a=1, b=0, c=0):
  y = a*x**2 + b*x + c
  return y
```

Chamando a função fornecendo apenas o valor de `x` essa usará a expressão padrão $y = x^2$, dados os valores de `a,b,c`.

[38]:
```
quad(2)
```

[38]: 4

Agora embaralhando os argumentos podemos verificar a outra raiz obtida anteriormente.

[39]:
```
quad(c=2, x=r2, a=1, b=-3)
```

[39]: 0.0

Ok! A função retornou o valor esperado.

3 MÓDULOS BÁSICOS- PARTE 1

3.1 TRABALHANDO COM MÓDULOS

Módulos são bibliotecas de funções e classes que podem ser facilmente importados e introduzem novas funcionalidades à linguagem. Os principáis módulos para aplicações científicas são: - NumPy para operações matriciais; - SciPy para cálculo numérico; - matplotlib para plotagem de gráficos; - Pandas para trabalho com séries de dados.

Além desses, diversos outros módulos criados pela comunidade estão disponíveis nos repositórios do PyPi e do Anaconda. A criação de módulos pelo usuário pode ser realizada de forma simplificada inserindo todas as funções e variáveis de interesse em um arquivo `.py` (ou em uma pasta) no diretório de trabalho.

A utilização de módulos requer sua importação através do comando `import`, o código a seguir realiza a importação do módulo `scipy` e solicita ao Python o tipo de `scipy`.

[1]:
```
import scipy
print(type(scipy))
```

```
<class 'module'>
```

A importação de módulos pelo comando `import` mantem um vínculo entre o conteúdo importado e o nome do módulo, dessa forma todas suas funções devem ser chamadas na forma `modulo.funcao(...)` onde `funcao(...)` é a função de interesse que pertence ao módulo.

A importação de módulos pode ainda ser realizada na forma "importar como", onde é criado um "apelido" para o módulo, através da sintaxe `import modulo as apelido`. A seguir é importado o módulo `numpy` com o "apelido" `np` e é printando o tipo de `np`.

[2]:
```
import numpy as np
print(type(np))
```

```
<class 'module'>
```

Existem ainda módulos internos à outros módulos, sua importação é realizada na forma `import modulo.submodulo`. Um exemplo de submódulo é o `pyplot` que pertence ao módulo `matplotlib`, usualmente sua importação é realizada com o "apelido" `plt`, dessa forma:

[3]:
```
import matplotlib.pyplot as plt
```

A importação de módulos pode ainda ser realizada sem a manutenção de vínculo entre o nome do módulo e a função através da sintaxe `from modulo import funcao`, entretanto tal prática não é recomendada na presença de diferentes módulos.

3.2 NUMPY

O NumPy é o principal módulo para operação de matrizes e similares em Python. Sua base são arrays representadas através do objeto `ndarray` que é basicamente uma tabela de dimensão `n`, podendo assim representar escalares, vetores, matrizes e outros elementos de ordem superior.

Usualmente o NumPy é importado como `np`, conforme realizado anteriormente (`import numpy as np`).

3.2.1 Criação de arrays

Arrays do NumPy podem ser criadas de diversas maneiras.

3.2.1.1 np.array

Arrays podem ser criadas a partir de listas e tuplas com a função `np.array(lista)`. Ao passar uma lista de valores para a função um vetor é gerado, uma lista de listas produz uma matriz e etc.

Como exemplo vamos criar e printar 3 listas: `l1`, `l2` e `l3`, onde `l3` é uma lista de listas formada pelas outras duas.

[6]:
```
l1 = [1,3,5]
l2 = [20,30,40]
l3 = [l1, l2]

print('l1:', l1)
print('l2:', l2)
print('l3:', l3)
```

```
l1: [1, 3, 5]
l2: [20, 30, 40]
l3: [[1, 3, 5], [20, 30, 40]]
```

Tranformando as três listas em arrays temos 2 vetores (`v1` e `v2`) e uma matriz (`m1`):

[7]:
```
v1 = np.array(l1)
v2 = np.array(l2)
m1 = np.array(l3)

print('v1:\n', v1)
print('v2:\n', v2)
print('m1:\n', m1)
```

```
v1:
 [1 3 5]
v2:
 [20 30 40]
m1:
 [[ 1  3  5]
 [20 30 40]]
```

Os termos dos vetores e da matriz gerada podem ser acessados como os itens de listas, tuplas e dicionários:

[8]:
```
print('Primeiro termo de v2:', v2[0])
print('Último termo da segunda linha de m1:', m1[1,-1])
```

```
Primeiro termo de v2: 20
Último termo da segunda linha de m1: 40
```

Linhas e colunas inteiras ou parte dessas também podem ser acessadss:

[9]:
```
print('Primeira linha completa de m1:', m1[0,:])
print('Última coluna de m1:', m1[:,2])
```

```
Primeira linha completa de m1: [1 3 5]
Última coluna de m1: [ 5 40]
```

3.2.1.2 np.zeros

Arrays de zeros podem ser criadas através da função `np.zeros(dims)` fornecendo as dimensões desejadas em uma lista ou tupla de dimensões, por exemplo:

Para um vetor de tamanho 5:

[11]:
```
v_zeros1 = np.zeros([5])
print(v_zeros1)
```

```
[0. 0. 0. 0. 0.]
```

Para uma matriz de 3 linhas 4 colunas:

[13]:
```
m_zeros1 = np.zeros([3,4])
print(m_zeros1)
```

```
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
```

3.2.1.3 np.ones

Arrays unitários podem ser criados através da função `np.ones(dims)` forncendo as dimensões como em `np.zeros(dims)`.

Para uma matriz 3x4:

[14]:
```
m_ones1 = np.ones((3,4))
print(m_ones1)
```

```
[[1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]]
```

Essa função pode ser usada para outros valores, como um vetor onde todos termos são 25:

[16]:
```
v_25s = 25*np.ones((6))
print(v_25s)
```

```
[25. 25. 25. 25. 25. 25.]
```

3.2.1.4 np.eye

Matrizes identidade podem ser criadas através da funcão `np.eye(dim1,dim2)` passando as dimensões da matriz, caso apenas uma dimensão seja informada será gerada uma matriz quadrada.

Para gerar uma matriz identidade 4x4:

[39]:
```
m_id4 = np.eye(4)
print(m_id4)
```

```
[[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]]
```

Para uma matriz identidade 3x6

[40]:
```
m_id46 = np.eye(4,6)
print(m_id46)
```

```
[[1. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0.]]
```

3.2.1.5 np.diag

A função `np.diag(array)` ao receber uma matriz como entrada retorna um vetor com os termos da sua diagonal, o inverso também é valido, ao receber um vetor a função retorna uma matriz com os termos do vetor na diagonal.

Criando uma matriz 3x3 e extraindo sua diagoal:

[17]:
```
m3 = np.array([[1,2,3],
               [4,5,6],
               [7,8,9]])
diag = np.diag(m3)
```

```
print(diag)
```

```
[1 5 9]
```

Ok! A função retornou os termos da diagonal, vamos agora construir uma matriz com base nesse vetor:

[19]:
```
m4 = np.diag(diag)
print(m4)
```

```
[[1 0 0]
 [0 5 0]
 [0 0 9]]
```

A matriz gerada contem apenas termos na diagonal.

3.2.1.6 np.linspace

A função `np.linspace(inicio, fim, nptos)` cria um vetor de `nptos` igualmente espaçados no intervalo `[inicio, fim]`.

Criando 10 valores de 1 a 10:

[43]:
```
v1_10 = np.linspace(1, 10, 10)
print(v1_10)
```

```
[ 1.  2.  3.  4.  5.  6.  7.  8.  9. 10.]
```

Criando 12 valores entre 5 e 40:

[44]:
```
v5_40 = np.linspace(5, 40, 12)
print(v5_40)
```

```
[ 5.          8.18181818 11.36363636 14.54545455 17.72727273␣
↪20.90909091
 24.09090909 27.27272727 30.45454545 33.63636364 36.81818182␣
↪40.        ]
```

3.2.1.7 Outras formas:

Existem ainda diversas outras funções para criação de arrays no NumPy, essas podem ser acessadas aqui.

3.2.2 Manipulação de arrays

Arrays podem ser modificadas, empilhadas, operadas matricialmente e etc.

3.2.2.1 Alteração de elemento

A alteração de um elemento de um array pode ser realizada como a alteração de um elemento de uma lista. Adotando a matriz diagonal `m4` criada anteriormente podemos alterar o termo da terceira coluna e primeira linha por:

[45]:
```
m4[0, 2] = 99
print(m4)
```

```
[[ 1  0 99]
 [ 0  5  0]
 [ 0  0  9]]
```

lembrando que os índices em Python iniciam em `0`, dessa forma a primeira linha é a linha `0` e a terceira coluna é a coluna `2`.

Toda uma linha ou coluna pode ser alterada, usando `:` para sinalizar todos seus elementos. A seguir vamos definir todos elementos da primeira coluna de `m4` como `-1`.

[46]:
```
m4[:,0] = -1
print(m4)
```

```
[[-1  0 99]
 [-1  5  0]
 [-1  0  9]]
```

3.2.2.2 Determinação do tamanho e forma

O tamanho e a forma de arrays podem ser determinados pelos comandos `size` e `shape`, respectivamente, onde ambos podem ser acessados tanto através da biblioteca principal (`np.size(array)` e `np.shape(array)`) como através do próprio elemento

(`array.size`, `array.shape`), onde `array` é uma array genérica. Adotando a array `m4` vamos plotar sua forma e tamanho pelos dois métodos apresentados:

[20]:
```
print('Forma de m4')
print(np.shape(m4))
print(m4.shape)
print('Tamanho de m4')
print(np.size(m4))
print(m4.size)
```

```
Forma de m4
(3, 3)
(3, 3)
Tamanho de m4
9
9
```

Como resultado para a forma de `m4` obtivemos `(3,3)`, o que significa que a array `m4` tem 3 linhas e 3 colunas. Já o tamanho de `m4` encontrado é `9`, ou seja, `m4` tem 9 elementos (`3x3=9`). Vamos printar agora os tamanhos e formas de outros vetores e matrizes:

[48]:
```
print('Forma e tamanho de m_id46')
print(m_id46.size)
print(m_id46.shape)

print('Forma e tamanho de v1_10')
print(v1_10.size)
print(v1_10.shape)
```

```
Forma e tamanho de m_id46
24
(4, 6)
Forma e tamanho de v1_10
10
(10,)
```

3.2.2.3 Cópia de arrays

A realização de cópias de arrays deve ser feita através do comando `np.copy()` ou do método `array.copy()`, visto que uma simples atribuição `arrayA=arrayB` faz uma conexão de memória que poderá ocasionar em erros futuros.

[49]:
```
print('m4 original:\n', m4)
A = m4
B = m4.copy() #ou B=np.copy(m4)
A[1,1] = 666
B[1,1] = 999
print('m4:\n', m4)
print('A:\n', A)
print('B:\n', B)
```

```
m4 original:
 [[-1  0 99]
 [-1  5  0]
 [-1  0  9]]
m4:
 [[ -1   0  99]
 [ -1 666   0]
 [ -1   0   9]]
A:
 [[ -1   0  99]
 [ -1 666   0]
 [ -1   0   9]]
B:
 [[ -1   0  99]
 [ -1 999   0]
 [ -1   0   9]]
```

[50]:
```
print(A is m4)
print(B is m4)
```

