INTRODUÇÃO AO PYTHON

MÉTODOS E APLICAÇÕES

JOÃO VICTOR VILELA CASSIANO
WASHINGTON ERLY CONTATO

Olá meu nome é Washington Erly, responsável pelo capitulo ”I”, sou estudante de Engenharia Química pela Universidade Federal de Uberlândia. Junto com meu amigo João Victor decidi escrever essa apostila em busca de aprimorar um pouco mais meus conhecimentos e ajudar outras pessoas que tenham vontade de aprender mais sobre programação.

O capitulo ”I”trata-se um uma introdução a linguagem Phyton e diversas ferramentas que estaremos utilizando nos capítulos ”II”e ”III”, com o intuito de deixar esse capitulo de fácil entendimento busquei utilizar exemplos simples e diversos comentários dentro dos códigos para que todos consigam absorver de maneira rápida o que está acontecendo em cada linha.

Prazer, me chamo João Victor e sou um dos ”autores” desta apostila, responsável pelos capítulos ”II”e ”III”e vou me apresentar rapidamente. Sou estudante de Física de Materiais pela Universidade Federal de Uberlândia e tive a ideia dessa apostila durante o curso de ”Física Computacional”ao perceber que quase não havia materiais em português sobre programação com foco em cursos de exatas, então decidi escrevê-la com meu amigo Washington Erly (responsável pelo primeiro capítulo).

Ao longo dos capítulos II e III falaremos sobre métodos de extrema importância para qualquer estudante, no primeiro alguns métodos serão deduzidos e implementados com a linguagem bruta, já no segundo você aprenderá a utilizar pacotes como NumPy (Numerical Python), SciPy (Scientific Python), Matplotlib (visualização de dados) e etc.

Trata-se de um livro gratuito, direto (ou seja, sem abordar questões mais complicadas sobre interpretador, camadas e armazenamento de memória e etc) focado em estudantes de cursos de exatas. Todos os códigos presentes no livro estão no repositório do GitHub (GIT), quaisquer sugestões sobre a apostila podem ser enviadas para (joaocassianox7x@gmail.com)

Desejo uma boa leitura, que esse material te auxilie durante a graduação e que todos seus objetivos se concretizem!

Recomendamos fortemente que você abra e leia as documentações apresentadas ao longo de toda a apostila.

CONTEÚDO

Parte I. Introdução à Linguagem

1 RESUMO

Ao longo da apostila escreveremos vários código e dentre as inúmeras possibilidades de ambientes de desenvolvimento escolhi o Spyder por se tratar de um ambiente de desenvolvimento leve, gratuito e de fácil instalação (ele já vem com todos os pacotes utilizados ao longo da apostila, o que facilitará nossa vida).

Vídeos de como instalar o Spyder no Windows e Linux são encontrados no apêndice, ao final da apostila. Recomendo que você instale conforme é mostrado nos apêndices "A"e "B", depois veja o apêndice "C"para dar seus primeiros no Spyder, neste vídeo coisas básicas como "salvar"e "executar"um programa são apresentadas.

2 ATRIBUIÇÃO DE VARIÁVEIS

2.1 *Como as variáveis funcionam no Python*

Diferente de algumas linguagens em que você precisa definir o tipo de variável na sua declaração, como C e FORTRAN por exemplo, o Python nos permite definir uma variável conforme o dado atribuído a ela:

```
a=1 #inteiro
b= 3.1415 #float
c= "python" #string
d= True #Boolean
e = False #Boolean
print(a)
print(type(a))
print(b)
print(type(b))
print(c)
print(type(c))
print(d)
print(type(d))
print(e)
print(type(e))

#As variaveis foram impressas e a funcao type
#nos diz o tipo de variavel que estamos utilizando
# class 'int'
# class 'float'
# class 'str'
# class 'bool'
# class 'bool'
```

De um modo resumido essas são as variáveis que utilizamos constantemente em Python e observamos algumas diferenças em relação a outras linguagens no quesito de declaração. Porém devemos tomar cuidado com as variáveis globais e locais, que servem para o código todo e para uma parte do código respectivamente:

```
"""
Escopo de variaveis
1 - variaveis globais
    -  Sao reconhecidas em todo o programa

2 - variaveis locais
    - apenas no bloco onde foram declaradas
"""
variavel=True
if variavel == True:
    local='Verdadeiro'

print(local)

#Deste modo temos a impressao da string 'verdadeiro'
#no final do meu código
```

Mas o que aconteceria se nossa variavel recebe-se um valor 'False'? Veremos a seguir.

```
variavel=False
if variavel == True:
    local='Verdadeiro'

print(local)

#Deste modo recebemos "NameError: name 'local' is not defined"
#que nos mostra que a varivael local nao esta sendo criada
#pois ela esta dentro no nosso comando condicional 'if'
#para que isso nao ocorra a variavel local deveria ser inicializada
#fora do comando if
```

Usando o mesmo código vemos a diferença entre variáveis globais e locais e os possíveis erros que podem ocorrer caso tenhamos que acessar uma variável local sem que ela tenha sido inicializada.

3 PACOTES BÁSICOS

Ao longo de toda a apostila utilizaremos alguns pacotes. Tratam-se de arquivos de códigos previamente escritos para o Python. Dois dos mais básicos e muito utilizados são os pacotes "random"e "time", o primeiro serve para a geração de números pseudo-aleatórios, já o segundo para operações que envolvem tempo (geralmente para avaliarmos o tempo de código).

Usaremos pouquíssimas ferramentas desses dois camaradas, mas é interessante apresentá-los por dois motivos, o primeiro é para que você não se sinta perdido quando forem chamados e o segundo é apresentar a ferramenta "import"para chamar os módulos.

```
import time #chamamos o pacote time
import random #chamamos o pacote random
```

```
x=random.random() #numero aletorio entre 0 e 1
t=time.time() #tempo de execucao da maquina

#random() e time() são funcoes previamente escritas
#dentro dos modulos
```

Caso seja de seu interesse os dois pacotes possuem muitas ferramentas, os links para as documentações são Random e Time.

4 ESTRUTURA CONDICIONAL

4.1 *If*

Estruturas condicionais são a maneira mais simples de tomada de decisões que podemos observar na programação, um simples 'If' nos ajuda a tomarmos decisões, ja mostramos um exemplo na seção de variáveis de como ele é utilizado mas iremos aprofundar um pouco mais nessa seção. A estrutura 'if' utiliza uma condição verdadeira para ler o código que está dentro de sua estrutura: Porém as estruturas de condicionais possuem o Elif e Else, que também estão presentes em outras linguagens(o Elif em C por exemplo é o else if), são utilizadas para complementar o If e deixá-lo mais completo e dinâmico, como mostrado abaixo:

```
"""
Neste codigo verificamos se a pessoa possui a idade
menos de 18 anos, igual a 18 anos ou maior de 18 anos
"""
    idade=26

if idade < 16:
    print("menos que 18 anos")
elif idade==18:
    print("Tem exatamente 18 anos")
else:
    print("mais que 18 anos")

#A mensagem impressa nesse codigo é: Maior que 18 anos
# Se a variável idade que é do tipo Inteira receber outro valor como
# 17 a primeira condição que será verdade de modo que a mensagem
#impressa seja: menos de 18 anos
```

Contudo a estrutura If não seria de tamanha utilidade sem os estruturas lógicas And,Or, Not e Is. Esses comandos nos ajudam a aprimorar nossos códigos e aumentar a complexabilidade das verificações do If. Ja que sabemos que o If necessita de uma afirmação verdadeira (uma expressão booleana: True) para que possa ler o que esta dentro do seu bloco precisamos entender como que funcionam esses estruturas e como vamos utiliza-los nessa estrutura.

4.2 *Expressões lógicas ou Booleanas*

Para entendermos melhor essas expressões lógicas utilizaremos alguns exemplos simples comparando valores numéricos e informar o valor impresso para que possamos visualizar como eles funcionam.

```
"""
Para o 'and' - ambos devem ser true
Para o 'or' - um deles devem ser true
Para o 'not' - o valor do booleano é invertido, ou seja,
se for true vira false e o inverso tambem ocorre
"""
print(18 > 10) #resposta: True
Print(18 < 10) #resposta: False

#agora vamos usar as expressoes booleanas.

print(18 > 10 and 18 < 10)
# a resposta sera False pois a expressao and necessita
#que as duas expressoes sejam True

print(18 > 10 or 18 < 10)
# a resposta sera True pois a expressao Or necessita
#que apenas uma das expressoes sejam True

#A expressao not inverte o valor booleano
# usaremos o exemplo de and para verificar esse comando

print(not(18 > 10 and 18 <10))
#Nesse caso a expressao and nos retornara False, porem o Not ira
#inverte-la, deste modo temos True como resposta
```

4.3 *If com Expressões Lógicas*

Agora que revisamos If e as expressões logicas iremos utilizar as duas em conjunto para melhorarmos nossas estruturas condicionais. Neste exemplo usaremos o Coeficiente de Reynols para avaliar o regime :

```
"""
Coeficiente de Reynolds
O coeficiente de reynolds nos mostra que tipo de regime esta
acontecendo em um escoamento
Se o numero de reynolds esta abaixo de 2100: Regime Laminar
Acima de 4000: Regime Turbulento
e entre os dois Regime transiente
"""
Re=1500 #valor aleatorio

if Re<2100:
    print("Regime Laminar")
elif Re>2100 and Re<4000:
```

```
    print("Regime Transiente")
else:
    print("Regime Turbulento")

#Deste modo conseguimos ver que o And no elif esta impondo
#a condição de que as duas afirmacoes sejam verdadeiras
```

Outro exemplo que podemos utilizar é o primeiro exemplo de If um pouco modificado.

```
idade=25
sexo='F'
if idade < 16 and sexo=='F':
    print("menos que 18 anos e sexo feminino")
elif idade==18 and sexo == 'F':
    print("Tem exatamente 18 anos e do sexo feminino ")
elif idade > 18 and sexo=='F':
    print("mais de 18 anos e do sexo feminino")
elif idade < 16 and sexo=='M':
    print("menos que 18 anos e do sexo masculino")
elif idade==18 and sexo=='M':
    print("Tem exatamente 18 anos e do sexo masculino")
else:
    print("mais que 18 anos e do sexo masculino")
```

Alguns exercícios para fixação:

- Faça um código que some a+b e multiplique o resultado da soma por a. Para isso verifique se a=0 e escreva o resultado final na tela.
- Verificar se um número é par ou ímpar, negativo, positivo ou zero.
- Elabore um programa que forneça as raízes de uma equação de segundo grau, informando se existe 2 raízes, 1 raiz ou nenhuma (leve em consideração raízes complexas, para ajudá-lo nessa tarefa link).

5 VETORES E MATRIZES

5.1 *Introdução a Listas e Tuplas*

Antes de começarmos a mexermos com vetores e matrizes precisamos entender os conceitos de Listas e Tuplas para não existir confusões futuras. Listas e Tuplas são conjuntos de dados armazenados em "espaços"determinados e com direcionamento (posições) que podem ser chamados conforme a ordenação da lista. As listas e tuplas são conhecidas de modo geral como vetores na programação, também são encontradas em outras linguagens, como C, Java e FORTRAN por exemplo (dependendo da linguagem são chamadas de arrays, mas aqui esse nome será usado em outra situação). Os índices iniciam-se em 0 e vão ate N-1 sendo N o tamanho total da sua lista ou tupla.