```
True
False
```

3.2.2.4 Transposição de arrays

O transposto de uma array é obtida adicionando `.T` à sua chamada. No caso de vetores o resultado é indiferente. Como exemplo vamos transpor a matriz `m_id46` e o vetor `v1_10`.

[79]:
```
print(m_id46.shape)
print(m_id46.T.shape)

print(v1_10.shape)
print(v1_10.T.shape)
```

```
(4, 6)
(6, 4)
(10,)
(10,)
```

3.2.3 Operações e funções matemáticas

O módulo Numpy possui uma série de funções e constantes matemáticas, como:

- np.exp(): função exponencial;
- np.sin(), np.cos(), np.tan(), etc : funções trigonométricas;
- np.sqrt() : realiza a raiz quadrada;
- np.pi: π;
- np.log(): logarítmo natural;
- np.log10(): logarítmo na base 10.

Diferentemente das listas, as arrays do NumPy são operadas termo a termo, dessa forma as funções anteriores possuem a mesma propriedade. Vamos então ilustrar calculando a expressão a seguir para valores de x de 0 à 10.

$$y(x) = \exp^{5x-10} \sin(3\pi + \sqrt{4x})$$

[21]:
```
x = np.linspace(0,10,41) # 41 valores resulta em valores a␣
↪cada 0.25
print(x)
```

```
[ 0.    0.25  0.5   0.75  1.    1.25  1.5   1.75  2.    2.25 ␣
↪2.5   2.75
  3.    3.25  3.5   3.75  4.    4.25  4.5   4.75  5.    5.25 ␣
↪5.5   5.75
  6.    6.25  6.5   6.75  7.    7.25  7.5   7.75  8.    8.25 ␣
↪8.5   8.75
  9.    9.25  9.5   9.75 10.  ]
```

[69]:
```
y = np.exp(+5*x-10)*np.sin(3*np.pi+np.sqrt(4*x))
print(y)
```

```
[ 1.66796636e-20 -1.33340607e-04 -5.46317906e-04 -1.
↪90540967e-03
 -6.12679787e-03 -1.85025661e-02 -5.23831686e-02 -1.
↪36310914e-01
 -3.08071742e-01 -4.92557226e-01  2.51976998e-01  7.
↪40461201e+00
  4.70391298e+01  2.31806467e+02  1.02099414e+03  4.
↪21494202e+03
  1.66696843e+04  6.39132894e+04  2.39271597e+05  8.
↪78680102e+05
  3.17512400e+06  1.13139890e+07  3.98151706e+07  1.
↪38518524e+08
  4.76743911e+08  1.62383558e+09  5.47423593e+09  1.
↪82625012e+10
  6.02643613e+10  1.96557493e+11  6.32895323e+11  2.
↪00829404e+12
  6.26415699e+12  1.91329134e+13  5.68889302e+13  1.
↪63091288e+14
  4.43156739e+14  1.10207358e+15  2.28945992e+15  2.
↪57468835e+15
 -9.73511413e+15]
```

3.2.4 Algebra linear com arrays

Arrays podem representar vetores e matrizes, todavia a multiplicação de uma array `x` por ela mesma `x*x` resulta em uma operação termo a termo. A multiplicação matricial

da array `x` por ela mesma é realizada pela função `np.dot(x,x)` ou `x.dot(x)`. Cabe lembrar que no caso de matrizes o número de colunas de primeira matriz (`n x j`) deve ser igual o número de linhas da segunda matriz (`j x m`), que podem ser acessados por `np.shape()`, resultando em uma matriz `n x m`. Já no caso de vetores esses devem ter o mesmo comprimento, não havendo diferença entre vetores linha e vetores coluna.

Adotando duas matrizes `X1(4,2)` e `X2(2,6)` temos que:

[45]:
```
np.random.seed(8198916)
X1 = np.random.randint(0, 10, (4,2)) # Gera número inteiros␣
↪aleatórios entre 0 e 10
X2 = np.random.randint(0, 10, (2,4))
print('X1:\n', X1)
print('X2:\n', X2)

X3 = X1.dot(X2)
print('X3:\n', X3)
```

```
X1:
 [[7 8]
 [9 4]
 [0 2]
 [0 7]]
X2:
 [[6 9 6 1]
 [7 2 4 9]]
X3:
 [[98 79 74 79]
 [82 89 70 45]
 [14  4  8 18]
 [49 14 28 63]]
```

O submódulo `numpy.linalg` oferece várias operações matriciais, como a inversão da matriz (`np.linalg.inv(array)`) e o cálculo do determinante (`np.linalg.det(array)`).

[46]:
```
print(np.linalg.inv(X3))
print(np.linalg.det(X3))
```

```
[[ 1.22583439e+14 -9.53426749e+13  3.96446149e+13 -9.
↪69408634e+13]
 [-4.03602217e+13  3.13912835e+13  1.09115720e+15 -2.
↪83571108e+14]
 [-5.14592826e+13  4.00238865e+13 -1.76129431e+15  5.
↪39166763e+14]
 [-6.35029445e+13  4.93911791e+13  5.09483394e+14 -1.
↪01215421e+14]]
-2.3271643223052853e-26
```

É possível, também, solucionar sistemas pela forma matricial, sendo necessário apenas o conhecimento da matriz de coeficientes e o vetor resposta. Vamos utilizar o exemplo abaixo:

$$\begin{bmatrix} 5 & 2 & 1 \\ 2 & 5 & 2 \\ 3 & -2 & 4 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 63 \\ 18 \\ 11 \end{bmatrix} \tag{3.1}$$

Utilizando a função `np.linalg.solve(A, b)`, é possível solucionar o sistema.

[53]:
```
A = np.array([[5, 2, 18],
              [2, 5, 2],
              [3,-2, 4]])

b = np.array([63, 18, 11])

x = np.linalg.solve(A, b)
print(x)
```

```
[1. 2. 3.]
```

3.3 SCIPY

A Scipy é um módulo eficiente para a realização de procedimentos numéricos, como integração numérica, interpolação, otimização, problemas da algebra linear e problemas de estatística. O módulo pode ser importando por `import scipy as sc`.

[55]:
```
import scipy as sc
import scipy.optimize as opt
```

```
import scipy.linalg as lin
```

3.3.1 Determinação das raízes de uma função.

Para determinar a raiz de uma função pelo método de Newton-Raphson, utiliza-se a função `sc.optimize.newton()`.

[64]:
```
def func(x, a, b, c):
    return a*x**2 + b*x + c
res = opt.newton(func, args=[-1, 2, 3], x0=0)
print(res)
```

```
-1.0000000000000004
```

Uma ferramenta bastante útil deste módulo, é a `scipy.linalg.eig(array)`, que retorna os autovalores e autovetores de uma dada matriz.

[67]:
```
w, phi = lin.eig(X3)

iw = w.argsort()                        # Garantir que os␣
↪autovetores e autovalores estejam em ordem crescente
w  = w[iw]
phi = phi[:, iw]
print(phi)
print(w)
```

```
[[-0.78895194 -0.38560276 -0.12196812 -0.70612365]
 [ 0.37413673 -0.31857084 -0.7338288  -0.63176069]
 [ 0.15047878  0.86580853  0.18359493 -0.087854  ]
 [ 0.46360833 -0.01409701  0.64258227 -0.30748901]]
[-1.45112789e-14+0.j -6.96947394e-17+0.j  4.57113453e+01+0.j
  2.12288655e+02+0.j]
```

[ ]:
```

```

4 MÓDULOS BÁSICOS- PARTE 2

4.1 O MÓDULO PANDAS

O módulo Pandas é fundamental para a realização de uma análise de dados. Com ele, é possível a importação de arquivos de dados (como planilhas eletrônicas e arquivos *csv*) e a criação de **Series** e **DataFrames**. As Series são sequências de dados unidimensionais que possuem índice (Uma espécie de dicionário) e os Dataframes são estruturas de dados bidimensionais, como planilhas. Nesta aula, iremos focar nos objetos de DataFrame.

[1]:
```
import pandas as pd
```

4.1.1 Importação de dados

Para realizar a importação de arquivos, utiliza-se a função `read`. Com ela, é possível a criação de um objeto DataFrame a partir de algum formato de arquivo escolhido pelo usuário. Abaixo, realizaremos a importação de uma planilha de excel.

[2]:
```
data = pd.read_excel('Resources/Arquivo_teste.
↪xlsx',index_col = 0)
```

Os dois argumentos indispensáveis para a importação do arquivo são: o caminho para o arquivo e a coluna que deve ser usada como índice. Para a visualização dos dados importados, basta escrever o nome da váriavel que armazena o DataFrame em uma célula de código.

[3]:
```
data
```

[3]:
```
          Coluna 1  Coluna 2  Coluna 3  Coluna 4  Coluna 5 ␣
↪Coluna 6  \
Linha 1          3         8        13        18        23 ␣
↪      28
Linha 2         22        27        32        37        42 ␣
↪      47
Linha 3         23        28        33        38        43 ␣
↪      48
```

```
Linha 4          24         29         34         39         44 ␣
↪     49
Linha 5          25         30         35         40         45 ␣
↪     50
...             ...        ...        ...        ...        ... ␣
↪    ...
Linha 84        104        109        114        119        124 ␣
↪    129
Linha 85        105        110        115        120        125 ␣
↪    130
Linha 86        106        111        116        121        126 ␣
↪    131
Linha 87        107        112        117        122        127 ␣
↪    132
Linha 88        108        113        118        123        128 ␣
↪    133

          Coluna 7  Coluna 8  Coluna 9  Coluna 10  ... ␣
↪Coluna 26  Coluna 27  \
Linha 1          33         38         43         48  ...      ␣
↪ 128        133
Linha 2          52         57         62         67  ...      ␣
↪ 147        152
Linha 3          53         58         63         68  ...      ␣
↪ 148        153
Linha 4          54         59         64         69  ...      ␣
↪ 149        154
Linha 5          55         60         65         70  ...      ␣
↪ 150        155
...             ...        ...        ...        ...  ...      ␣
↪ ...        ...
Linha 84         134        139        144        149  ...      ␣
↪ 229        234
Linha 85         135        140        145        150  ...      ␣
↪ 230        235
Linha 86         136        141        146        151  ...      ␣
↪ 231        236
```

```
Linha 87       137       142       147       152  ...
↪ 232       237
Linha 88       138       143       148       153  ...
↪ 233       238

          Coluna 28  Coluna 29  Coluna 30  Coluna 31  Coluna
↪32  Coluna 33  \
Linha 1         138       143       148       153
↪158       163
Linha 2         157       162       167       172
↪177       182
Linha 3         158       163       168       173
↪178       183
Linha 4         159       164       169       174
↪179       184
Linha 5         160       165       170       175
↪180       185
...             ...       ...       ...       ...
↪...       ...
Linha 84        239       244       249       254
↪259       264
Linha 85        240       245       250       255
↪260       265
Linha 86        241       246       251       256
↪261       266
Linha 87        242       247       252       257
↪262       267
Linha 88        243       248       253       258
↪263       268

          Coluna 34  Coluna 35
Linha 1         168       173
Linha 2         187       192
Linha 3         188       193
Linha 4         189       194
Linha 5         190       195
...             ...       ...
```

```
Linha 84         269         274
Linha 85         270         275
Linha 86         271         276
Linha 87         272         277
Linha 88         273         278

[88 rows x 35 columns]
```

4.1.2 Manipulação de dados

O objeto DataFrame possui uma vasta variedade de métodos que podem ser utilizada para a manipulção dos dados. A seguir, veremos alguns deles.

O método `.loc()` permite ao usuário selecionar uma determinada parte do seu DataFrame.

[4]:
```
cols = ['Coluna 1','Coluna 17','Coluna 32']
lins = ['Linha 20','Linha 42','Linha 43']
reduc = data.loc[lins,cols]
reduc
```

[4]:
```
          Coluna 1  Coluna 17  Coluna 32
Linha 20        40        120        195
Linha 42        62        142        217
Linha 43        63        143        218
```

Você pode, também, usar o método `sort_values()` para ordenar os valores conforme a sua vontade.

[5]:
```
reduc.sort_values(by='Coluna 1')
```

[5]:
```
          Coluna 1  Coluna 17  Coluna 32
Linha 20        40        120        195
Linha 42        62        142        217
Linha 43        63        143        218
```

Esta operação não altera o nosso DataFrame original.

Utilizando os métodos `.mean()` e `.std()` você também consegue estrair a média e o desvio padrão das colunas do seu DataFrame.

```
[6]: reduc.mean()
```

```
[6]: Coluna 1      55.0
     Coluna 17    135.0
     Coluna 32    210.0
     dtype: float64
```

```
[7]: reduc.std()
```

```
[7]: Coluna 1     13.0
     Coluna 17    13.0
     Coluna 32    13.0
     dtype: float64
```

Para uma análise completa do DataFrame, também é possível utilizar o método `.describe()`.

```
[8]: reduc.describe()
```

```
[8]:         Coluna 1  Coluna 17  Coluna 32
     count        3.0        3.0        3.0
     mean        55.0      135.0      210.0
     std         13.0       13.0       13.0
     min         40.0      120.0      195.0
     25%         51.0      131.0      206.0
     50%         62.0      142.0      217.0
     75%         62.5      142.5      217.5
     max         63.0      143.0      218.0
```

4.1.3 Integração Pandas e Numpy

É possível transformar dados do seu DataFrame em um ndarray, tornando possível a utilização de todos os seus métodos e atributos.

```
[9]: matriz = data.values
     print(matriz)
     print(type(matriz))
```

```
[[  3   8  13 ... 163 168 173]
 [ 22  27  32 ... 182 187 192]
 [ 23  28  33 ... 183 188 193]
 ...
 [106 111 116 ... 266 271 276]
 [107 112 117 ... 267 272 277]
 [108 113 118 ... 268 273 278]]
<class 'numpy.ndarray'>
```

4.1.4 Criação e exportação de um DataFrame

Neste tópico será mostrado como se cria um DataFrame e como deve-se fazer a sua exportação.

[10]:
```
import numpy as np
colunas = np.linspace(0,100,101)
linhas = np.linspace(0,60,61)
x,y = np.meshgrid(linhas,colunas)
z = x**2 -y
dados = pd.DataFrame(data = z.T, index = linhas, columns =␣
↪colunas)
dados
```

[10]:
```
        0.0     1.0     2.0     3.0     4.0     5.0     6.0  ␣
↪  7.0     8.0    \
0.0      0.0    -1.0    -2.0    -3.0    -4.0    -5.0    -6.0␣
↪   -7.0    -8.0
1.0      1.0     0.0    -1.0    -2.0    -3.0    -4.0    -5.0␣
↪   -6.0    -7.0
2.0      4.0     3.0     2.0     1.0     0.0    -1.0    -2.0␣
↪   -3.0    -4.0
3.0      9.0     8.0     7.0     6.0     5.0     4.0     3.0␣
↪    2.0     1.0
4.0     16.0    15.0    14.0    13.0    12.0    11.0    10.0␣
↪    9.0     8.0
...      ...     ...     ...     ...     ...     ...     ...␣
↪    ...     ...
56.0  3136.0  3135.0  3134.0  3133.0  3132.0  3131.0  3130.0␣
↪ 3129.0  3128.0
```

```
57.0  3249.0  3248.0  3247.0  3246.0  3245.0  3244.0  3243.0␣
↪ 3242.0  3241.0
58.0  3364.0  3363.0  3362.0  3361.0  3360.0  3359.0  3358.0␣
↪ 3357.0  3356.0
59.0  3481.0  3480.0  3479.0  3478.0  3477.0  3476.0  3475.0␣
↪ 3474.0  3473.0
60.0  3600.0  3599.0  3598.0  3597.0  3596.0  3595.0  3594.0␣
↪ 3593.0  3592.0

       9.0    ...   91.0    92.0    93.0    94.0    95.0    ␣
↪96.0    97.0   \
0.0     -9.0  ...   -91.0   -92.0   -93.0   -94.0   -95.0   ␣
↪-96.0   -97.0
1.0     -8.0  ...   -90.0   -91.0   -92.0   -93.0   -94.0   ␣
↪-95.0   -96.0
2.0     -5.0  ...   -87.0   -88.0   -89.0   -90.0   -91.0   ␣
↪-92.0   -93.0
3.0      0.0  ...   -82.0   -83.0   -84.0   -85.0   -86.0   ␣
↪-87.0   -88.0
4.0      7.0  ...   -75.0   -76.0   -77.0   -78.0   -79.0   ␣
↪-80.0   -81.0
...      ...  ...     ...     ...     ...     ...     ...    ␣
↪ ...     ...
56.0  3127.0  ...  3045.0  3044.0  3043.0  3042.0  3041.0 ␣
↪3040.0  3039.0
57.0  3240.0  ...  3158.0  3157.0  3156.0  3155.0  3154.0 ␣
↪3153.0  3152.0
58.0  3355.0  ...  3273.0  3272.0  3271.0  3270.0  3269.0 ␣
↪3268.0  3267.0
59.0  3472.0  ...  3390.0  3389.0  3388.0  3387.0  3386.0 ␣
↪3385.0  3384.0
60.0  3591.0  ...  3509.0  3508.0  3507.0  3506.0  3505.0 ␣
↪3504.0  3503.0

       98.0    99.0    100.0
0.0    -98.0   -99.0  -100.0
1.0    -97.0   -98.0   -99.0
```

```
2.0    -94.0   -95.0   -96.0
3.0    -89.0   -90.0   -91.0
4.0    -82.0   -83.0   -84.0
...      ...     ...     ...
56.0  3038.0  3037.0  3036.0
57.0  3151.0  3150.0  3149.0
58.0  3266.0  3265.0  3264.0
59.0  3383.0  3382.0  3381.0
60.0  3502.0  3501.0  3500.0

[61 rows x 101 columns]
```

Para realizar a exportação, utilizamos o método `to_excel()`.

[11]:
```
dados.to_excel('Resources/planilha_python.xlsx')
```

4.2 O MÓDULO MATPLOTLIB.PYPLOT

O matplotlib.pyplot é um módulo que permite a plotagem de gráficos no estilo Matlab. Este módulo é compatível com objetos do módulo Pandas e do módulo Numpy. Abaixo será ilustrada uma "receita de bolo" para a plotagem de gráficos.

[12]:
```
import matplotlib.pyplot as plt
```

[21]:
```
#-- Calculando os eixos X e Y.--#

t = np.linspace(0,50,204)                    # Criando o␣
↪eixo do tempo[s]
A = 2                                        # Amplitude do␣
↪movimento[m]
w = 0.5                                      # Frequência de␣
↪oscilação[rad/s]
phi = -np.pi/2                               # Ângulo de␣
↪fase[rad]

X = A*np.cos(phi+w*t)                        # Equação da␣
↪elongação (MHS)
```

[22]:
```
#-- Preparando a plotagem --#

plt.figure(1,figsize=(8,4))                    # Criando a
 ↪figura e definindo seu número e tamanho em X e Y
plt.plot(t,X,'b',linewidth = 4)                                #
 ↪Plotando os valores de X e Y e definindo a cor do gráfico
plt.title('Movimento harmônico simples')     # título do
 ↪gráfico
plt.xlabel('tempo(s)')                         # Nome do eixo X
plt.ylabel('Elongação(m)')                     # Nome do eixo y
plt.xlim(0,50) ; plt.ylim(-3,3)                # Definindo os
 ↪limites de cada eixo
plt.grid(True)                                 # Plotando a
 ↪grade
```

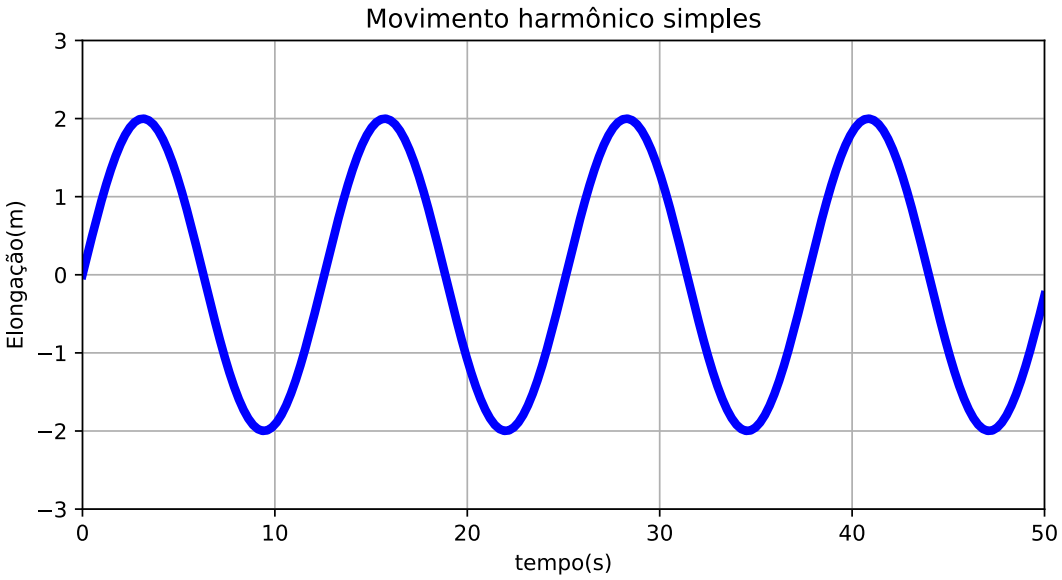


Figura 5 – Plotagem de figura utilizando o matplotlib

[24]:
```
plt.figure(2,figsize=(16,4))                    # Criando a
 ↪figura e definindo seu número e tamanho em X e Y

#plt.subplot(No de linhas, No de colunas, No da plotagem )
plt.subplot(1,2,1)
```