5.2 *Listas*

O Python possui uma diferença crucial em relação a outras linguagens com relação as listas, em C e Java essas listas possuem tamanho fixo, ou seja, esse tamanho não pode ser alterado após sua declaração. Já o Python nos permite adicionar novos valores a nossa lista conforme faz-se necessário. Outra vantagem que podemos observar é a possibilidade de adicionar qualquer tipo de dado em nossa lista, ou seja, podemos colocar valores inteiros, reais, strings, booleanos e etc. No exemplo abaixo iremos ver como utilizamos listas e alguns comandos de modificação da mesma.

```
"""
Exemplos de listas

"""
lista=[1,9,4,25,23,3,1,44,42,27]
#a nomenclatura de lista se da pela variavel e [] e os
#elementos separados por virgula

lista2=['a','b','t','c','g']
#a lista pode receber qualquer tipo de variavel

lista3=list('washington erly')
# o comando list eh utilizado pra transformar qualquer iteravel
#em lista, como o exemplo acima que eh uma string se
#tornou uma lista


lista4=list(range(11))
#e como mostrado acima a funcao list pode ser usada em diversos
#iteraveis nao apenas strings

lista5 = [1, 5.34, 'a','ola', True]
#Uma lista pode receber qualquer tipo de dado.
```

Agora, alguns comandos que podem nos ajudar a utilizar melhor as listas, usaremos as listas anteriores como exemplo:

```
    lista=[1,1,9,4,25,23,3,1,44,42,27]
lista2=['a','b','t','c','g']
lista3=list('washington erly')
lista4=list(range(11))

#exemplo de If em listas
if 8 in lista4:
    print('Encontrei')
else:
    print('Não existe o numero 8 nesta lista')

#como existe o 8 na lista o que será impresso é: Encontrei

#len()
```

```
#A função Len nos mostra o tamanho de um iterável

print(len(lista))

#Será impresso: 10 e sempre retorna um inteiro

#.sort()
lista.sort()
print(lista)
#O comando sort() colocar nossa lista em ordem crescente
#A impressao de lista após o .sort() sera:
#[1, 1, 1, 3, 4, 9, 23, 25, 27, 42, 44]

#.count é um comando de contagem dentro do iterável que busca
#quantas vezes um determinado valor se encontra dentro do mesmo

print(lista.count(1)) #resposta: 3
print(lista3.count('n')) #resposta: 2


#Comando .append() adiciona um valor no final da lista

lista.append(43)
print(lista)

#o que sera impresso é: [1, 1, 1, 3, 4, 9, 23, 25, 27, 42, 44, 43]

#Além do .append() existe o .insert(,) que nos permite adicionar
#um valor em uma posição especifica

lista.insert(0,43)
print(lista)
#o que sera impresso é: [43, 1, 1, 1, 3, 4, 9, 23, 25, 27, 42, 44, 43]

# Além dos comandos acima existem vários outros e com funções
#muito relevantes para a área de exatas.

lista=[1, 1, 1, 3, 4, 9, 23, 25, 27, 42, 44]
print(max(lista)) #Mostra maior valor na lista: 44
print(min(lista)) #Mostra menor valor na lista: 1
print(sum(lista)) #Mostra soma dos valores na lista: 180
```

Antes de passarmos para Tuplas devemos ter certeza que o entendimento de listas esta bem fixado pois a manipulação desses tipos de variáveis se assemelham muito. Além de tudo que foi falado acima, como acessar pontos específicos de listas é muito importante. Vimos na função .insert() que ela necessita de uma posição (índice) onde o valor será atribuído. Os índices começam em 0 e vão até N-1, com N o tamanho total da lista. Mas como acessamos, por exemplo, a posição 3 de uma lista de 10 números.

```
lista=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

print(lista[3]) #sera impresso: 3
#lembrando que as posições (indices) começam em 0
#entao se peço o indice 3, busco o quarto numero da lista que
#é um 4, para nos ajudar podemos usar a função Len()
#para descobrir o tamanho da lista e termos noção
#de onde ela termina

print(len(lista))
# o valor impresso é: 10 entao sabemos que os indices
#vao de 0 a 9

#Podemos selecionar um pedaço da lista

print(lista[0:2]
#Será impresso (0,1)

#Além de selecionarmos um pedaço podemos usar índices negativos

print(lista[-1])
#valor impresso: 10

#Além disso podemos inverter a impressão da lista ou
#selecionar apenas um pedaço dela
print(lista[::-1])

#o que será impresso: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

5.3 *Tuplas*

Tuplas de um modo simples de explicar são listas não mutáveis, ou seja, que não aceitam modificação referentes a adição,alteração ou remoção de elementos contidos nela. A maneira mais fácil de diferenciar listas de tuplas são os caracteres delimitadores. Sendo que listas utilizam colchetes e tuplas usa parêntesis para delimitar, porém, na documentação oficial do Python consta que uma tupla é declarada pela utilização do operador vírgula, e que é utilizado para separar os elementos.

```
tupla1=(1,2,3,4,5,6)
print(tupla1)
print(type(tupla1))

tupla2=1,2,3,4,5,6
print(tupla2)
print(type(tupla2))

#Vemos que é impresso a mesma mensagem:
#(1, 2, 3, 4, 5, 6)
#—<—class 'tuple'—>—
```

```
#Nesse exemplo vemos que podemos declarar
#tuplas das duas maneiras e
#são a mesma coisa pois mostra que a classe delas é igual
```

Conseguimos perceber a proximidade de listas e tuplas em suas declarações diversas funções que utilizamos em listas podem ser usadas em tuplas como os exemplos abaixo:

```
tupla1=(1,2,3,4,5,6)
tupla2=1,2,3,4,5,6

print(max(tupla1))
#valor impresso: 6
print(min(tupla1))
#valor impresso: 1
print(sum(tupla1))
#valor da soma: 21

tupla3=tupla1+tupla2
print(tupla3)

#nessa soma de tuplas não estamos modificando as tuplas
#originais e sim atribuindo os valores das duas tuplas e
#uma terceira, esse é o tipo de 'modificação'
#que utilizamos em tuplas.
```

Para acessar pontos específicos de tuplas usamos o mesmo conceito de listas utilizando colchetes e indicando a posição através do índice como visto anteriormente

5.4 *Matrizes*

Após falarmos de listas e tuplas chegamos em uma das partes mais importante do capitulo "I", as Matrizes, uma matriz é um conjunto de valores organizados em linhas e colunas. O modo de leitura dos valores de uma matriz se da pelos índices referentes a posição em linha e coluna, podemos pensar em nesse sistema como coordenadas, muitas vezes podemos encontrar matrizes escritas da forma M_{ij} , sendo i a linha e j as colunas Nesse tópico iremos ter apenas uma ideia de como declarar e ler matrizes e nos aprofundar nelas dentro de estruturas de repetição, mais especificamente em For (uma das duas estruturas que iremos ver nessa apostila). A utilização de matrizes nos ajudam a resolução em diversos problemas encontrados nas mais diversas áreas e facilitam a organização de dados de modo geral.

```
"""
Como gerar matrizes:
matrizes são basicamente listas dentro de uma lista.

"""
listas = [[1,2,3],[4,5,6],[7,8,9]] #matriz 3x3
```

```
#Para acessar pontos dessa matriz usamos o mesmo
#formato de listas
print(listas[0][0]) #Será impresso o valor: 1
print(listas[0][2]) #Será impresso o valor: 3
print(listas[1][1]) #Será impresso o valor: 5
print(listas[2][0]) #Será impresso o valor: 7
print(listas[2][2]) #Será impresso o valor: 9
#lembrando que nossos índices começam em 0 e vão ate N-1
```

Normalmente após ver um exemplo simples pensamos que matrizes são coisas bem fáceis e acabamos subestimando porém isso não deve ser feito pois para uma matriz pequena 3x3 é fácil manipula-la e preenche-la. Mas se começarmos a gerar matrizes cada vez maiores (6x6, 10x10, 100x100 e 1000x1000 e assim por diante) não seria tão fácil fazer isso "na mão". Por isso existem funções que nos auxiliam quando precisamos de questões repetitivas. Podemos colocar qualquer tipo de variável dentro de uma matriz pois ela é formada de listas e vimos anteriormente que listas aceitam todos os tipos de variáveis

6 ESTRUTURA DE REPETIÇÃO

Na programação existem estruturas de repetição nos permitem repetir comando diversas vezes utilizando condições ou contadores como restrições para determinar o momento de parada.

6.1 *While*

A estrutura While possui uma condição booleana para que o comando dentro de seu bloco seja repetido, ou seja, é necessário que condição nos de uma resposta True (tipo booleana).

```
    num=1
while num < 10:
    print(num)
    num+=1 #que é igual num=num + 1

    #nesse exemplo estamos fazendo a impressão dos números de 1 a 9
    #algumas coisas que devem ser observadas são:
    # 1- para inicializar um while a afirmação condicional
    # ja deve ser True
    # 2 - a cada 'loop' é como se voltássemos a linha inicial do while
    # e o motivo de falar isso e que um erro inicial de quem
    #nunca mexeu com programação é esquecer a condição de parada
    # 3 - a condição de parada é que a variavel 'num' chege a 10
    # para isso temos num+=1 que acrescenta +1 a cada
    #loop até chegar em 10
```

Pelo exemplo anterior vimos que em 3 linhas tem muita coisa acontecendo, estruturas de repetição nos ajudam a reduzir diversos esforços desnecessários e reduzem muito o tempo que utilizamos para escrever um código desde que bem entendidos. Um dos exemplos clássicos de While é um codigo para verificar se um numero é par ou não:

```
num=37
i=1
contador=0
while(i<=37): #
    if(num%i==0):
        contador+=1
i+=1

if(contador==1 or contador==2 ):
    print('É primo')
else:
    print('Não é primo')

#Um erro comum é esquecer de colocar a variavel i
#para ter um acréscimo a cada loop
#o que faz o loop ser 'infinito', ou seja, ele não tera fim
```

No exemplo acima pode-se observar um exemplo melhor a utilização de loops e junto a isso é visto que as estruturas estão começando a mostrar mais blocos, que foram citados Atribuição de Variáveis, que diferente de linguagens como C por exemplo utilizando chaves( ) para definir seus blocos, o phyton usa apenas um espaçamento (de 4 espaços ou um 'Tab') e devemos atentar a esses detalhes.

6.2 *For*

Diferente do While em python que se assemelha muito com outras linguagens, o For já não é tão parecido. Na condição do For, em C/C++ por exemplo, é necessário declarar uma variável de controle ou já ter sido declarado anteriormente (um contador global por exemplo), a condição de parada (onde é analisado se é True ou False) e o acréscimo dado ao contador. Em Python o For possui uma estrutura um pouco diferente para aqueles que já a viram em outras linguagens. Em python o For é utilizado em iteráveis, que são basicamente variáveis que possuem valores em sequencia falando a grosso modo. Exemplos de iteráveis são Strings, Listas, Tuplas, Dicionários entre outros. A utilização do for em iteráveis nos permite que determinados sua parada apenas no tamanho do nosso iterável. E podemos ver que de certo modo é muito mais simples do que em C. No exemplo abaixo vemos o for sendo utilizado em iteráveis diferentes.

```
    string= 'string aleatoria'
lista =[1,3,5,7,9]

for letra in string:
print(letra)
```

```
#será impresso cada letra da string um em cada linha

for num in lista:
print(num)

#será impresso cada numero da lista um em cada linha

for num in range(1,5):
print(num)

    #será impresso os números de 1 a 4 pois a função range
    #é um iterável
```

Além disso podemos utilizar a função for dentro de outra função for, comumente conhecida por laço for. Esse tipo de estrutura é muito utilizada para modificar ou preencher matrizes.

Exercícios para fixação:

- Escreva um código que mostra uma contagem regressiva na tela, iniciando em 10 e terminando em 0. Mostrar uma mensagem "FIM!" após a contagem;
- Apresentar o total da soma obtida dos cem primeiros números inteiros;
- Somar os números pares, entre zero e 1000, e ao final imprimir o resultado;
- Em Matemática, o número harmônico designado por H(n) define-se como sendo a soma da série harmônica: ´ H(n) = 1 + 1/2 + 1/3 + 1/4 + ... + 1/n Faça um programa que leia um valor n inteiro e positivo e apresente o valor de H(n);
- Encontre os cem primeiros números da Sequência de Febonacci ($a_{n+2} = a_n + a_{n+1}$).

7 FUNÇÕES

Funções são estruturas que utilizamos quando queremos usar um mesmo código diversas vezes em nossos códigos, diferentes de estruturas de repetição que utilizamos para repetir um mesmo conjunto de linhas em uma única vez por exemplo. Para declarar uma função o Python possui uma palavra reservada, o 'def' (print, and ou or são exemplos de palavras reservadas) esse tipo de palavra você não pode usar para criar variáveis e normalmente os compiladores já as destacam em cores diferentes para mostrar que elas já possuem funções especificas. Além da declaração devemos entender que funções são 'chamadas' em nossos códigos, chamar uma função se da pelo nome que você da a sua função e os valores informados para que ela possa fazer sua função. Com o exemplo abaixo iremos ver como declarar uma função, como chama-la e além disso algumas 'regras' devem ser seguidas.

```
def quadrado(numero): #return numero*numero
return numero**2
```

```
#O nome que foi atribuído a função é a forma
#de chamar a função, neste caso ela se chama 'quadrado'

print(quadrado(7))
#valor impresso: 49

print(quadrado(2))
#valor impresso : 4

print(quadrado(5))
#valor impresso: 25


    #Instruções a serem seguidas:
    #1 - nomes de funções seguem a regra de nome de variáveis
    #não devem ter espaços e caso necessário usar underline
    #2 - devemos sempre respeitar o bloco da função
    #igual ja vimos em estruturas de condição e repetição
    #3 - funções são declaradas no inicio do codigo
    #(essa não é uma regra mas é fortemente indicado, pois
    #caso seja indicado depois e você volte e tente chama-la
    #ira dar erro pois ela está declarada após aonde você chamou
```

Como vimos acima, para declarar a função a palavra 'def' se encontra em minusculo. As instruções citadas devem ser seguidas mais como apoio mas são fortemente indicadas para todos. Não devemos nos prender ao exemplo anterior, as funções podem receber vários valores e podem ou não retornar um valor. E todas as estruturas anteriores podem ser utilizadas dentro das funções.

```
"""
Além de passarmos um parametro, podemos passar varios e
também colocar um parametro opcional
"""
def exponencial(num,pot=2):
    return num**pot
#a quantidade de valores que devemos informar se encontra
#dentro do parenteses porém um desse deixamos opcional
#colocando ele padronizado em 2
#caso deseje elevar a outro valor diferente de 2
#eu informo o numero conforme os exemplos abaixo

print(exponencial(2,3))
#valor impresso: 8

print(exponencial(3,2))
#valor impresso: 9

print(exponencial(3))
#valor impresso: 9
```

```
#Como visto se padronizamos um dos valores não precisamos
#informar um segundo valor quando chamamos a função
```

Além disso podemos utilizar uma função dentro de estruturas.