```
plt.plot(t,X,'g.')                                  # Plotando os␣
 ↪valores de X e Y e definindo a cor do gráfico
plt.title('Movimento harmônico simples')            # título do␣
 ↪gráfico
plt.xlabel('tempo(s)')                              # Nome do eixo X
plt.ylabel('Elongação(m)')                          # Nome do eixo y
plt.xlim(0,50) ; plt.ylim(-2,2)                     # Definindo os␣
 ↪limites de cada eixo
plt.grid(True)                                      # Plotando a␣
 ↪grade


plt.subplot(1,2,2)
plt.plot(t,X,'m:',linewidth = 3)                                  #␣
 ↪Plotando os valores de X e Y e definindo a cor do gráfico
plt.title('Movimento harmônico simples')            # título do␣
 ↪gráfico
plt.xscale('log')                                      # Mudando a␣
 ↪escala do gráfico
plt.xlabel('tempo(s)')                              # Nome do eixo X
plt.ylabel('Elongação(m)')                          # Nome do eixo y
plt.xlim(0,50) ; plt.ylim(-2,2)                     # Definindo os␣
 ↪limites de cada eixo
plt.grid(True)                                      # Plotando a␣
 ↪grade
```

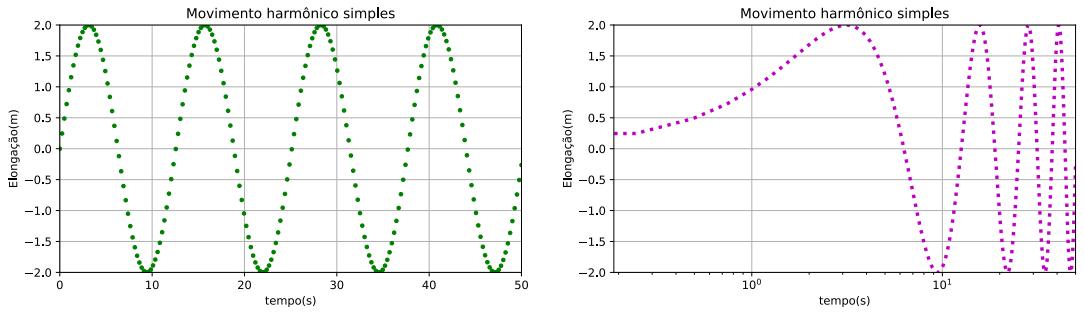


Figura 6 – Exemplo de subplot

Você também pode plotar gráficos do tipo scatter, para a plotagem de uma nuvem de

pontos.

[42]:
```
#-- Preparando a plotagem --#

plt.figure(1,figsize=(8,4))                    # Criando a
↪figura e definindo seu número e tamanho em X e Y
rand = np.random.randn(len(t))
plt.scatter(t,rand,color ='b',marker ='2')
↪          # Plotando os valores de X e Y e definindo a cor
↪do gráfico
plt.title('Movimento harmônico simples')       # título do
↪gráfico
plt.xlabel('tempo(s)')                         # Nome do eixo X
plt.ylabel('Elongação(m)')                     # Nome do eixo y
plt.xlim(0,50) ; plt.ylim(-max(rand),max(rand))
↪# Definindo os limites de cada eixo
plt.grid(True)                                 # Plotando a
↪grade
```

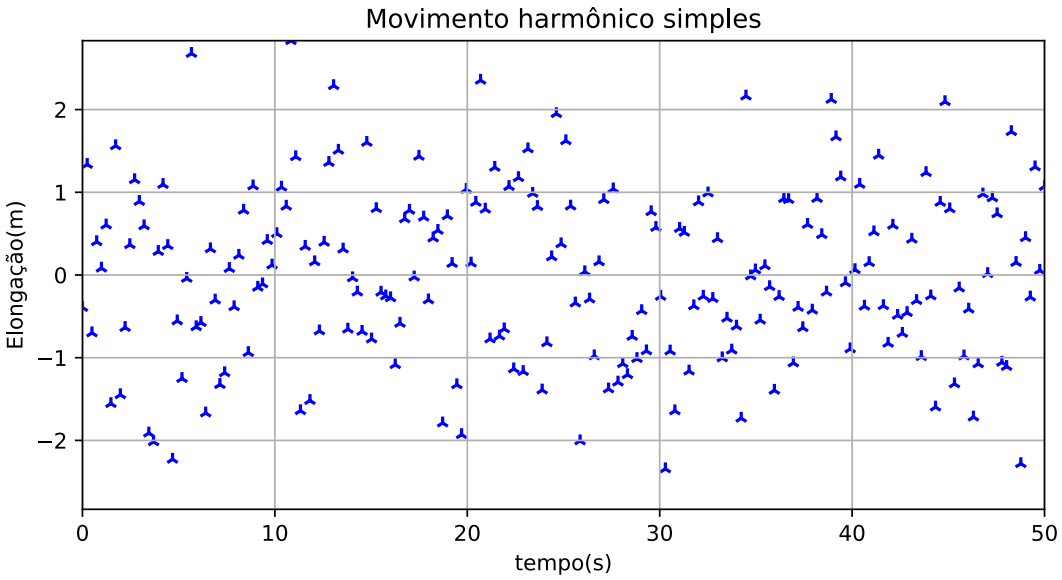


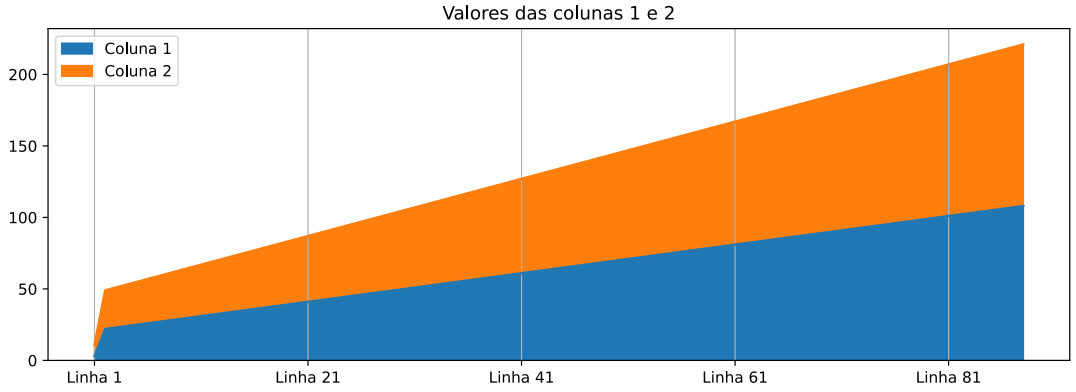
Figura 7 – Exemplo de scatter plot

Para visualizar mais opções de estilos de plotagem, basta clicar aqui

4.2.1 Integração matplotlib e Pandas

Os objetos do módulo pandas possuem um método de plotagem. Abaixo, veremos algumas possibilidades.

[85]:
```
data[['Coluna 1','Coluna 2']].plot.area(figsize=(12,4));plt.
↪grid(axis ='x')
plt.title('Valores das colunas 1 e 2')
data[['Coluna 1','Coluna 2']].plot.hist();plt.grid(axis ='y')
plt.savefig('Resources/salvando_figura.png') #comando para␣
↪salvar a figura no diretório
```



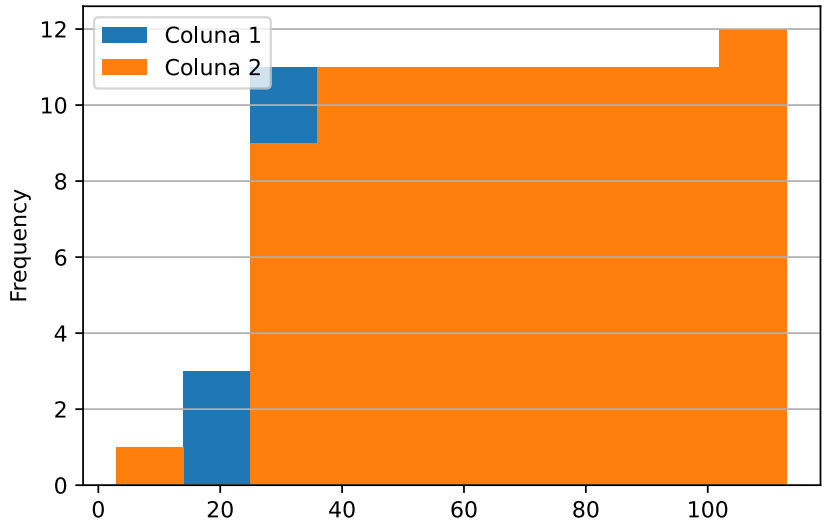


Figura 8 – Exemplo de integração Pandas-Matplotlib

4.2.2 O que é possível fazer com o matplotlib?

[18]:
```
from scipy.interpolate import interp2d
x = np.linspace(0, 4, 13)
y = np.array([0, 2, 3, 3.5, 3.75, 3.875, 3.9375, 4])
X, Y = np.meshgrid(x, y)
Z = np.sin(np.pi*X/2) * np.exp(Y/2)

x2 = np.linspace(0, 4, 65)
y2 = np.linspace(0, 4, 65)
f = interp2d(x, y, Z, kind='cubic')
Z2 = f(x2, y2)

plt.figure(6,figsize=(8,6))
plt.subplot(1,2,1)
plt.pcolormesh(X, Y, Z)

X2, Y2 = np.meshgrid(x2, y2)
plt.subplot(1,2,2)
plt.pcolormesh(X2, Y2, Z2)

plt.show()
```

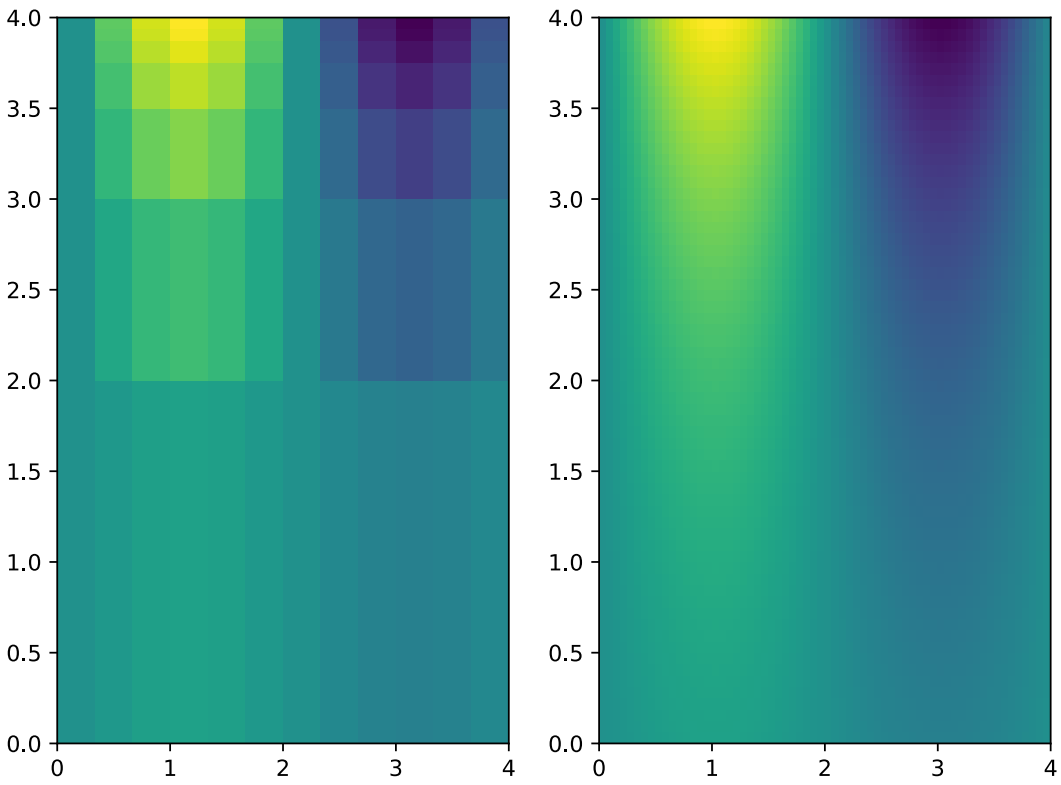


Figura 9 – Plotagem de um *color map* utilizando o matplotlib

[19]:
```
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
plt.figure(7,figsize = (12,12))
x = plt.subplot(211,projection ='3d')
y = plt.subplot(212,projection ='3d')