```
from random import random

def joga_moeda():
    valor=random()
    if valor > 0.5:
        return 'cara'
    return 'coroa'
cont1=0
cont2=0
for n in range(100000):
    if joga_moeda() =='cara':
        cont1+=1
    else:
        cont2+=1

print(cont1/cont1+cont2,cont2/cont1+cont2)


#Os valores dependem da função random porem ao colocar diversas
#em uma função for ira chegar proximo de 0,5
```

Funções podem receber qualquer tipo de dado ou estrutura e sempre deve-se lembrar que a função pode ou não retornar um valor, tudo depende de como ela vai ser utilizada.

- Faça uma função que receba os valores de uma função de segundo grau e retorne as raízes dessa funções;
- Faça um programa que receba o valor de e adicione em uma lista.

8 LER E ESCREVER ARQUIVOS

Até o momento utilizamos os dados que precisávamos diretamente no código, criávamos listas e variáveis 'na mão' porém isso não é viável e por isso que precisamos de usar outros tipos de arquivos para suprir essa necessidade. Os arquivos de texto (.txt) são a forma mais comum para essa função. O motivo é bem simples, podem armazenar uma quantidade grande de dados e facilmente modificados por simples programas como bloco de notas.

Repare que no código abaixo escrevemos "/", mas o correto é ao contrário. Trata-se de uma limitação no programa em que a apostila foi escrita.

```
#Para ler e escrever em um arquivo utilizamos
#os comandos 'r' (read -—>— ler) 'w' (write -—>— escrever)
#'a' (write -—>— escrever)
```

```
#Para criar um arquivo podemos usar o "w"
#caso exista um arquivo com o mesmo nome
#ele substitui se não ele cria o arquivo

#Para criar um arquivo podemos usar o "a"
#caso exista um arquivo com o mesmo nome
#ele escreve no final senão ele cria o arquivo


#o exemplo abaixo mostra a criação de um arquivo
# e o comando de escrita onde podemos observar
#que podemos escrever basicamente tudo

arquivo=open('texto.txt','w')
arquivo.write('posso escrever frases/n') #a barra deve estar
                                         #para a esquerda
arquivo.write('numeros: 3,14...../n')
arquivo.write('Basicamente qualquer coisa/n')
arquivo.close()


#E para ler este arquivo
arquivo=open('texto.txt','r')
print(arquivo.read())
#O que foi impresso:
#posso escrever frases
#numeros: 3,14.....
#Basicamente qualquer coisa
arquivo.close()

#Para adicionar novas informações
#sem subistituir o que já foi escrito
arquivo=open('texto.txt','a')
arquivo.write('para adicionar novas informações')
arquivo.close()
```

Porém dessa maneira estamos criando arquivos de texto usando caminho relativo, ou seja, os arquivos .txt que criamos estão sendo armazenados na mesma pasta onde nossos codigos estão sendo salvos. Para direcionar aonde devem ser salvos devemos usar o que é chamado de caminho absoluto, que indica exatamente aonde esse arquivo deve ser criado e/ou escrito para armazena-lo.

```
arquivo˙novo=open('C:"projetos"dados.txt','w')
arquivo˙novo.write('Agora estamos criando e armazenando este'
                   ' arquivo na pasta projeto dentro de C:')
arquivo˙novo.close()
#Como vimos no exemplo anterior, estamos criando um
#arquivo dentro desta pasta
```

Um exemplo útil seria armazenar os valores de x e f(x) de uma função especifica.

```
x=range(100)
f=[]
for i in x:
    f.append(i**2)
    arq=open('dados.txt','w')
for i in range(len(x)):
    arq.write(str(x[i]) + ' ')
    arq.write(str(f[i]) + ' ')
    arq.write('“n')
arq.close()


#Deste modo conseguimos diversos pontos de uma função
#podemos usar diversas funções
```

- Escreva um código que receba os valores de x e f(x) da função seno e cosseno e armazene em um arquivo .txt em duas colunas separadas por espaço;
- Faça um programa que receba uma lista de compras e transforme em um arquivo .txt.

9 FUNÇÃO LAMBDA E COMPREHENSION

Nesse último tópico do Capitulo I iremos conhecer as funções Lambda e Comprehension, são assuntos um pouco delicados então tomaremos uma abordagem bem simples para que tudo possa ser entendido da forma mais rápida possível. Essas duas funções nos ajudam a melhorar nossos códigos em questão de velocidade e deixa eles com uma estrutura mais limpa simplificando muita coisa.

9.1 *Lambda*

Funções Lambda ou Funções anonimas podem ser comparadas com funções que vimos anteriormente porém escritas de forma mais simples.

```
#função em python

def soma(a,b)
    return a+b

def funcao(x):
    return 3*x +1

#e para utiliza-las

print(funcao(4))
#valor impresso: 13

print(funcao(7))
#valor impresso: 22
```

```
#As funções acima são exemplo do que vimos anteriormente e agora
#iremos ver como escreve-las no formato Lambda

#para declarar uma função lambda usamos sua palavra reservada
#porem devemos atribui-la a uma variável

calcular = lambda x: 3*x +1

#e para utiliza-la

print(calcular(4))

print(calcular(7))

mult = lambda a,b: a*b

print(mult(2,3))

print(mult(3*3))

#Deste modo vemos o quão mais simples é escrever uma função
#utilizando o lambda.
```

E essas funções pode receber outras variáveis não só inteiros.

```
nomecompleto = lambda n,sn: n+' '+sn
print(nomecompleto('washington','erly'))

#neste exemplo estamos recebendo 2 strings, concatenando elas e
#colocando a primeira letra em maiúsculo e cortando o excesso de espaços
#caso exista
```

9.2 *Comprehension*

Utilizando Comprehension podemos gerar listas, conjuntos e dicionários através de outros iteráveis, mas iremos focar em listas como já citado acima. As list comprehension utilizam uma estrutura dentro de colchetes sendo ela um for utilizando um iterável como podemos observar abaixo.

```
iteravel= [1,2,3,4,5]

res = [numero*10 for numero in iteravel]
print(res)
#lista impressa: [10,20,30,40,50]

"""
para entender melhor devemos dividir o que esta
acontecendo em duas partes
```

```
1 -  for numero in num
2 - a segunda numero*10
"""
#Podemos utilizar outros tipo de calculos não necessariamente
#multiplicação

res2 = [numero/2 for numero in iteravel]
print(res)
#Lista impressa: [0.5, 1.0, 1.5, 2.0, 2.5]

#E podemos fazer outros cálculos só lembrando de que a parte
#inicial é referente a esses cálculos e o For ao iterável base


numeros=[1,2,3,4,5,6]

pares = [numero for numero in numeros if numero % 2 == 0]
impares = [numero for numero in numeros if numero % 2 == 1]

print(pares)
#lista impressa: [2, 4, 6]
print(impares)
#lista impressa: [1, 3, 5]
```

Podemos observar que esse tipo de estrutura facilita a criação de novas listas e de um modo simples e bem 'limpo' e podemos mesclar outras estruturas que vimos anteriormente, como a funções e/ou funções Lambda. Todos esses processos poderiam ser feito utilizando For e as funções de listas porem ficariam muito maiores e trabalhosos.

```
lista= [1,2,3,4,5]
def quadrado(numero):
    return numero**2

res3 = [quadrado(numero) for numero in lista]
print(res3)
#Lista impressa: [1, 4, 9, 16, 25]
```

Como você já chegou no final do primeiro capítulo, espero que tenha aprendido bastante, mas como muitas ideias foram apresentadas, faça os exercícios abaixo com total liberdade para utlizar todas as ferramentas aprendidas até agora:

- Escreva uma função que retorne o fatorial de um número "n";
- Escreva uma matriz quadrada em que os elementos são 0 e 1 distribuídos aleatoriamente (utilize a função random.randint());
- Escreva a matriz acima em um arquivo ".txt", depois leia esse arquivo, armazene os elementos em uma nova matriz e transponha esses elementos (linhas viram colunas);
- Faça um código que retorne a sequência de Fibonacci com no máximo duas linhas.

Parte II. Métodos Numéricos

Uma Revisão aos Velhos Conhecidos Como a "apostila"tem por objetivo encapsular os conhecimentos computacionais básicos necessários para um curso de exatas, a partir de agora o leitor irá se deparar com métodos numéricos comuns e extremamente úteis.

Neste capítulo serão feitas apresentações analíticas dos métodos e implementação com funções e métodos simples do Python, sem utilizar pacotes. Nos capítulos seguintes olharemos para os métodos aqui apresentados e mais alguns igualmente interessantes, mas amparados pelos pacotes científicos do Python, ou seja, primeiro os algorítimos serão apresentados aqui e escritos manualmente e depois usaremos os mesmos algorítimos mas com a implementação por pacotes que no geral são mais rápidos e completos que os nossos.

10 MÉTODOS DE INTEGRAÇÃO

10.1 *Método Trapezoidal*

Trata-se do método mais intuitivo possível para o cálculo de integrais. A área de um trapézio é "$A = \frac{h(b+B)}{2}$", ou seja, se considerarmos os valores das bases como as imagens das funções e a altura como o intervalo no eixo horizontal podemos reescrever o ente anterior como "$A = \frac{\Delta x(f_{j+1}+f_j)}{2}$", mais claramente:

$$A = \frac{\Delta x(f_{i+1} + f_i)}{2} \tag{1}$$

Para um intervalo dividido em "n"pontos:

$$\begin{aligned} A_T = A_1 + A_2 + A_3 + A_4 + A_5 + A_6 + A_7 + A_8 + ... + A_n = \\ = \frac{\Delta x(f_1 + f_0)}{2} + \frac{\Delta x(f_2 + f_0)}{2} \frac{\Delta x(f_3 + f_2)}{2} + ... + \frac{\Delta x(f_n + f_{n-1})}{2} \end{aligned} \tag{2}$$

O camarada acima pode ser reescrito em forma de somatório como:

$$A_T = \frac{\Delta x}{2} \sum_{j=0}^{n-1} (f_{j+1} + f_j) \tag{3}$$

Simples não? Façamos agora um código em Python para calcular a integral a seguir que não possui solução analítica: $\int_a^b e^{-x^2} dx$

```
import math

e=math.e

def func(x): #funcao que desejamos integrar
    return(e**(-x))

n=10**4 #numero de pontos
a=0
b=10
```

```
dx=(b-a)/n

I=0

x=[(i*b/n)-a for i in range(n)] #vetor que vai de a ate b em n pontos

A=0
for i in range(n-1):
    A+=(dx/2)*(func(x[i+1])-func(x[i]))
```

Agora, vamos aos exercícios! Integre as seguintes funções e compare com as soluções analíticas:

- $f(x) = e^{-x^2}$, $a = 0$ *e* $b = \infty$ $\longrightarrow$ (R = $\frac{\sqrt{\pi}}{2}$);
- $f(x) = sin(x)cos(x)$, $a = 0$ *e* $b = 2\pi$ $\longrightarrow$ $(R = 0)$;
- $f(x) = sin(x)sin(2x)$, $a = 0$ *e* $b = 2\pi$ $\longrightarrow$ 0 $(R = 0)$;
- $f(x) = cos^2(x)$, $a = 0$ *e* $b = 2\pi$ $\longrightarrow$ (R=π) ;

10.2 *Método de Monte Carlo*

Já vimos um método simples de recorrência, agora veremos um que utiliza da estatística! O algorítimo em si parte de um princípio muito simples, a ideia do "Teorema do Valor Médio", que consiste em:

$$f(x) \approx \frac{f(a) + f(b)}{2}, \; x \in [a, b] \tag{4}$$

Para um número maior de elementos:

$$f(x) \approx \frac{\sum_i^n f(x_i)}{n}, \; x \in [x_{min}, x_{max}] \tag{5}$$

Agora, partindo dessa ideia:

$$\int_a^b f(x)dx = \langle f \rangle (b - a) \tag{6}$$

Ou seja, a integral pode ser dada pelo valor médio da função num intervalo vezes o próprio intervalo. Obteremos esse valor médio via números aleatório, é bastante intuitivo que para um grande número de testes o valor médio obtido convergirá para o valor médio real.

Para poucas dimensões ($N < 3$) métodos como o trapezoidal são consideravelmente mais rápidos, mas para integrais multi-dimensionais o método de Monte Carlo converge consideravelmente mais rápido!

Calculemos então o valor de π via integração de Monte Carlo com o código escrito em Python. A função será $f(x) = \sqrt{1 - x^2}$ que se integrada de "0 $\longrightarrow$ 1"obteremos a resposta "$\frac{\pi}{4}$"e, esse será nosso parâmetro de convergência:

```
import random #pacote de numeros aleatorio do python
import math #pacote com constantes importantes

pi=math.pi
```

```
res=pi/4 #resultado analitico

def func(x):
    return(math.sqrt(1-x**2))
n=10**5 #convergência pequena
media=0

for i in range(n): #random.random() retorna um número
                   #aleatório entre 0 e 1
    media+=func(random.random())
Integral=media*1/n #produto entre a media e o intervalo
erro=((res-Integral)**2)**0.5

#Integral=0.785204679641125
#Erro=0.0001934837563233316
#Três casas de precisão
```

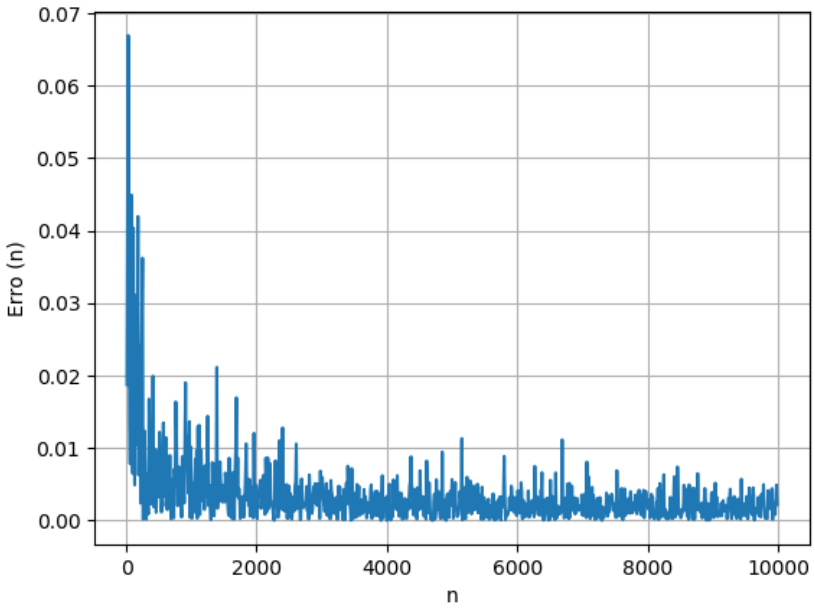


Fig. 1: Erro da integral com relação ao número de pontos utilizados,

Resolva agora os seguintes problemas com os métodos acima com o intervalo (-10,10):

- Integral acima, utilizando a estrutura de Comprehension ao invés do "for";
- $f(x) = x^2$
- $f(x) = sin^2(x)$
- $f(x) = cos^2(x)$
- $f(x) = sin^2(x)cos^2(x)$

11 SOLUÇÃO DE EDO'S

Todos os métodos a seguir são bem simples e exigirão do leitor apenas o conhecimento básico de expansão de funções via métodos de Taylor/Maclaurin, caso não se lembre, fique calmo, as expansões e manipulações serão demonstradas passo a passo!

11.1 *Euler de Segunda Ordem*

Como trabalharemos em espaços completos, vale o seguinte conceito:

$$f(t+\Delta t) \approx \sum_n \frac{f^{(n)}(t_0)}{n!}\Delta t^n \tag{7}$$

Tipicamente, "Δt"é o tamanho do "passo"no seu problema ($\Delta t = \frac{Pf-Pi}{N}$) onde "Pf"é o ponto final, "Pi"o ponto inicial e "N"o número de pontos entre os pontos inicial e final, ou seja, imagine que resolvameos uma EDO de "0"até "10"com "100"pontos, para esse caso "$\Delta t = \frac{10-0}{100}$". Agora, ao expandir essa série até os termos de segunda ordem:

$$f(t+\Delta t) = f(t_0) + f'(t_0)\Delta t + f''(t_0)\Delta t^2 + ... \tag{8}$$

Assumindo "$\Delta t \longrightarrow 0$ "podemos reescrever a expansão como:

$$f(t+\Delta t) \approx f(t) + f'(t)\Delta t + O^2 \tag{9}$$

Aqui podemos escrever os termos de ordens superiores como "O^2"e esse será o erro do nosso método, reescrevendo a equação acima em notação indicial e simplificando os termos:

$$f_{n+1} = f_n + f'_n\Delta t \tag{10}$$

Pode não parecer, mas agora, estamos aptos a resolver qualquer EDO de primeira ordem se possuirmos o valor de "f(0)", com isso podermos ir a um caso bem geral :

EDO:

$$y_1(x)\frac{df(x)}{dx} + y_2(x)f(x) = y_3(x)$$

Isolando a derivada:

$$y_1(x)\frac{df(x)}{dx} = y_3(x) - y_2(x)f(x)$$

Agora, sob notação indicial:

$$f'_n = \frac{y_{3_n} - y_{2_n}f_n}{y_{1_n}}$$

Supondo $y(x_0)$ conhecido, podemos usar a fórmula de Euler encontrada:

$$y_{n+1} = y_n + \frac{y_{3_n} - y_{2_n}f_n}{y_{1_n}}\Delta t$$

Explicitamente:

$$n = 0 \longrightarrow y_1 = y_0 + \frac{y_{3_0} - y_{2_0}f_0}{y_{1_0}}\Delta t$$

$$n = 1 \longrightarrow y_2 = y_1 + \frac{y_{3_1} - y_{2_1}f_1}{y_{1_1}}\Delta t$$

$$n = 2 \longrightarrow y_3 = y_2 + \frac{y_{3_2} - y_{2_2}f_2}{y_{1_2}}\Delta t$$

Agora que já implementamos o conceito para um caso bem geral, façamos explicitamente para o caso mais simples, onde "$y_1 = 1,\ y_2 = 1\ e\ y_3 = 0$": $y' = y$ com $y_0 = 10$.

```
import math

y=[] #lista vazia para os valores da funcao
ydot=[] #lista vazia para as derivadas

y0=10
y.append(y0) #adicionando o ponto inicial

n=10**5 #numero de pontos
pi=0 #ponto inicial
pf=15 #ponto final
delta=(pf-pi)/n

def edo(x):
    return(x)

for i in range(n):
    y.append(y[i]+delta*edo(y[i]))
```

Baseado no exemplo anterior, resolva as seguintes edos para condições iniciais de sua escolha.

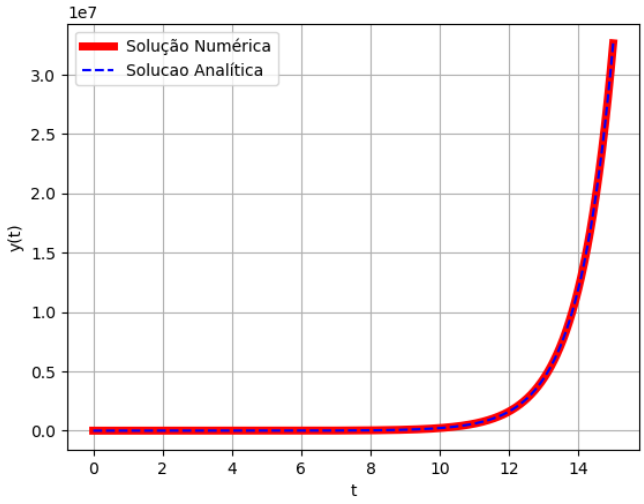


Fig. 2: Comparação entre a solução numérica (Euler) e a resposta analítica y(x)=10e^x

- $y' + t = y$
- $x^2y' - y = sen(t)$
- $y' + y = sin(y)$
- $y'^2 + y^2 = e^y$

Para EDO's de segunda ordem, basta fazer por partes. Suponha que y'_0 *e* y''_0 sejam conhecidas para a seguinte EDO $y'' + y' + y = f(t)$.

EDO = $y'' + y' + y = f(t)$. Basta fazermos uma redução de ordem, ou seja, ao invés de uma EDO de segunda ordem, resolvamos duas EDO's de primeira ordem:

$$y' = z$$
$$y'' = z'$$

O que nos leva a dois sistemas recorrentes:

$$z = y'$$
$$z' = y'' = f(t) - y' - y$$

Em notação indicial:

$$z_i = y'_i$$
$$z'_i = f(t_i) - y'_i - y_i$$

Dentro do método de Euler:

$$y'_{i+1} = y_i + \Delta t y'_i$$
$$y''_{i+1} = y'_i + \Delta t y''_i = y'_i + \Delta t f(t_i) - y'_i - y_i$$

Ok, agora implementemos o código acima em python, utilizando $f(t) = sen(e^x)$, $y_0 = 10$ e $y'_0 = \pi$:

```
#EDO y''+y'+y=sen(e**x)
import math

e=math.e
y=[]
ydot=[]
ydotdot=[]

y0=10
y0dot=math.pi

y.append(y0)
ydot.append(y0dot)

pi=0
pf=10
n=10**5
delta=(pf-pi)/n

for i in range(n):
    y.append(y[i]+delta*ydot[i])
    ydot.append(ydot[i]+delta*
    (math.sin(i*delta)-ydot[i]-y[i]))
```

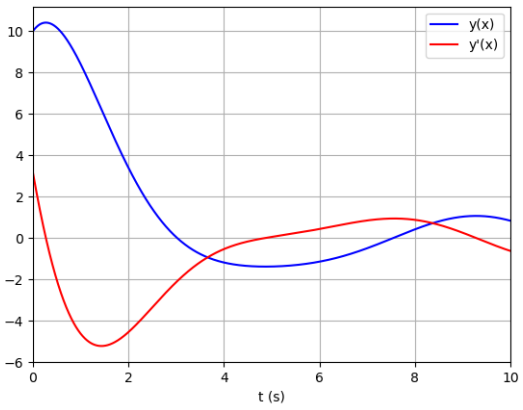


Fig. 3: Plotagem dos dados obtidos com o algorítimo acima para a função e sua derivada.

Agora meu caro leitor, você está apto a resolver quase todos os problemas da graduação numericamente, mas vale uma ressalva, por mais que o método de Euler seja extremamente fácil, ele é meio inútil pois diverge do resultado real muito facilmente, exigindo sempre que o valor de "n"seja muito elevado, o que aumenta o custo computacional. Para implementações simples ele serve muito bem, mas veremos a seguir que existem métodos razoavelmente claros que nos levam a resultados mais preciso e com menor

custo computacional. Resolva os seguintes problemas a seguir considerando $y_0 = y_0' = 1$:

- y''+y'=o
- y''+y'=e^y
- y''+y'=sen(ty)
- y''+y'=t
- y'''+y''+y'=sen(y), $[y_0 = y_0' = y_0'' = 1]$
- y'''+sen(t)y'+y=o, $[y_0 = y_0' = y_0'' = 1]$

11.2 *Runge Kutta de Quarta Ordem*

Continuaremos nossa jornada com um método mais robusto, estudaremos agora o Runge Kutta de quarta ordem. A ideia para todos os métodos chamados de "Runge Kutta de Quarta Ordem"é a mesma, então conhecer a dedução de um deles deve ser suficiente para que você possa se virar com quaisquer alterações necessárias.

Assim como para Euler, o método estudado vale para EDO's de primeira ordem, ou seja, para problemas de ordem superior você deverá utilizar a redução de ordem e, caso não tenha ficado claro anteriormente não se preocupe, aqui faremos passo a passo novamente).

Novamente, assim como para Euler expandiremos nossa função em Taylor, contudo agora truncaremos nosso somatório na quarta ordem e, já que o leitor está acostumado com a notação indicial introduzida no capítulo anterior, todas as deduções a partir de agora usarão a mesma.

Definiremos:

$$\frac{dy}{dt} = f(t, y) \tag{11}$$

A expansão em Taylor:

$$\begin{aligned} y_{i+1} = y_i + \frac{dy}{dt}\Delta t + \frac{1}{2!}\frac{d^2}{dt^2}\Delta t^2 + \\ + \frac{1}{3!}\frac{d^3}{dt^3}\Delta t^3 + \frac{1}{4!}\frac{d^4}{dt^4}\Delta t^4 + O^5 \end{aligned} \tag{12}$$

Reescrevendo o somatório acima partir da definição 11:

$$\begin{aligned} y_{i+1} = y_i + f(x_i, y_i)\Delta t + \frac{1}{2!}f'(x_i, y_i)\Delta t^2 + \\ + \frac{1}{3!}f''(x_i, y_i)\Delta t^3 \frac{1}{4!}f'''(x_i, y_i)\Delta t^4 \end{aligned} \tag{13}$$

Agora é o momento do "pulo do gato"meu caro leitor, nosso objetivo é escrever a equação acima como:

$$y_{i+1} = y_i + (a_1 k_1 + a_2 k_2 + a_2 k_3 + a_2 k_4)\Delta t \tag{14}$$

Como o produto das constantes é uma constante, podemos reescrever a equação acima como:

$$y_{i+1} = y_i + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)\Delta t \tag{15}$$

Para que 15 respeite 13 é necessário que:

$$\begin{aligned} k_1 &= f(x_i, y_i) \\ k_2 &= f(x_i + \frac{\Delta t}{2}, y_i + \frac{k_1 \Delta t}{2}) \\ k_3 &= f(x_i + \frac{\Delta t}{2}, y_i + \frac{k_2 \Delta t}{2}) \\ k_4 &= f(x_i + \Delta t, y_i + k_3 \Delta t) \end{aligned} \tag{16}$$

Novamente, agora você está apto a resolver quaiser problemas com EDO's de primeira ordem. Vamos entender então o algorítimo para uma EDO qualquer do tipo "$y' + y = t$", "$y_0 = 2.5$"e "$\Delta t = 0.5$":

enhanced,breakable Nesse problema: $f(t_i, y_i) = -y_i + t_i$, utilizando 16:

$$y_{0+1} = y_i = y_0 + \frac{1}{6}(k_1 + 2k_2 + 2K_3 + k_4)0.5$$

$$k_1 = f(x_0, y_0) = -y_0 + 0$$

$$k_2 = -(2.5 + 0.25k1) + 0 + 0.25$$

$$k_3 = -(2.5 + 0.25k2) + 0 + 0.25$$

$$k_4 = -(2.5 + 0.5k1) + 0 + 0.5$$

$$y_1 = y_0 + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)0.5$$

E assim recursivamente até o ponto final.

E no Python, como fica isso tudo? Vejamos a seguir!