x.plot_surface(X,Y,Z,cmap = cm.ocean)
y.plot_surface(X2,Y2,Z2,cmap = cm.ocean)
```

[19]:
```
<mpl_toolkits.mplot3d.art3d.Poly3DCollection at␣
↪0x17d26ff8fa0>
```

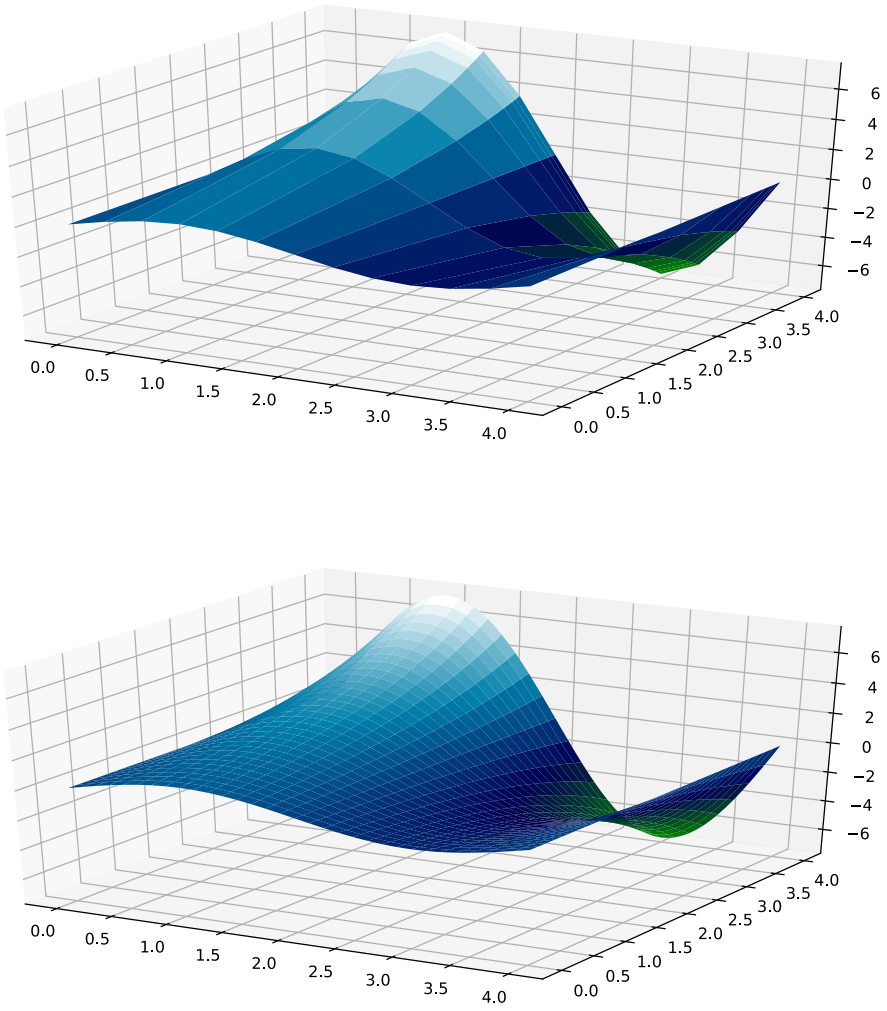


Figura 10 – Plotagem de superfície utilizando o matplotlib

[ ]:

5 PROGRAMA PARA RESOLUÇÃO DE TRELIÇAS PLANAS

5.1 FUNDAMENTAÇÃO TEÓRICA.

Nesta sessão serão apresentados todos os conceitos necessários de análise matricial de estruras para o compreendimento e realização da aplicação prática.

5.1.1 Matriz de rigidez de uma barra de treliça plana.

A análise de estruturas por elementos de barra é o princípio do ** método dos elmentos finitos**, sendo a barra de treliça plana o elemento finito mais simples, que sofre apenas carregamentos axiais em torno dos seus eixos locais, como mostrado na figura abaixo.

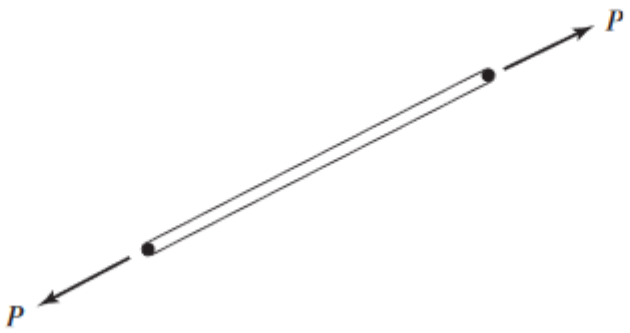


Figura 11 – Barra de treliça no sistema local de coordenadas

Num sistema local, a barra de treliça plana possui apenas um grau de liberdade (gdl) por nó, tendo sua matriz de rigidez expressa por:

$$K_l^l = \frac{EA}{L}\begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} \tag{5.1}$$

A partir do momento em que se deseja realizar uma análise global da estrutura, é necessário reescrever a matriz de rigidez local de forma que essa possua um grau de liberdade por eixo do espaço adotado. Isso ocorre porque devem existir graus de liberdade suficientes para a correta decomposição dos deslocamentos de todos os nós, indiferentemente da orientação das barras. Dessa forma, em uma treliça plana devem haver 2 graus de liberdade, o gdl de translação horizontal (x) e o gdl de translação vertical (y), conforme a figura a seguir:

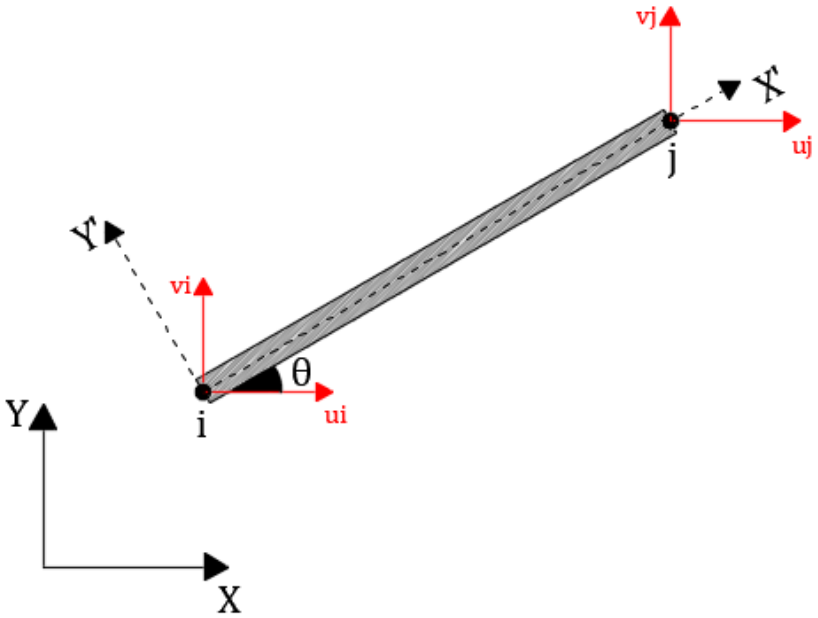


Figura 12 – Barra de treliça no sistema global de coordenadas

Adotando o eixo x' (local) sobre o eixo da barra (conforme figura anterior), a matriz local, considerando o grau de liberdade adicional y' perpendicular ao eixo x', tem a seguinte configuração, que não altera suas propriedades.

$$K_l^l = \frac{EA}{L}\begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{5.2}$$

Perceba que, para transformar as coordenadas globais em coordendadas locais, podem-se usar as seguintes equações:

$$u_{i'} = u_i \cos\theta + v_i \sin\theta \tag{5.3}$$
$$v_{i'} = -u_i \sin\theta + v_i \cos\theta \tag{5.4}$$
$$u_{j'} = u_j \cos\theta + v_j \sin\theta \tag{5.5}$$
$$v_{j'} = -u_j \sin\theta + v_j \cos\theta \tag{5.6}$$

Colocando o sistema de equações acima em forma matricial, tem-se o exposto abaixo:

$$\begin{bmatrix} u_{i'} \\ v_{i'} \\ u_{j'} \\ v_{j'} \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & cos\theta & 0 & 0 \\ 0 & 0 & \cos\theta & \sin\theta \\ 0 & 0 & -\sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} u_i \\ v_i \\ u_j \\ v_j \end{bmatrix} \tag{5.7}$$

Ou ainda:

$$U^l = \Theta U^G \tag{5.8}$$

Em que Θ é chamada de matriz de rotação.

$$\Theta = \begin{bmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & cos\theta & 0 & 0 \\ 0 & 0 & \cos\theta & \sin\theta \\ 0 & 0 & -\sin\theta & \cos\theta \end{bmatrix} \tag{5.9}$$

Escrevendo, agora, as equações de equilíbrio local (de uma barra de treliça plana) e equilibrio global (de toda estrutura), temos:

$$K_l^L U^l = F^l \tag{5.10}$$

$$K^G U^G = F^G \tag{5.11}$$

Sabendo que o vetor de forças globais F_G pode ser escrito como

$$F^G = \Theta^T F^l \tag{5.12}$$

E utilizando a relação entre deslocamentos locais e globais,é possível reescrever a equação de equilíbrio multiplicando ambos os lados pelo transposto da matriz de rotação:

$$\Theta^T K_l^L \Theta U^G = \Theta^T F^l \tag{5.13}$$

$$K_l^G U^G = F^G \tag{5.14}$$

Sendo possível afirmar que a matriz de rigidez em coordendas globais pode ser expressa por:

$$K_l^G = \Theta^T K_l^L \Theta \tag{5.15}$$

5.1.2 Montagem da matriz de rigidez global da estrutura

A matriz de rigidez total da estrutura é uma matriz que comtempla todos os graus de liberdade das barras utilizadas, sendo necessário o correto acoplamento dos mesmos. Abaixo será realizado um exemplo de como deve ser realizado este procedimento.

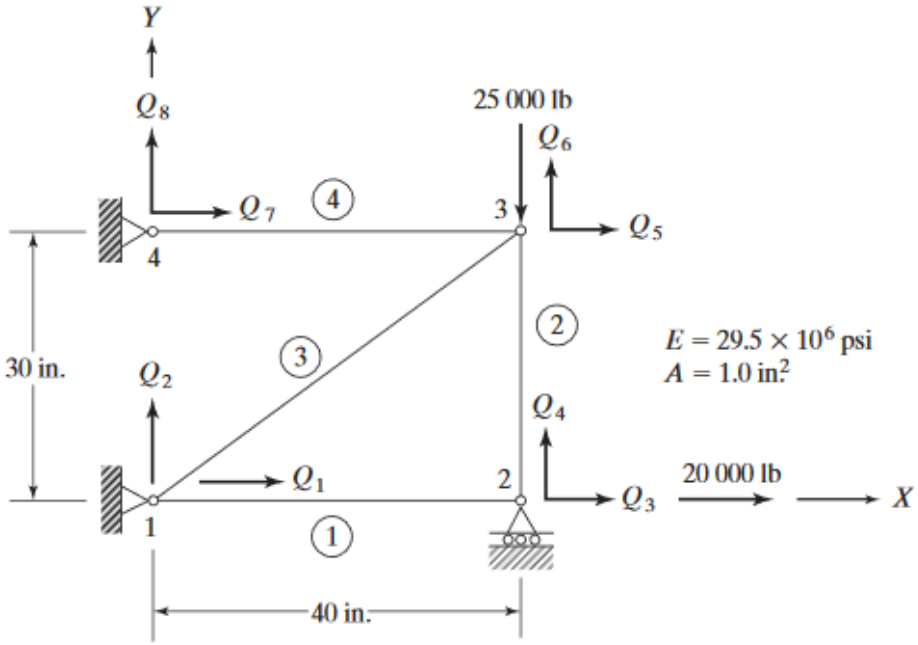


Figura 13 – Exemplo de estrutura com barras de treliça plana

Neste exemplo é possível verificar a presença de 4 barras de treliça plana, 4 nós e 8 gdl (2 por nó). A numeração dos graus de liberdade é feita nó a nó, onde a direção x tem o primeiro gdl do nó e a direção y o segundo, dessa forma, a numeração dos gdls para o nó n é dada por:

$$gdl_h = 2n - 1 \tag{5.16}$$

$$gdl_v = 2n \tag{5.17}$$

Montando as matrizes de rigidez locais (já rotacionadas) de cada um dos elementos temos:

$$K_1 = \frac{29.5 * 10^6}{40} \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix}$$

$$K_2 = \frac{29.5 * 10^6}{30} \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 \\ -1 & 0 & 1 & 0 \\ 0 & -1 & 0 & 1 \end{bmatrix} \begin{matrix} 5 \\ 6 \\ 3 \\ 4 \end{matrix}$$

$$K_3 = \frac{29.5 * 10^6}{50} \begin{bmatrix} .64 & .48 & -.64 & -.48 \\ .48 & .36 & -.48 & -.36 \\ .48 & -.36 & .48 & .36 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{matrix} 1 \\ 2 \\ 5 \\ 6 \end{matrix}$$

$$K_4 = \frac{29.5 * 10^6}{40} \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{matrix} 7 \\ 8 \\ 5 \\ 6 \end{matrix}$$

Perceba que cada linha e coluna estão numeradas conforme o seu grau de liberdade correspondente.

As matrizes locais serão agora alocadas na matriz global(8x8) conforme os seus graus de liberdade. Caso haja a superposição de valores, esses devem ser somados.

$$K = \frac{29.5 * 10^6}{600} \begin{bmatrix} 22.68 & 5.76 & -15 & 0 & -7.68 & -5.76 & 0 & 0 \\ 5.76 & 4.32 & 0 & 0 & -5.76 & -4.32 & 0 & 0 \\ -15 & 0 & 15 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 20 & 0 & -20 & 0 & 0 \\ -7.68 & -5.76 & 0 & 0 & 22.68 & 5.76 & -15 & 0 \\ -5.76 & -4.32 & 0 & -20 & 5.76 & 24.32 & 0 & 0 \\ 0 & 0 & 0 & 0 & -15 & 0 & 15 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{matrix}$$

De forma genérica, podemos acoplar as matrizes locais na matriz global na forma de quadrantes, ao invés de acopla-las termo a termo, visto que cada nó (quadrante) tem 2 gdls em sequencia.

$$K_l^G = \begin{bmatrix} k_{ii} & k_{ij} \\ k_{ji} & k_{jj} \end{bmatrix} \tag{5.18}$$

$$K = \begin{bmatrix} k_{ii} & \dots & k_{ij} \\ \vdots & \ddots & \vdots \\ k_{ji} & \dots & k_{jj} \end{bmatrix} \tag{5.19}$$

5.1.3 Montagem do vetor de forças externas

O vetor de forças externas deve possuir dimensões n x 1, em que n é a quantidade de graus de liberdade. Dessa forma a posição de um valor no vetor corresponde ao seu gdl (as linhas com numeração ímpar correspondem aos gdl horizontais, enquanto as linhas com numeração par estão relacionadas aos gdl verticais). Montando o vetor de forças para o exemplo acima, temos:

$$F^G = \begin{bmatrix} 0 \\ 0 \\ 20000 \\ 0 \\ 0 \\ -25000 \\ 0 \\ 0 \end{bmatrix} \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{matrix} \tag{5.20}$$

5.1.4 Aplicação das condições de contorno

Para a aplicação das condições de contorno existem duas possibilidades: a remoção das linhas e as colunas dos graus de liberdade restritos ou a substituição dessas linhas e colunas por zeros, mantendo 1 na diagonal principal. A primeira metodologia apresenta vantagens em relação ao processamento da estrutura, reduzindo as dimensões da matriz, enquanto a implementação da segunda é de maior facilidade, sendo essa aqui abordada.

$$K_R = \frac{29.5 * 10^6}{600} \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 15 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 22.68 & 5.76 & 0 & 0 \\ 0 & 0 & 0 & 0 & 5.76 & 24.32 & 0 & \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{matrix} \tag{5.21}$$

5.1.5 Cálculo dos deslocamentos

Para a realização do cálculo dos deslocamentos, basta resolver o sistema de equações:

$$K_R U = F^G \tag{5.22}$$

5.1.6 Determinação das reações de apoio

Agora em posse dos deslocamentos (U), ao realizar o produto da matriz de rigidez sem restrições pelo vetor deslocamento (KU), o vetor de forças resultante(F^G) possuirá valores referentes aos graus de liberdade restritos, sendo esses os valores das reações de apoio.

5.1.7 Determinação dos esforços em cada barra

Para a determinação dos esforços atuantes em cada barra, devemos resolver a equação de equilíbrio local de cada barra. Para isso, será necessário a utilização da matriz de rigidez local da barra (K_l^l) e do deslocamento de cada nó da barra em coordenadas locais (U^l). Para realizar a conversão do deslocamento global para o local, utilizaremos novamente a expressão $U^l = \Theta U^G$. Portanto, para o cálculo dos esforços, basta resolver o sistema abaixo:

$$K_l^l U^l = F^l \tag{5.23}$$

5.2 INTRODUÇÃO

A análise matricial de treliças requer a criação de arrays que nos permitem modelar a estrutura de forma matemática. São necessárias:

- array de coordendas nodais: coordenadas `X` e `Y`;
- array de conectividade: contendo os nós que delimitam cada barra;
- array de propriedades: contendo a área e o módulo de elasticidade longitudinal;
- array de forças externas : contendo os valores de forças horizontais e verticais;
- array de restrições : contendo os graus de liberdade que são restritos.

Para criação dessas arrays os dados serão importados através do módulo `pandas` de planilha do Excel chamada `entrada.xlsx`. A planilha de entrada de dados deve ter duas abas, a primeira chamada de `Nós` contendo as coordenadas de cada nó e a segunda chamada `Barras` com as conectividades (nó inicial e final) e as propriedades de cada barra.

5.2.1 Esturutra em estudo

A estrutura em estudo é uma treliça plana com 12 nós, 21 barras e 2 gdl por nó. As coordenadas dos nós e os elementos podem ser visualizadas na figura.

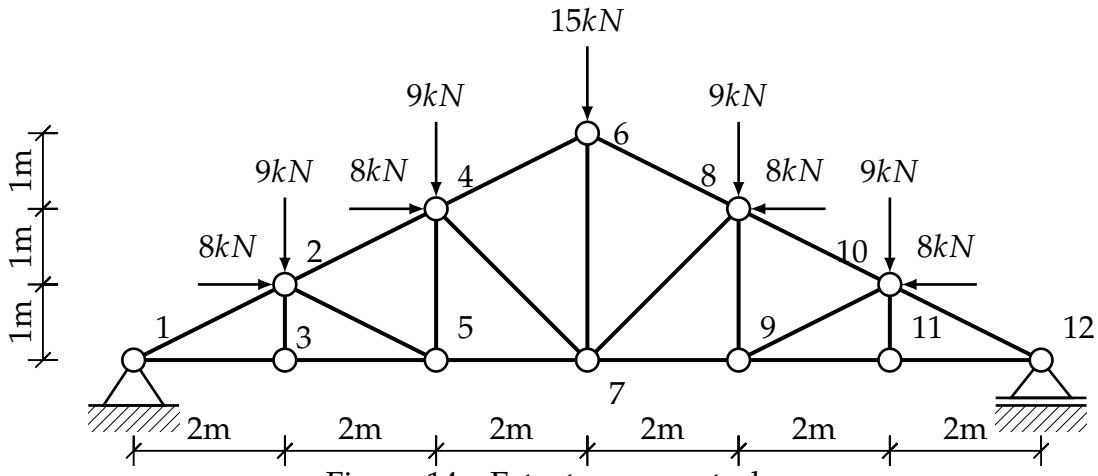


Figura 14 – Estrutura em estudo

Todas as barras possuem a mesma área de seção transversal $A = 0.01m^2$ e o mesmo módulo de elasticidade $E = 21 * 10^{10} \frac{N}{m}$.

Vamos, então, importar os módulos necessários:

[3]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

5.3 ENTRADA DE DADOS

Conforme apresentado anteriormente a entrada de dados é realizada através do arquivo `entrada.xlsx`. Vamos então importar cada aba separadamente especificando o `keyarg sheet_name`.

[4]:
```
nos = pd.read_excel('resources/entrada.xlsx',␣
↪sheet_name='Nós')
barras = pd.read_excel('resources/entrada.xlsx',␣
↪sheet_name='Barras')