```
y0=5 #valor inicial

t0=0 #ponto inicial
tf=10 #ponto final

n=10**5 #numero de pontos
dt=(tf-t0)/n

y=[]
y.append(y0)

def func(a,b): #a=y e b=t
    return(-a+b)

for i in range(n):
    t=i*dt #t
    k1=func(y[i],t)
    k2=func(y[i]+k1*dt/2,t)
    k3=func(y[i]+k2*dt/2,t+dt/2)
    k4=func(y[i]+k3*dt,t+dt)
    y.append(y[i]+(dt/6)*(k1+k4+2*(k2+k3)))
```

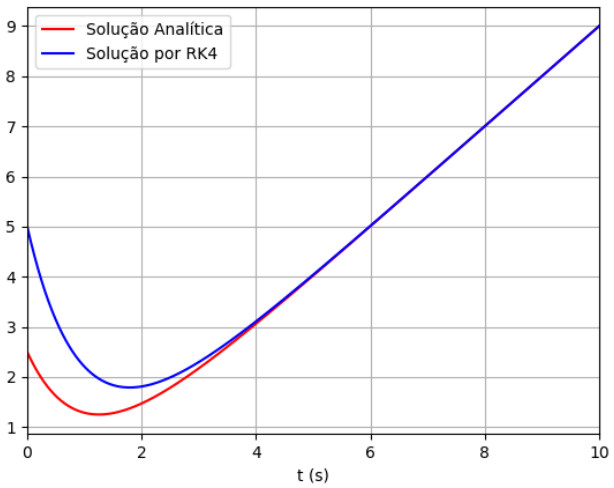


Fig. 4: Comparação entre o as soluções analítica e por Runge Kutta.

Baseado no exemplo anterior, resolva as seguintes edos para condições iniciais diversas.

- $y' + t = y$
- $x^2y' - y = sen(t)$
- $y' + y = sin(y)$
- $y'^2 + y^2 = e^y$

Para EDO's de segunda ordem, usaremos o truque de redução de ordem novamente. Agora definiremos:

$$\begin{aligned} \frac{dy}{dx} &= f(t,y) \\ \frac{d^2y}{dx^2} &= f(t,y') \end{aligned} \tag{17}$$

Aplicando 17 em 16:

$$\begin{aligned} k_{11} &= f(x_i, y_i) \\ k_{21} &= f(x_i + \frac{\Delta t}{2}, y_i + \frac{k_1 \Delta t}{2}) \\ k_{31} &= f(x_i + \frac{\Delta t}{2}, y_i + \frac{k_2 \Delta t}{2}) \\ k_{41} &= f(x_i + \Delta t, y_i + k_3 \Delta t) \\ \\ k_{12} &= f(x_i, y_i') \\ k_{22} &= f(x_i + \frac{\Delta t}{2}, y_i + \frac{k_1 \Delta t}{2}) \\ k_{32} &= f(x_i + \frac{\Delta t}{2}, y_i + \frac{k_2 \Delta t}{2}) \\ k_{42} &= f(x_i + \Delta t, y_i + k_3 \Delta t) \end{aligned} \tag{18}$$

De modo que:

$$\begin{aligned} y_{n+1} &= y_n + \frac{\Delta t}{6}(k_{11} + 2k_{21} + 2k31 + k_{41}) \\ y'_{n+1} &= y'_n + \frac{\Delta t}{6}(k_{12} + 2k_{22} + 2k32 + k_{42}) \end{aligned} \tag{19}$$

Agora, utilize seus conhecimentos já estabelecidos e praticados com Euler sobre redução de ordem e resolva os problemas abaixo pelo método de Runge-Kutta aprendido (utilize condições iniciais de sua escolha):

- y''+y'=o
- y''+y'=e^y
- y''+y'=sen(ty)
- y''+y'=t
- y'''+y''+y'=sen(y), $[y_0 = y'_0 = y''_0 = 1]$
- y'''+sen(t)y'+y=o, $[y_0 = y'_0 = y''_0 = 1]$

11.3 *Verlet*

O último método recorrente que estudaremos será o método de Verlet, um método especialmente útil para resolver EDO's de segunda ordem. A dedução do método vai de encontro ao que já vimos acima, ou seja, partiremos da expansão em série de Taylor da função, mas o método de

Verlet possui uma característica muito interessante, trata-se de um método adiabático! Bem, mas o que isso quer dizer? Essencialmente o método se comportará de maneira equivalente para "$t > 0$ *e* $t < 0$", além do fato de que para grandes "Δt"o método também convergirá! Vamos à dedução:

$$\begin{aligned} f(t+\Delta t) &= f(t) + f'(t)\Delta t + \frac{1}{2}f''(t)\Delta t^2 + \frac{1}{6}f'''(t)\Delta t^3 + O^4 \\ f(t-\Delta t) &= f(t) - f'(t)\Delta t + \frac{1}{2}f''(t)\Delta t^2 - \frac{1}{6}f'''(t)\Delta t^3 + O^4 \end{aligned} \tag{20}$$

Somando as duas equações acima:

$$f(t+\Delta t) + f(f-\Delta t) = 2f(t) + f''(t)\Delta t^2 + O^4 \tag{21}$$

Em notação indicial:

$$f_{i+1} = -f_{i-1} + 2f_i + f_i''\Delta t^2 \tag{22}$$

Repare meu caro leitor, que aqui precisaremos de y_0 *e* y_1. Geralmente o problema nos dará "y_0 *e* y_0'"pois tratam-se de EDO's de segunda ordem. Então, cabe a você encontrar o y_1 com um método de sua escolha e, a partir daí dar sequência. Vejamos o algoritmo em ação para o problema a seguir "$y'' + y = 0$".

Para o problema acima $y_i'' = -y_i$:

$$y_1 = y_0 + \Delta t y_0'$$

$$\begin{aligned} n = 2 &\longrightarrow y_3 = -y_1 + 2y_2 + \Delta t^2 f_2'' \\ n = 3 &\longrightarrow y_4 = -y_2 + 2y_3 + \Delta t^2 f_3'' \\ n = 4 &\longrightarrow y_5 = -y_3 + 2y_4 + \Delta t^2 f_4'' \\ &\qquad\qquad\qquad\qquad\qquad\qquad ... \end{aligned}$$

Dos algorítmos vistos até agora esse é essencialmente o mais simples, não? Agora vamos resolver o mesmo problema no Python!

```
#CODIGO PARA VERLET

y0=1 #condicoes iniciais
y0dot=0

y=[y0]

tf=10
ti=0
n=5*10**1
dt=(tf-ti)/n

def func(a): #y''
    return(-a)

y.append(y0+dt*y0dot) #primeiro passo por euler

for i in range(1,n):
    y.append(-y[i-1]+2*y[i]+(dt**2)*func(y[i]))
```

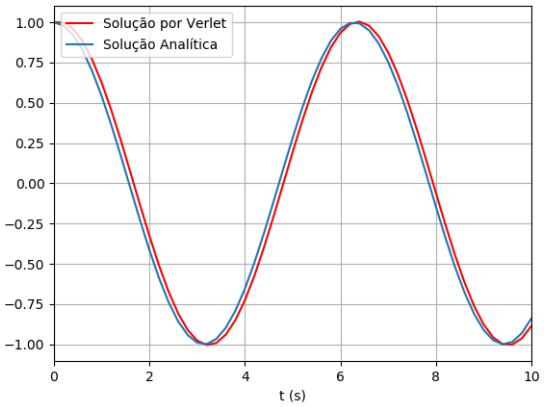


Fig. 5: COmparação entre a solução analítica e numérica via Verlet. Repare que Verlet converge extremamente rápido, para n=50 a solução é quase igual a analítica.

Agora que você já detém todos os conhecimentos necessários para resolver EDO's de segunda ordem sem redução de ordem, resolva os problemas abaixo:

- y''+y=sin(t)
- y''+$y\frac{k}{m}$=o
- y''+y^2=sin(t)y
- y''+y=$cos^2(t)$

12 TRANSFORMADA DE FOURIER DISCRETA

Nessa seção falaremos sobre a Transformada Discreta de Fourier, assim como na seção anterior primeiro será mostrada uma dedução teórica do

método e depois a implementação no Python. Importante ressaltar que esse capítulo é interessante para o entendimento do algorítimo, mas no futuro aprenderemos a utilizar o "FFT", um pacote bem mais rápido que nossa implementação pronto no NumPy. Ok, definiremos alguns pontos importantes.

Primeiro definiremos os pontos no eixo das coordenadas como "$x_j = x_0 + j\Delta x$", condições de contorno periódicas "$f(x+L) = f(x)$" e o comprimento "$L = N\Delta x$".

Será importante definir também que "$f(x) = \sum_n^N c_n e^{ik_n x}$" e a coordenada recíproca "$k_n = \frac{2\pi}{L} n$", as definições são portanto:

$$
\begin{aligned}
&\longrightarrow x_j = x_0 + j\Delta x \\
&\longrightarrow f(x+L) = f(x) \\
&\longrightarrow L = N\Delta x \\
&\longrightarrow f(x) = \sum_n^N c_n e^{ik_n x} \\
&\longrightarrow k_n = \frac{2\pi}{L} n
\end{aligned}
$$

Dito isso, vamos façamos algumas contas para determinar "c_n"via truque de fourier:

$$
c_n = \frac{1}{L}\int_{x_0}^{x_0+L} f(x) e^{-ik_n x} dx \approx \frac{e^{-ik_n x_0}}{N} \sum_{j=0}^{N-1} f(x_j) e^{-ik_n j\Delta x} \tag{23}
$$

Repare que podemos escrever "$k_n \Delta x = \frac{2n\pi}{N}$, passemos também a equação acima para notação indicial:

$$
c_n = \frac{e^{-ik_n x_0}}{N} \sum_{j=0}^{N-1} f_j e^{-2\pi i \frac{jn}{N}} \tag{24}
$$

Tipicamente apenas a parte em vermelho da equação acima é implementada nos programas, é importante sabermos disso pois quando usarmos os pacotes no Numpy e no SciPy colocaremos a parte preta na mão, para evitar possíveis erros (repare que a ausência desses termos possui duas consequências, a primeira é que se "$x_0 \neq 0$"haverá uma defasagem de fase entre a resposta real e a obtida pelos métodos usuais, já a segunda é a normalização provocada pela ausência do termo "$\frac{1}{N}$").

No final das contas, nossa transformada de fourier ficou do tipo:

$$
F_n = \sqrt{N} c_n = \frac{e^{-ik_n x_0}}{\sqrt{N}} \sum_{j=0}^{N-1} f_j e^{-2i\pi \frac{jn}{N}}. \tag{25}
$$

Na transformada acima utilizamos "$\sqrt{N}$", não foque muito nisso, poderíamos utilizar tranquilamente "N"ou "0", o ponto é que o produto da normalização da transformada e da sua inversa seja igual a "$\frac{1}{N}$". A transformada inversa:

$$
f_j = \frac{1}{\sqrt{N}} \sum_n e^{ik_n x_0} F_n e^{2\pi i \frac{jn}{N}}. \tag{26}
$$

Trata-se de um algorítimo consideravelmente mais complexo que os anteriores, estão ao invés de fazermos um pseudo-código primeiro e depois

o código em si, como foi feito anteriormente, aqui partiremos direto já paro o Python:

```
#CODIGO PARA FFT EM PYTHON

import math

pi=math.pi
e=math.e

def func(x): #funcao a qual desejamos a transformada
    return(math.e**x)

N=2**11+1 #numero de pontos
x0=0 #ponto inicial
xf=100 #ponto final
L=xf-x0 #comprimento

dx=L/N
x=[L*i/N-x0 for i in range(N)] #vai de 0 ate 10, com intervalo L/n

k=[2*pi*i/L for i in x]
dk=k[1]-k[0]
k=k[:-1] #retiramos o ultimo ponto devido as condicoes de
    #periodicidade, repare que ele e importante para o dk,
    #mas deve ser retirado para o calculo da transformada