# Acertando os números dos nós e das barras conforme a␣
↪figura (iniciando em 1)
nos.index += 1
barras.index += 1

# Trocando celulas vazias por zeros
nos.fillna(0, inplace=True)

# Printando os valores na tela
nos
```

[4]:

	X	Y	RX	RY	FX	FY
1	0	0	1.0	1.0	0.0	0.0
2	2	1	0.0	0.0	8000.0	-9000.0
3	2	0	0.0	0.0	0.0	0.0
4	4	2	0.0	0.0	8000.0	-9000.0
5	4	0	0.0	0.0	0.0	0.0
6	6	3	0.0	0.0	0.0	-15000.0
7	6	0	0.0	0.0	0.0	0.0
8	8	2	0.0	0.0	-8000.0	-9000.0
9	8	0	0.0	0.0	0.0	0.0
10	10	1	0.0	0.0	-8000.0	-9000.0
11	10	0	0.0	0.0	0.0	0.0
12	12	0	0.0	1.0	0.0	0.0

[5]:
```
barras
```

[5]:

	N1	N2	A	E
1	1	2	0.01	2100000000000
2	1	3	0.01	2100000000000
3	2	3	0.01	2100000000000
4	2	4	0.01	2100000000000
5	2	5	0.01	2100000000000
6	3	5	0.01	2100000000000
7	4	5	0.01	2100000000000
8	4	6	0.01	2100000000000
9	4	7	0.01	2100000000000
10	5	7	0.01	2100000000000
11	6	7	0.01	2100000000000
12	6	8	0.01	2100000000000
13	7	8	0.01	2100000000000
14	7	9	0.01	2100000000000
15	8	9	0.01	2100000000000
16	8	10	0.01	2100000000000
17	9	10	0.01	2100000000000
18	9	11	0.01	2100000000000
19	10	11	0.01	2100000000000
20	10	12	0.01	2100000000000
21	11	12	0.01	2100000000000

5.4 PLOTAGEM DA ESTRUTURA

O código abaixo permite a montagem da estrutura plotando elemento a elemento a partir das matrizes das barras e dos nós .

[6]:
```
plt.figure(1,figsize=(12,4))
plt.ylim(-1,5)

# Plotagem dos apoios e das forças
for no in nos.index:
    X,Y,RX,RY,Fx,Fy = nos.loc[no]

    # Se RX restrito aplica sobre gl de X
    if RX == 1:
```

```
        plt.scatter(X,Y,400,marker =5,zorder = -2,color␣
↪='gray')
    if RY == 1:
        plt.scatter(X,Y,400,marker =6,zorder = -2,color␣
↪='gray')

    if Fx >0:
        plt.arrow(X-1.5,Y,1,0,width =0.05,color='k')
        plt.text(X-1.5,Y,'{}kN'.format(Fx/1000),va='bottom')
    if Fx <0:
       plt.arrow(X+1.5,Y,-1,0,width =0.05,color='k')
       plt.text(X+.5,Y,'{}kN'.format(Fx/1000),va='bottom')
    if Fy >0:
        plt.arrow(X,Y-1.5,0,1,width =0.05,color='k')
        plt.text(X,Y,'{}kN'.format(Fy/
↪1000),va='bottom',rotation=90)
    if Fy <0:
       plt.arrow(X,Y+1.5,0,-1,width =0.05,color='k')
       plt.text(X,Y+.5,'{}kN'.format(Fy/
↪1000),ha='right',rotation=90)


# Plotagem das barras
for barra in barras.index:
    # Vamos passar os nós para as variáveis N1 e N2
    N1 = barras.loc[barra, 'N1']
    N2 = barras.loc[barra, 'N2']

    # Agora vamos acessar as coordendas de cada um dos nós
    x1, y1 = nos.loc[N1, ['X','Y']]
    x2, y2 = nos.loc[N2, ['X','Y']]
    y = [y1,y2]
    x = [x1,x2]

    plt.plot(x,y,'black')
    plt.scatter(x,y, s=80,marker ='o',color ='black')
    #plt.grid(True)
```

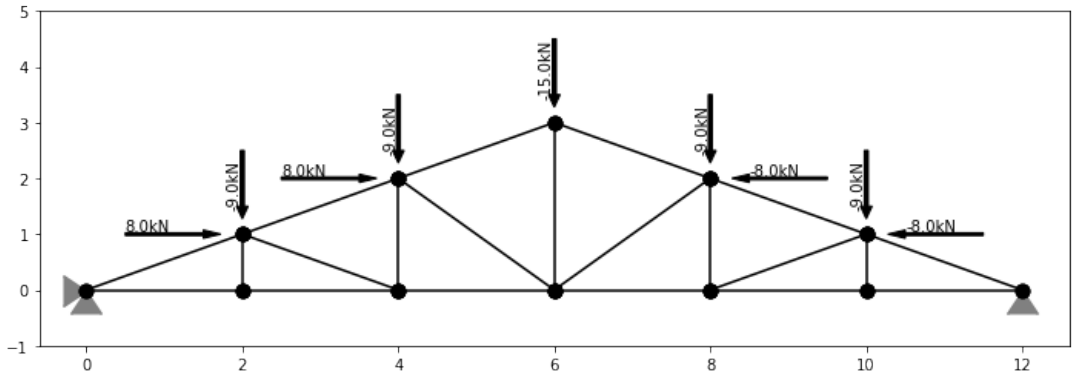


Figura 15 – Plotagem da estrutura em estudo

Ok! Podemos visualizar que os dados foram importados corretamente!

5.5 COMPRIMENTO DAS BARRAS E COSSENOS DIRETORES

Vamos, então, determinar os comprimentos das barras e seus cossenos diretores. Para isso vamos percorrer DataFrame de barras e a cada passo acessar o DataFrame de nós para obter suas coordendas. Os valores cálculados serão armazenados em listas que serão inseridas no DataFrame ao fim do processo.

[7]:
```
# Criação de listas vazias para armazenar as variáves
Ls   = []
sens = []
coss = []

for barra in barras.index:
    # Vamos passar os nós para as variáveis N1 e N2
    N1 = barras.loc[barra, 'N1']
    N2 = barras.loc[barra, 'N2']

    # Agora vamos acessar as coordendas de cada um dos nós
    x1, y1 = nos.loc[N1, ['X','Y']]
    x2, y2 = nos.loc[N2, ['X','Y']]

    # O comprimento da barra é dado pelo teorema de Pitagoras
    Lx = x2 - x1
    Ly = y2 - y1
```

```
    L = np.sqrt(Lx**2 + Ly**2)

    # Os cossenos diretores são então:
    sen = Ly/L
    cos = Lx/L

    # Inserindo nas listas
    Ls.append(L)
    sens.append(sen)
    coss.append(cos)

# Agora que saimos do loop vamos inserir no DataFrame
barras['L']   = Ls
barras['sen'] = sens
barras['cos'] = coss

# Printando novo DataFrame
barras
```

[7]:

	N1	N2	A	E	L	sen	cos
1	1	2	0.01	210000000000	2.236068	0.447214	0.894427
2	1	3	0.01	210000000000	2.000000	0.000000	1.000000
3	2	3	0.01	210000000000	1.000000	-1.000000	0.000000
4	2	4	0.01	210000000000	2.236068	0.447214	0.894427
5	2	5	0.01	210000000000	2.236068	-0.447214	0.894427
6	3	5	0.01	210000000000	2.000000	0.000000	1.000000
7	4	5	0.01	210000000000	2.000000	-1.000000	0.000000
8	4	6	0.01	210000000000	2.236068	0.447214	0.894427
9	4	7	0.01	210000000000	2.828427	-0.707107	0.707107
10	5	7	0.01	210000000000	2.000000	0.000000	1.000000
11	6	7	0.01	210000000000	3.000000	-1.000000	0.000000
12	6	8	0.01	210000000000	2.236068	-0.447214	0.894427
13	7	8	0.01	210000000000	2.828427	0.707107	0.707107
14	7	9	0.01	210000000000	2.000000	0.000000	1.000000
15	8	9	0.01	210000000000	2.000000	-1.000000	0.000000
16	8	10	0.01	210000000000	2.236068	-0.447214	0.894427
17	9	10	0.01	210000000000	2.236068	0.447214	0.894427
18	9	11	0.01	210000000000	2.000000	0.000000	1.000000

```
19   10   11   0.01   210000000000   1.000000 -1.000000   0.000000
20   10   12   0.01   210000000000   2.236068 -0.447214   0.894427
21   11   12   0.01   210000000000   2.000000  0.000000   1.000000
```

5.6 MONTAGEM DA MATRIZ DE RIGIDEZ

Em posse das propriedades das barras podemos agora montar as matrizes de rigidez locais e acoplalas na matriz de rigidez global. Por se tratar de barras de treliça plana cada nó possui 2 graus de liberdade (deslocamentos horizontal e vertical), portanto em treliças planas o números de graus de liberdade sempre será o dobro do número de nós.

Neste nosso exemplo teremos então 24 graus de liberdade, esses são numerados em função do número do nó conforme as expressões a seguir:

$$gdl_h = 2n - 1$$

$$gdl_v = 2n$$

Nossa matriz de rigidez global será então uma matriz de ordem `24x24`, em que os graus de liberdade ímpares correspondem aos deslocamentos horizontais e pares aos verticais.

A matriz de rigidez global é a obtida pela superposição das matrizes de rigidez locais nos respectivos graus de liberdade. Inicialmente a matriz global `K` deve ser pré alocada como uma matriz de zeros.

[8]:
```
maxgl = 2*len(nos)
K = np.zeros([maxgl,maxgl])
```

A alocação é realizada percorrendo todas as barras novamente, cálculando suas matrizes de rigidez local e alocando suas componentes na matriz global.

[9]:
```
for barra in barras.index:
    # Vamos importar as propriedades necessárias para␣
↪construção da matriz local e da matriz de rotação
    L   = barras.loc[barra, 'L']
    sen = barras.loc[barra, 'sen']
    cos = barras.loc[barra, 'cos']
    A   = barras.loc[barra, 'A']
    E   = barras.loc[barra, 'E']
```

```
    N1  = barras.loc[barra, 'N1']
    N2  = barras.loc[barra, 'N2']

    # Matriz de rigidez no sistema local
    Kl = E*A/L*np.array([[ 1, 0,-1, 0],
                         [ 0, 0, 0, 0],
                         [-1, 0, 1, 0],
                         [ 0, 0, 0, 0]])

    # Matriz de rotação
    Mrot = np.array([[ cos,  sen,    0,   0],
                     [-sen,  cos,    0,   0],
                     [    0,   0,  cos, sen],
                     [    0,   0, -sen, cos]])

    # Rotação da matriz de coordenadas locais para globais
    Klr = np.dot(np.dot(Mrot.T, Kl), Mrot)

    # Cálculo dos graus de liberdade
    gl1 = 2*N1 - 1
    gl2 = 2*N1
    gl3 = 2*N2 - 1
    gl4 = 2*N2

    # Aloca a matriz local na matriz global
    # Lembrando as propriedades das listas do Python!
    K[gl1-1:gl2, gl1-1:gl2] += Klr[0:2, 0:2]
    K[gl3-1:gl4, gl1-1:gl2] += Klr[2:4, 0:2]
    K[gl1-1:gl2, gl3-1:gl4] += Klr[0:2, 2:4]
    K[gl3-1:gl4, gl3-1:gl4] += Klr[2:4, 2:4]
```

5.6.1 Aplicação das condições de contorno

Agora, para solução do problema, a matriz de rigidez global é clonada e as restrições de apoio são impostas zerando as respectivas linhas e colunas.

[10]:
```
Kcompleta = K.copy()     # Alocando a matriz em outro espaço de memória

for no in nos.index:
    RX, RY = nos.loc[no, ['RX','RY']]
    # Se RX restrito aplica sobre gl de X
    if RX == 1:
        gl = 2*no - 1
        K[:,    gl-1] = 0
        K[gl-1,    :] = 0
        K[gl-1, gl-1] = 1
        print('Aplicando restrição horizontal no nó {:d}.'.format(no))

    if RY == 1:
        gl = 2*no
        K[:,    gl-1] = 0
        K[gl-1,    :] = 0
        K[gl-1, gl-1] = 1
        print('Aplicando restrição vertical no nó {:d}.'.format(no))
```

```
Aplicando restrição horizontal no nó 1.
Aplicando restrição vertical no nó 1.
Aplicando restrição vertical no nó 12.
```

5.7 MONTAGEM DO VETOR DE FORÇAS

Com a matriz global restringida vamos montar o vetor de forças aplicadas. Este vetor tem como dimensão o número de graus de liberdade.

[11]:
```
F = np.zeros(maxgl)
```

[12]:
```
for no in nos.index:
    FX, FY = nos.loc[no, ['FX','FY']]
    gl1 = 2*no - 1
    gl2 = 2*no
    F[gl1-1] = FX
```

```
    F[gl2-1] = FY
```

5.8 RESOLUÇÃO DA EQUAÇÃO DE EQUILÍBRIO

Resolvendo o sistema pelo `numpy.linalg.solve()` obtemos os deslocamentos nodais.

[13]:
```
U = np.linalg.solve(K, F)
```

5.9 PLOTAGEM DA TRELIÇA DEFORMADA

Em posse dos valores de deslocamento, é possível realizar a plotagem da treliça deslocada, junto com suas reações de apoio.

[14]:
```
e = 1000
for barra in barras.index:
    # Vamos passar os nós para as variáveis N1 e N2
    N1 = barras.loc[barra, 'N1']
    N2 = barras.loc[barra, 'N2']

    # Agora vamos acessar as coordendas de cada um dos nós
    x1, y1  = nos.loc[N1, ['X','Y']]
    x2, y2  = nos.loc[N2, ['X','Y']]

    dx = np.array([U[2*N1-2],U[2*N2-2]])
    dy = np.array([U[2*N1-1],U[2*N2-1]])
    y = [y1,y2]
    x = [x1,x2]
    plt.figure(1,figsize=(12,4))
    plt.plot(x,y,'g:')
    plt.plot(x+dx*e,y+dy*e,'black')
    plt.scatter(x+dx*e,y+dy*e, s=80,marker ='o',color␣
 ↪='black')
    #plt.grid(True)

_=plt.title('Treliça deformada')
```

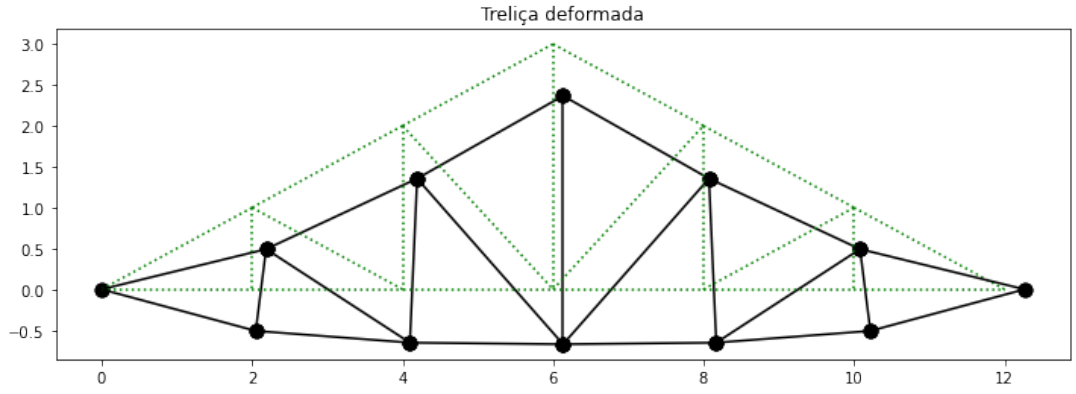


Figura 16 – Plotagem da treliça deformada

5.10 PLOTAGEM DAS REAÇÕES DE APOIO

Como comentado na fundamentação teórica, o cálculo das reações de apoio é feito a partir do produto da matriz de rigidez pelos deslocamentos.

[15]:
```
R = np.dot(Kcompleta,U)
print(R)
```

```
[ 7.27595761e-12  2.55000000e+04  8.00000000e+03 -9.
↪00000000e+03
  0.00000000e+00  0.00000000e+00  8.00000000e+03 -9.
↪00000000e+03
  5.82076609e-11  2.32830644e-10  5.82076609e-11 -1.
↪50000000e+04
  0.00000000e+00  0.00000000e+00 -8.00000000e+03 -9.
↪00000000e+03
  0.00000000e+00 -1.16415322e-10 -8.00000000e+03 -9.
↪00000000e+03
 -5.82076609e-11  0.00000000e+00 -5.82076609e-11  2.
↪55000000e+04]
```

Agora, para a realização da plotagem, é necessário escolher apenas os graus de liberdade que sãoo restritos.

[16]:
```
plt.figure(1,figsize=(12,4))
plt.xlim(-2,15)
plt.ylim(-1,5)

# Plotagem dos apoios e das forças
for no in nos.index:
    X,Y,RX,RY= nos.loc[no,['X','Y','RX','RY']]

    gl1 = 2*no - 1
    gl2 = 2*no
    Fx = R[gl1-1]
    Fy = R[gl2-1]

    # Se RX restrito aplica sobre gl de X
    if RX == 1:
        plt.arrow(X-1.5,Y,1,0,width =0.05,color='k')
        plt.text(X-1.5,Y,'{:.2f}kN'.format(Fx/
↪1000),va='bottom')
    if RY == 1:
        plt.arrow(X,Y,0,1,width =0.05,color='k')
        plt.text(X+0.2,Y+0.8,'{:.2f}kN'.format(Fy/
↪1000),va='bottom')

# Plotagem das barras
for barra in barras.index:
    # Vamos passar os nós para as variáveis N1 e N2
    N1 = barras.loc[barra, 'N1']
    N2 = barras.loc[barra, 'N2']

    # Agora vamos acessar as coordendas de cada um dos nós
    x1, y1 = nos.loc[N1, ['X','Y']]
    x2, y2 = nos.loc[N2, ['X','Y']]
    y = [y1,y2]
    x = [x1,x2]

    plt.plot(x,y,'black')
    plt.scatter(x,y, s=80,marker ='o',color ='black')
    #plt.grid(True)
```

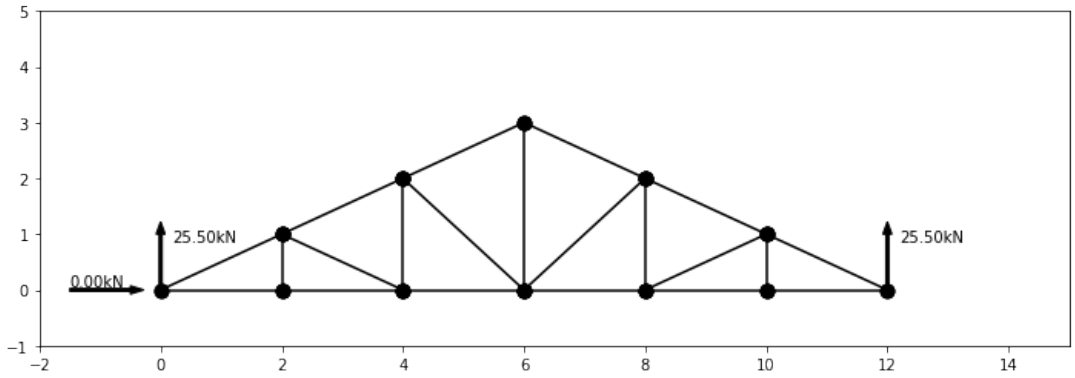


Figura 17 – Plotagem das reações de apoio

5.11 DETERMINAÇÃO DOS ESFORÇOS

Para finalizar a análise, é interessante o cálculo dos esforços atuantes em cada elemento, para isso, realiza-se o script abaixo:

[17]:
```
Esf = []

for barra in barras.index:
    # Vamos importar as propriedades necessárias para␣
↪construção da matriz local e da matriz de rotação
    L   = barras.loc[barra, 'L']
    sen = barras.loc[barra, 'sen']
    cos = barras.loc[barra, 'cos']
    A   = barras.loc[barra, 'A']
    E   = barras.loc[barra, 'E']
    N1  = barras.loc[barra, 'N1']
    N2  = barras.loc[barra, 'N2']

    # Matriz de rigidez no sistema local
    Kl = E*A/L*np.array([[ 1,  0,-1,  0],
                         [ 0,  0,  0,  0],
                         [-1,  0,  1,  0],
                         [ 0,  0,  0,  0]])

    # Matriz de rotação
    Mrot = np.array([[ cos,  sen,    0,   0],
```

```
                    [-sen,  cos,    0,   0],
                    [   0,    0,  cos, sen],
                    [   0,    0, -sen, cos]])
    # Recebendo os deslocamentos referentes ao elemento em␣
↪análise
    Ul = np.zeros(4)
    Ul[0] = U[2*N1-2]
    Ul[1] = U[2*N1-1]
    Ul[2] = U[2*N2-2]
    Ul[3] = U[2*N2-1]

    # Realizando o equilíbrio local

    F = np.dot(Kl,np.dot(Mrot,Ul))

    # Salvando o terceiro valor do vetor de forças por␣
↪convenção de sinais.
    Esf.append(F[2])
```

Agora, vamos plotar os esforços, sinalizando as barras comprimidas pela cor azul e as tracionadas pela cor vermelha.

[18]:
```
for barra in barras.index:
    # Vamos passar os nós para as variáveis N1 e N2
    N1 = barras.loc[barra, 'N1']
    N2 = barras.loc[barra, 'N2']
    ax = Esf[barra-1]
    cos = barras.loc[barra, 'cos']
    sen = barras.loc[barra, 'sen']
    tg  = sen/cos
    # Agora vamos acessar as coordendas de cada um dos nós
    x1, y1  = nos.loc[N1, ['X','Y']]
    x2, y2  = nos.loc[N2, ['X','Y']]
    y = [y1,y2]
    x = [x1,x2]

    plt.figure(1,figsize=(18,4.5))
    if ax>0:
```

```
        cor = 'r'
    elif ax == 0:
        cor = 'k'
    else:
        cor = 'b'
    plt.plot(x,y,cor,zorder = -1)

    plt.text(np.mean(x),np.mean(y),'{:.2f}kN'.format(ax/
↪1000),rotation =180*np.arctan(tg)/np.pi,
             horizontalalignment='center',
             verticalalignment='center',
             size = 16,
             weight = 'bold')

    plt.scatter(x,y, s=80,marker ='o',color ='black')


_=plt.title('Esforços atuantes')
```

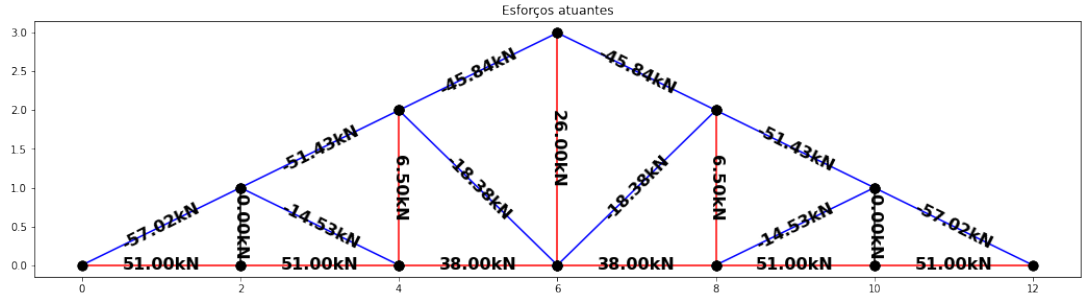


Figura 18 – Plotagem dos esforços atuantes