#agora faremos a transformada
F=[]

for n in range(len(k)):
    #print(n)
    cte=e**(-1j*k[n]*x[0])*(1/(N**0.5)) #termo constante
    Fn=0
    for j in range(N-1):
        Fn+=func(x[j])*e**(-2*1j*pi*j*n*(1/N))
    Fn*=cte
    F.append(Fn)
```

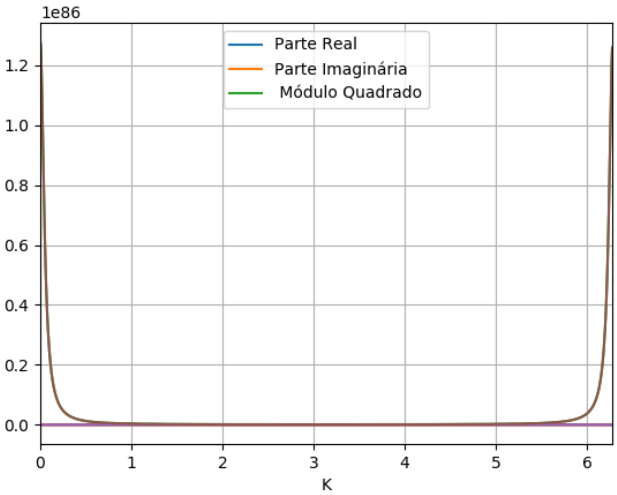


Fig. 6: Transformada de fourier de uma exponencial, a solução analítica https://en.wikipedia.org/wiki/Dirac_delta_function

Agora, para fixar as ideias apresentadas faça os seguintes exercícios:

- Faça a transformada inversa para o código acima;
- Escreva uma função para transformada de fourier;
- Escreva uma função que realize a transformada inversa de fourier.

Parte III. Pacotes

Matplotlib

13 MATPLOTLIB

O Matplotlib é um pacote para visualização de dados em Python. O pacote é extremamente simples de ser usado e aqui teremos uma introdução para os capítulos seguintes. É importante aprender a utilizar o matplotlib (nesse capítulo, mais especificamente o matplotlib.pyplot) pois em diversos momentos será de nosso interesse visualizar a solução de uma EDO, os auto vetores e auto valores de uma matriz, plotar um histograma e tudo mais.

Toda documentação do pacote pode ser encontrada no site oficial deles (https://matplotlib.org/), recomendo fortemente que o leitor entre no site e veja alguns exemplos e possibilidades que o pacote oferece visto que durante o curso utilizaremos poucos dos inúmeros recursos presentes no pacote.

Como trata-se de um pacote bem simples, escreverei os códigos e apresentarei os resultados de pouco a pouco, começaremos com um caso bem simples e terminaremos com casos bem completos. A documentação dessa classe está em https://matplotlib.org/tutorials/introductory/pyplot.html.

Primeiro, façamos um gráfico bem simples:

```
import matplotlib.pyplot as plt #sera a chamada usual
import math as m

n=10**6

x=[i*2*m.pi/n for i in range(n)] #de 0 ate 2pi com passo 0.01
y=[]
for i in x:
    y.append(m.sin(i)) #criamos uma lista [sin(0),...,sin(2*pi)]
plt.plot(x,y) #o primeiro eixo e sempre o horizontal
plt.show() #aparece o grafico na tela
plt.savefig("nume.png") #salva a figura no atual diretorio
```

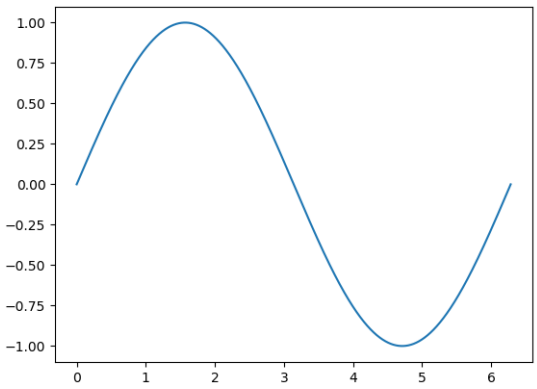


Fig. 7: Grafico do seno, escrito acima

Toda a evolução será sobre o primeiro código. Agora adicionaremos eixos, título, tirar as bordas brancas na vertical e na horizontal e adicionar legenda.

```
import matplotlib.pyplot as plt #sera a chamada usual
import math as m

n=10**6

x=[i*2*m.pi/n for i in range(n)] #de 0 ate 2pi com passo 0.01
y=[]
for i in x:
    y.append(m.sin(i)) #criamos uma lista [sin(0),...,sin(2*pi)]

plt.plot(x,y) #o primeiro eixo e sempre o horizontal

plt.title("Sin(x)") #adiciona "Sin(x)" sobre o gráfico

plt.xlabel("X") #escreve "X" no eixo horizontal
plt.ylabel("Y") #escreve "Y" no eixo vertical
plt.legend(["Grafico Sin(x)"],loc="upper right") #adiciona
                        #legenda no canto superior direito
plt.xlim(0,2*m.pi) #apresenta o eixo x de 0 a 2pi
plt.ylim(-1,1) #apresenta o eixo y de -1 ate 1
```

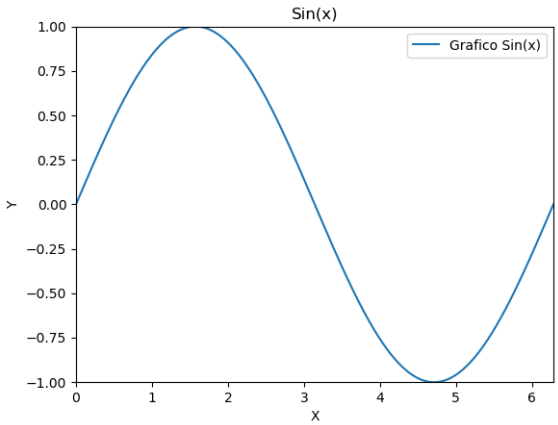


Fig. 8: Grafico do seno novamente, mas com algumas melhorias.

As últimas modificações, adicionaremos grades ao fundo do gráfico, mudaremos a cor e o formato com que as linhas são exibidas e plotaremos mais de um gráfico no mesmo espaço:

```
import matplotlib.pyplot as plt #sera a chamada usual
import math as m

n=10**6

x=[i*2*m.pi/n for i in range(n)] #de 0 ate 2pi com passo 0.01
y1=[]
```

```
y2=[]
y3=[]

for i in x:
    y1.append(m.sin(i)) #criamos uma lista [sin(0),...,sin(2*pi)]
    y2.append(2+m.cos(i)) #vetor de cossenos
    y3.append(4+m.sin(i)*m.cos(i)) # produto entre senos e cossenos

plt.plot(x,y1,'k') #k-¿preto
plt.plot(x,y2,'y') #y-¿amarelo
plt.plot(x,y3,'r') #r-¿vermelho

plt.title("Sin(x)") #adiciona "Sin(x)" sobre o gráfico
plt.xlabel("X") #escreve "X" no eixo horizontal
plt.ylabel("Y") #escreve "Y" no eixo vertical
plt.legend(["Sin(x)","Cos(x)","Sin(x)Cos(x) "]
           ,loc="upper right") #adiciona legendas em ordem de plot

plt.grid(True) #adiciona grade no fundo

plt.xlim(0,2*m.pi) #apresenta o eixo x de 0 a 2pi
plt.ylim(-1,6) #apresenta o eixo y de -1 ate 1

plt.show() #mostra a figura
```

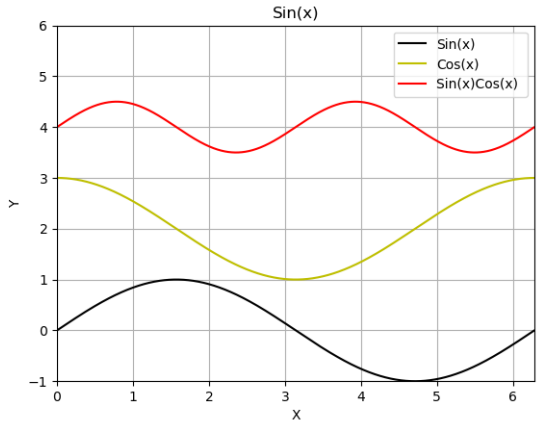


Fig. 9: Gráfico do seno, cosseno e o produto entre eles. Peço que o leitor procure sobre possíveis formatações do gráfico em https://matplotlib.org/3.2.1/api/_as_gen/matplotlib.pyplot.plot.html.

Como nosso interesse é em plotar figuras que nos permitam visualizar os resultados futuros, a qualidade do último exemplo já é suficientemente boa.

Por último nos concentraremos em plotar vários gráficos separados com o comando "subplot"e depois criar várias viguras através do mesmo programa com o comando "figure".

Subplot nos permitirá plotar gráficos em painéis, imagine que em uma figura haverá quatro diferentes plotes distribuídos igualmente, ou seja, duas linhas e duas colunas, para esse caso escreveremos "plt.subplot(2,2,x)",

"x"indica qual dos elementos na figura será escrito a seguir, para o primeiro gráfico (primeira linha e primeira coluna) ficará "plt.subplot(2,2,1)", para o gráfico na segunda linha e terceira coluna teremos "plt.subplot(2,2,3)"e assim vai.

Para que a ideia fique mais clara, escrevamos um código:

```
import matplotlib.pyplot as plt
from math import e

x=[-5+10*i/10**4 for i in range(10**4)] #vai de -5 a 5
y1=[i**2 for i in x] #parabola
y2=[i for i in x] #reta
y3=[e**(-i**2) for i in x] #gaussiana
y4=[e**-i for i in x]

plt.subplot(2,2,1) #de quatro plots, esse sera o primeiro
plt.plot(x,y1,'r') #vermelho

plt.subplot(2,2,2) #de quatro plots, esse sera o segundo
plt.plot(x,y2,'b') #azul

plt.subplot(2,2,3) #de quatro plots, esse sera o terceiro
plt.plot(x,y3,'y') #amarelo

plt.subplot(2,2,4) #de quatro plots, esse sera o quarto
plt.plot(x,y4,'k') #preto

plt.tight_layout() #utilizaremos esse comando para
    #evitar possiveis desformatacoes
plt.show()
```

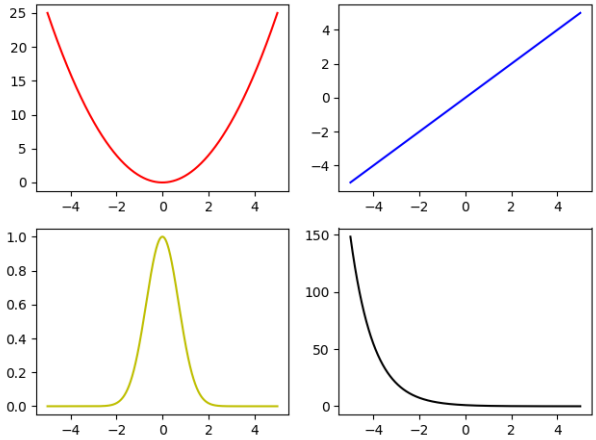


Fig. 10: Gráficos na ordem de plotagem, para mais informações vá em https://matplotlib.org/3.2.1/api/_as_gen/matplotlib.pyplot.subplots.html.

Podemos ainda complementar os plots para que fiquem mais apresentáveis:

```
import matplotlib.pyplot as plt
from math import e

x=[-5+10*i/10**4 for i in range(10**4)] #vai de -5 a 5
y1=[i**2 for i in x] #parabola
y2=[i for i in x] #reta
y3=[e**(-i**2) for i in x] #gaussiana
y4=[e**-i for i in x]

plt.subplot(2,2,1) #de quatro plots, esse sera o primeiro
plt.plot(x,y1,'r') #vermelho
plt.xlim(-5,5)
plt.ylim(min(y1),max(y1)) # do minimo ao maximo de y
plt.legend(["Parabola"],loc="upper right")
plt.grid(True)

plt.subplot(2,2,2) #de quatro plots, esse sera o segundo
plt.plot(x,y2,'b') #azul
plt.xlim(-5,5)
plt.ylim(min(y2),max(y2)) # do minimo ao maximo de y
plt.legend(["Reta"],loc="upper left")
plt.grid(True)

plt.subplot(2,2,3) #de quatro plots, esse sera o terceiro
plt.plot(x,y3,'y') #amarelo
plt.xlim(-5,5)
plt.ylim(min(y3),max(y3)) # do minimo ao maximo de y
plt.legend(["Gaussiana"],loc="best")
plt.grid(True)

plt.subplot(2,2,4) #de quatro plots, esse sera o quarto
plt.plot(x,y4,'k') #preto
plt.xlim(-5,5)
plt.ylim(min(y4),max(y4)) # do minimo ao maximo de y
plt.legend(["Exponencial Negativa"],loc="lower right")
plt.grid(True)

plt.tight_layout() #utilizaremos esse comando para

    #evitar possiveis desformatacoes
plt.show()
```

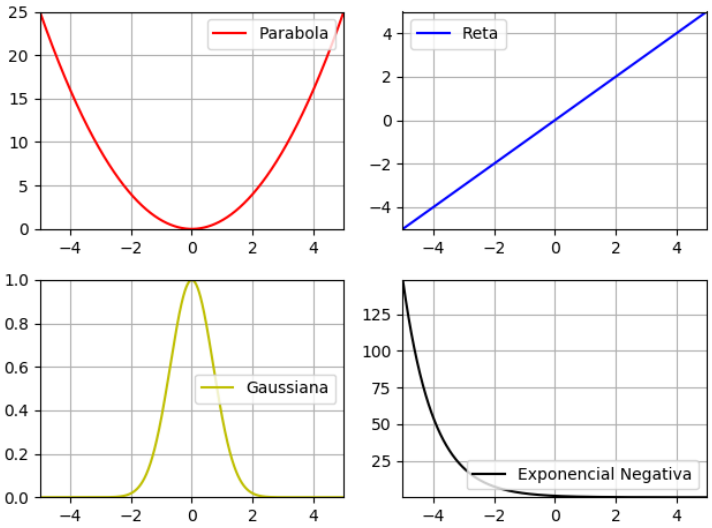


Fig. 11: Matplotlib aceita notação do latex, para usar é bem simples, basta colocar um "r"antes da string e utilizar a notação usual do instrumento por exemplo plt.title(r"$e \wedge x$").

Figure nos permitirá criar várias imagens dentro do mesmo código, repare que até agora tudo que plotamos era em apenas uma figura (independente do número de plotes diferentes). Agora você aprenderá a criar um código que gera várias figuras e, cada figura pode ter um plote ou mais, todas as regras vistas até aqui continuam válidas, o código terá a seguinte forma:

```
import matplotlib.pyplot as plt
from math import e
import math

n=10**5 #numero de pontos
x=[-5+i/n for i in range(n)]
y1=[i**2 for i in x]
y2=[e**(-i**2) for i in x]

plt.figure() #gera a nova figura em branco
plt.plot(x,y1,'r')
plt.title("Primeira Figura Gerada")
plt.tight_layout() #tenha o costume de sempre declarar esse camarada
plt.savefig("figura1.png") #salvar local atual


plt.figure() #gera a nova figura em branco
plt.plot(x,y2,'k')
plt.title("Segunda Figura Gerada")
plt.tight_layout() #tenha o costume de sempre declarar esse camarada
plt.savefig("figura2.png") #salvar local atual

plt.figure() #nova figura
plt.subplot(3,1,1)
```

```
plt.plot(x,y1,'y')

plt.subplot(3,1,2)
plt.plot(x,y2,'b')

plt.subplot(3,1,3)
y3=[math.sin(i) for i in x]
plt.plot(x,y3,'k')


plt.tight_layout() #tenha o costume de
#sempre declarar esse camarada, principalmente em subplots
plt.savefig("figura3.png") #salvar local atual
```

As figuras resultantes em ordem são:

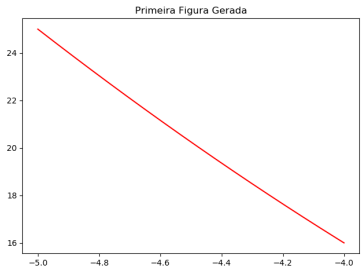


Fig. 12: Primeira figura.

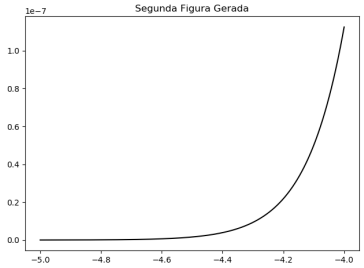


Fig. 13: Segunda figura.

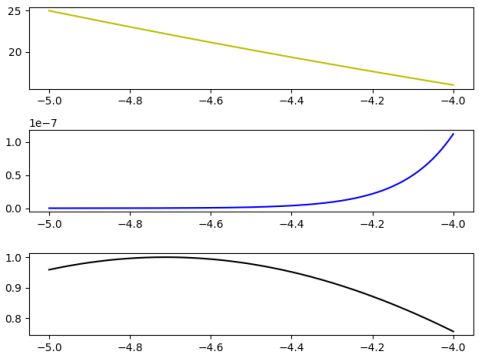


Fig. 14: Terceira figura.

Agora você já conhece o suficiente para fazer imagens extremamente bonitas e clara a partir de um potencial resultado, mas ainda insisto que você leia e analise os exemplos com cuidado presentes na documentação https://matplotlib.org/3.2.1/tutorials/introductory/sample_plots.html. Com uma leitura rápida, mas atenta tenho certeza que você será capaz de fazer histogramas em uma, duas e três dimensões, plotes em três dimensões, vídeos, gráficos vetoriais e muito mais. Algumas ferramentas presentes no Matplotlib são muito interessantes e não faz sentido mostrá-las aqui, mas que em alguns exemplos daqui pra frente posso utilizar (caso eu utilize, fornecerei o link para a documentação).

Para esse capítulo refeça todos os exemplos variando as funções e, depois volte nos programas desenvolvidos no capítulo anterior para solução de EDO's e confira os resultados.

NumPy

14 NUMPY

Numpy vem de "Numerical Python" e é um pacote fundamental para qualquer estudante de exatas pois possui inúmeras implementações para trabalhar com vetores e matrizes, álgebra linear, transformada de fourier, séries, números aleatórios, funções especiais e muito mais.

Além disso, todas as implementações presentes no NumPy são de altíssimo desempenho e no geral ganham de lavada do Python puro, isso será lembrado o tempo todo a partir de agora, tentaremos sempre escrever nossos códigos com o mínimo de estrutura condicional e de repetição possível pois geralmente elas podem ser substituídas pelo NumPy.

Ao longo da leitura mostrarei várias vezes o ganho de desempenho que você terá ao utilizar o NumPy com o objetivo de te convencer a sempre utilizá-lo.

Mas então, por que NumPy é tão mais rápido? Por dois motivos, o primeiro é que ele armazena suas lista em blocos de memória homogêneos (enquanto o Python aponta cada variável num endereço diferente), o segundo motivo é que muitas das implementações no NumPy estão escritas em C, então você escreve o código, ele interpreta pra C e executa o comando. Mais informações podem ser encontradas na documentação do próprio https://numpy.org/.

14.1 *Arrays*

Trata-se da estrutura primordial de dados dentro do NumPy, são mais rápida e simples de trabalhar do que as listas, agora veremos os comando essenciais:

```
import numpy as np #primeiro, chamamos o pacote

x1=np.arange(0,10,0.05) #vai de 0 a 10-0.05 com passo 0.05
x2=np.arange(0,10,100) #vai de 0 a 10 em 100 passos
y1=np.cos(x1) #podemos fazer funções de listas

m1=np.zeros((10)) #vetor com dez elementos nulos
m2=np.zeros((100,100)) #matriz 100x100 de zeros

A=np.sum(x1) #retorna a soma dos elementos de x1

l=[1,2,3] #lista
lnew=np.array(l) #transforma l em array

t=[1,2,3]
T=lnew+l #retorna uma array

a=np.arange(0,10,0.01)
b=np.linspace(-10,100,len(a))
c=a+b #diferente da lista, aqui os vetores que se somam
      #precisam ter o mesmo tamanho e, c[i]=a[i]+b[i]
```

```
#retornara uma lista do tipo y1=[cos(0),cos(0.05)...,cos(9.95)]

np.savetxt('arquivo.txt',a) #salva a array "a" no diretorio atual
                            #com nome arquivo e formato .txt
```

Repare que para acessar elementos de uma array é igual a acessar elementos de uma lista (por exemplo, caso eu queira o elemento 10,4 de uma matriz c[10,4] ou c[10][4]), enfatizo ainda a necessidade do leitor ler a documentação do NumPy e testar os casos acima, pois tratam-se de comandos essenciais para todos os códigos que virão a seguir.

Vamos fazer juntos um código que gera uma matriz, onde a primeira coluna possui elementos de seno, a segunda cosseno e a terceira a soma das outras duas colunas:

```
import numpy as np

n=10**3 #numero de elementos
M=np.zeros((n,3)) #matriz de n linhas e tres colunas
x=np.linspace(0,2*np.pi,n) #0 a 2pi em n pontos

M[:,0]=np.sin(x) #povoando todas a linhas da primeira
                 #coluna com seno(x)

M[:,1]=np.cos(x) #povoando a segunda coluna

M[:,2]=np.cos(x)+np.sin(x) #ultima coluna
```

Mais exemplos em https://cs231n.github.io/python-numpy-tutorial/.

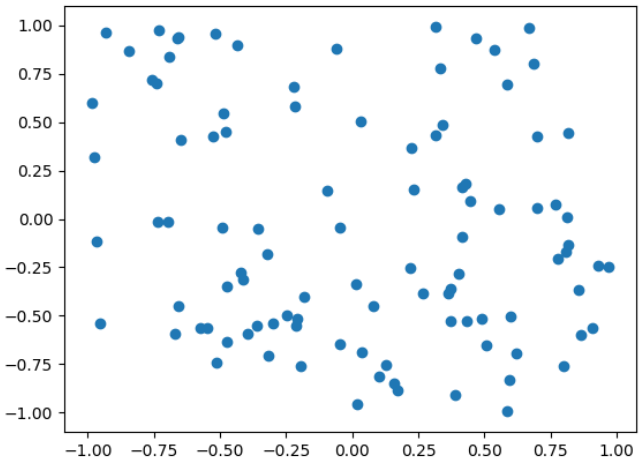
14.2 *Numeros Aleatórios*

Aqui veremos o básico sobre o pacote de números aleatório dentro do Numpy, como sempre, recomendarei que você leia um pouco sobre a biblioteca na documentação do pacote em https://numpy.org/doc/1.18/reference/random/index.html.

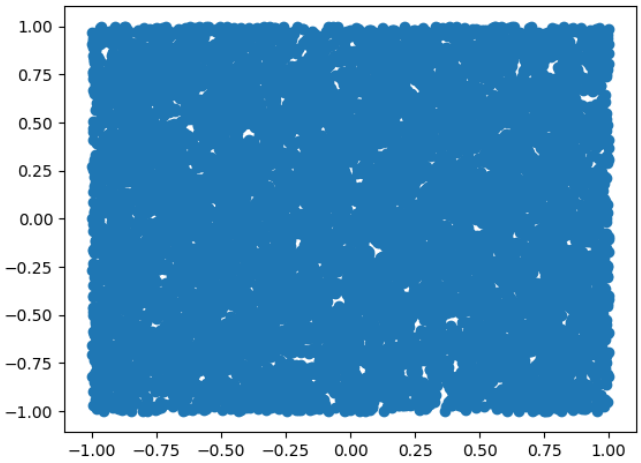
A seção será bem curta, faremos alguns código juntos (como o método de Monte Carlo), plotaremos os resultados, e analisaremos o ganho de desempenho do "numpy.random()"em relação ao "random"intrínseco ao Python.

Mãos a obra:

```
import numpy as np
import matplotlib.pyplot as plt
from numpy import random as rand

n=10**2
x=rand.random(n) #vetor de n numeros aleatorios entre 0 e 1
x=2*x-1 #novo vetor, com numeros entre -1 e 1

y=-1+2*rand.random(n) #vetor com numeros entre -1 e 1

plt.scatter(x,y,'b')
```



Se fizermos n=10^5:



Agora que já sabemos gerar números aleatório via "numpy.random", vamos calcular a integral "$\int_0^1 \sqrt{1-x^2}$" via método de Monte Carlo (farei o código com random do Python e o random do Numpy, depois irei comparar o tempo em cada um dos metodos).

```
import numpy as np
import matplotlib.pyplot as plt
import random as random˙python

n=10**3

#Python convencional

x=[random˙python.random() for i in range(n)]
y=[(1-i**2)**0.5 for i in x]
```

```
    media=sum(y)/len(y) #somo elementos de y
                        #depois divido pelo numeoro de elementos


    #NumPy
    x=np.random.random(100)
    y=np.sqrt(1-x**2)
    media=np.sum(y)/len(y)
```

O gráfico abaixo mostra o tempo gasto pelos dois métodos:

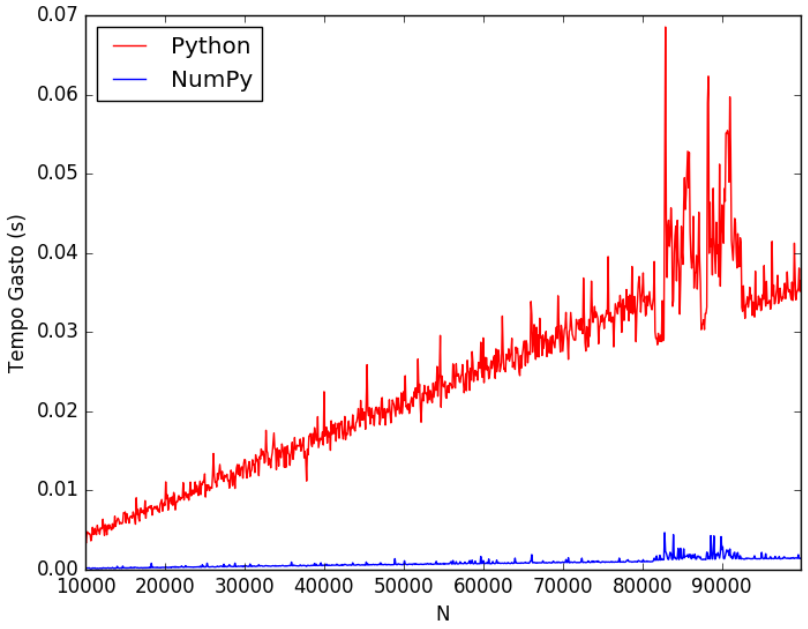


Fig. 15: Tempo gasto por número de amostras.

Em média, o tempo gasto pelo NumPy é vinte vezes menor que o tempo gasto pelo Python. Alguns exercícios interessantes:

- Faça um gráfico que analise a traxa de convergência do resultado de uma integral qualquer via método de Monte Carlo;
- Resolva o problema do caminho aleatório (exemplo https://www.geeksforgeeks.org/random-walk-implementation-python/);
- Calcule a área de uma esfera de raio unitário via Método de Monte Carlo (https://plotly.com/python/random-walk/#random-walk-in-1d);
- Implemente o método de Monte Carlo para a seguinte integral: $\int_0^1 \int_0^1 \int_0^1 x^2 ln(y+1) sin^2(z) + xyz \; dxdydz$.

14.3 *Transformada de Fourier*

Agora falaremos sobre o sub-pacote "FFT"do NumPy para realizar transformadas de Fourier; como a execução e montagem do código é bem simples,

escreveremos juntos um código bem otimizado com duas funções (uma para a transformada e outra para a transformada inversa de Fourier).

Vamos ao código:

```
import numpy as np
def pfft(x, y): #definimos a funcao transformada
    dx=x[1]-x[0] #encontramos o delta k
    N=len(x) #numero de pontos
    k=np.linspace(-np.pi/dx, np.pi/dx, N+1) #dominio em k
    k=k[:-1] #tiramos o ultimo ponto, devido as c.c periodicas
    yk=np.fft.fft(y, norm='ortho')#transformada de fourier
    yk*=np.exp(-1j*k*x[0]) #adicionamos a fase
    return k, yk
def ipfft(k, yk): #transformada inversa de fourier
    dk=k[1]-k[0]
    N=len(k)
    x=np.linspace(-dk/np.pi, dk/np.pi, N+1)
    x=x[:-1]
    y=np.fft.ifft(yk, norm='ortho') #transformada inversa
    y*=np.exp(+1j*k*x[0]) #adicionamos a fase
    return x, y

x1=np.linspace(-15*np.pi, 15*np.pi, 256 + 1)
x1=x1[:-1]
y1=np.sin(x1)
k1, yk1=pfft(x1, y1) #retorna da derivada e
                     #o dominio no espaço reciproco
```

Do código acima vale comentar que ”ortho”indica a normalização da transformada, sempre que possível utilize intervalos com expoentes de ”2” (2,4,8,16,32,64...1024,2048,...,2^n)e , o produto que fazemos após a transformada e após a inversa trata-se da constante de fase que discutimos anteriormente (eq. 24). Na leitura da documentação você encontra como oo método é implementado, alguns exemplos e comentários (https://numpy.org/doc/stable/reference/routines.fft.html).

Mais alguns exemplos em código:

```
import numpy as np
import matplotlib.pyplot as plt

x=np.linspace(-100,100,2**12+1)
dx=x[1]-x[0]
x=x[:-1] #retira o ultimo ponto
y=np.sin(x)+np.cos(x)


k=np.linspace(-np.pi/dx,np.pi/dx,len(x))
yk=np.fft.fftshift(np.fft.fft(y,norm="ortho")) #muda o
                          #intervalo, [-pi,pi]-¿[0,2pi]
yk*=np.exp(1j*x[0]*k)

plt.subplot(221) #mesmo que plt.subplot(2,2,1)
```

```
plt.plot(k,np.real(yk),'b')
plt.legend(["Parte Real"],loc='upper right')
plt.xticks([],[]) #retira eixo x
plt.xlim(-10,10)

plt.subplot(222) #mesmo que plt.subplot(2,2,2)
plt.plot(k,np.imag(yk),'r')
plt.legend(["Imaginária"],loc='upper right')
plt.xticks([],[]) #retira eixo x
plt.xlim(-10,10)

plt.subplot(212) #mesmo que plt.subplot(2,1,2)
plt.plot(k,(np.abs(yk))**2,'m')
plt.legend(["Módulo Quadrado"],loc='upper right')
plt.xlim(-10,10)

plt.tight_layout()
plt.savefig('transformadaplot.png')
```

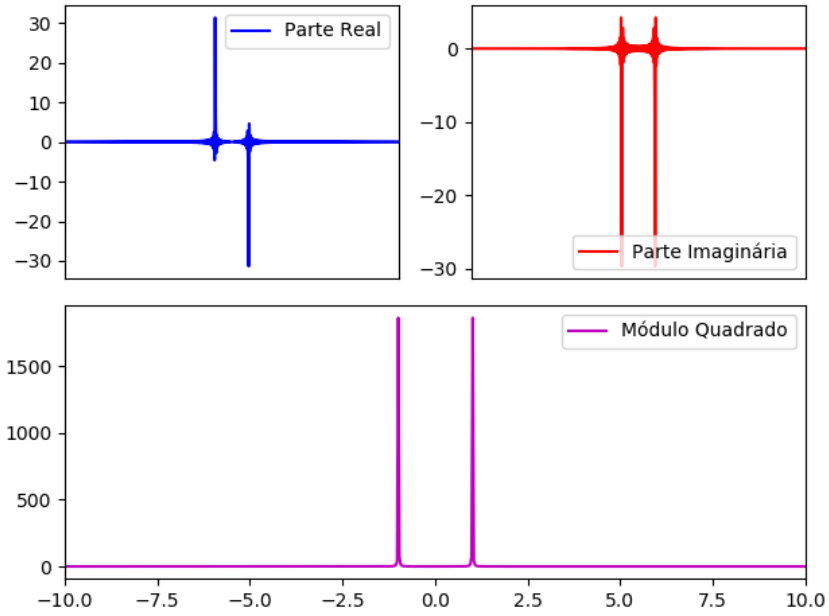


Fig. 16: Primeiro foi plotada a parte real da transformada de fourier, depois a parte imaginária e, por último o módulo ao quadrado.

Agora, peço que você escreva os seguintes códigos:

- Complemente o código acima com a transformada inversa e suas figuras;
- Faça a transformada de Fourier do seno e cosseno (confira com a resposta analítica);
- Resolva uma a EDO "y"+y'+y=0"via transformada de fourier (exemplo em http://www.thefouriertransform.com/applications/differentialequations.php)

14.4 *Álgebra Linear*

Por último veremos o pacote de álgebra linear dentro do NumPy chamado "linalg", muito simples, mas extremamente poderoso! Com esse pacote você será capaz de encontrar determinante de matrizes, inversa de matrizes, produto matricial, matriz adjunta, resolver problemas de auto-valores e auto-vetores, entre outros usos mais específicos.

Assim como todos os pacotes apresentados, recomendo fortemente que você leia a documentação do "linalg"em https://numpy.org/doc/stable/reference/routines.linalg.html.

Vamos aos exemplos:

```
import numpy as np
import numpy.linalg as alg #importanto como alg

M1=np.eye(100) #matriz unitaria de ordem 100
M2=np.eye(100)*5-np.eye(100,k=-2)-np.eye(100,k=2)
#k=n gera uma matriz com elementos na n-esima diagonal

V1=np.linspace(0,10,100) #vetor de cem elementos
V2=np.linspace(-np.pi,np.pi,100) #outro vetor

A=alg.solve(M1,V1) #resolve o sistema linear

B,C=alg.eig(M1) #B recebe os autovalores
                #C recebe os autovetores em linhas

B,C=alg.eigh(M1)#B recebe os autovalores caso a matriz
                #seja hermitiana

                #C recebe os autovetores em colunas
                #caso a matriz seja hermitiana

C=np.dot(M1,M2)#produto matricial entre M1 e M2
               #repare que M1*M2=m1[i,j]*m2[i,j]
               #logo "np.dot" difere de "*"

D=alg.int(M1)#retorna a inversa de M1

A=alg.norm(M1)#normaliza a matriz/vetor

B=alg.det(M1)#retorna o determinante de M1

C=np.inner(V1,V2)#retorna o produto
                 #interno entre os vetores

D=np.outer(V1,V2)#retorna o produto externo entre V1 e V2
```

O pacote é curto, grande parte das funções disponíveis são mostradas acima, mas para ter certeza de que as ideias ficaram claras, resolveremos alguns problemas:

- Gere uma matriz do tipo:

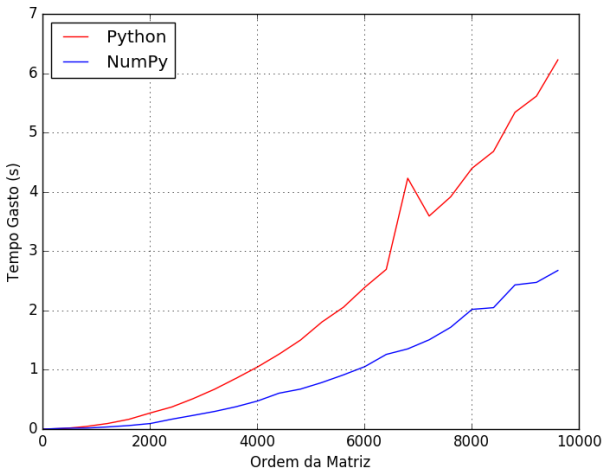
$$M = \begin{pmatrix} 2 & -1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 2 & -1 & 0 & 0 \\ 0 & 0 & 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & 2 \end{pmatrix}$$

- Encontre os auto-valores e auto-vetores da matriz acima (utilize n=100);
- Plote os três primeiros auto-vetores da matriz acima (utilize n=100);
- Encontra sua inversa e seu determinante.

Ok, vamos resolver os problemas juntos dessa vez, o primeiro problema:

```
import numpy as np
import numpy.linalg as alg

#PRIMEIRO

n=10**3 #ordem da matriz
#Do jeito mais tradicional
M=np.zeros((n,n))
for i in range(n):
    M[i,i]=2
    if i¿0:
        M[i-1,i]=-1
    if i!=(n-1):
        M[i,i+1]=-1

#Vetorialmente
M=2*np.eye(n)-np.eye(n,k=-1)-np.eye(n,k=+1)
```

O código em Numpy é em média sete vezes mais rápido:



O segundo problema:

```
import numpy as np
import numpy.linalg as alg

#SEGUNDO

n=10**2 #ordem da matriz
M=2*np.eye(n)-np.eye(n,k=-1)-np.eye(n,k=+1)

val,vec=alg.eigh(M) #trata-se de uma matriz hermitiana
```

O terceiro problema:

```
import numpy as np
import numpy.linalg as alg
import matplotlib.pyplot

#TERCEIRO

n=10**2 #ordem da matriz
M=2*np.eye(n)-np.eye(n,k=-1)-np.eye(n,k=+1)

val,vec=alg.eigh(M) #trata-se de uma matriz hermitiana

plt.plot(np.abs(vec[:,0:3])**2) #modulo quadrado dos tres
                                #primeiros auto-vetores

plt.savefig("autovetores.png")
plt.show() #mostrar em tela
```

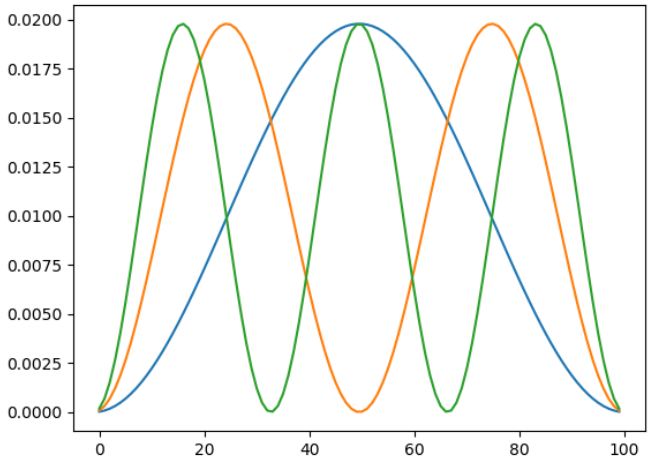


Fig. 17: Caption

O quarto problema:

```
import numpy as np
import numpy.linalg as alg

#QUARTO

n=10**2 #ordem da matriz
M=2*np.eye(n)-np.eye(n,k=-1)-np.eye(n,k=+1)

INV=alg.inv(M)
DET=alg.det(M)
```

Agora, você está mais que apto a resolver problemas de álgebra linear com tranquilidade, alguns links e páginas interessantes são (http://hplgit.github.io/pyhpc/doc/pub/._project001.html sobre métodos numérico com NumP e https://www.geeksforgeeks.org/numpy-linear-algebra/ sobre métodos algébricos).

SciPy

15 SCIPY

SciPy ou "Scientific Python"trata-se do pacote científico de Python escrito quase inteiramente sobre o NumPy, no SciPy vamos ter ferramentas mais voltadas ao meio científico, como resolução de EDO's, resolução de integrais, funções especiais(Bessel, Legendre, Hermite,...), interpolação, transformada de fourier, álgebra linear, estatística e etc.

Trata-se sem dúvidas de um pacote extenso e muito bem otimizado, recomendo que você leia sobre a documentação do pacote para que fique claro o quão importante ele é https://docs.scipy.org/doc/scipy/reference/.

Nesse capítulo focaremos em resolver diversas EDO's e integrais para que o uso do pacote (para esse propósito) fique bem claro.

15.1 *Equações Diferenciais Ordinárias*

Para solução de EDO's usaremos o comando "scipy.integrate.solve_ivp()"que nos permite resolver EDO's diversas com muita facilidade, mas antes de escrever o código vamos entender como chamar o algorítimo, o primeiro passo é entender a sintaxe:

OBS: devido a uma limitação no editor de texto que utilizo, sempre que você ver "solve-ivp"escrito substitua no seu código o hífen pelo "underline"(solve-ivp ⟶ solve_ivp)

```
import scipy
scipy.integrate.solve-ivp(fun, t-span, y0, method='RK45',
t-eval=None)

#fun -- sistema da EDO (explicarei abaixo do código)
##t-span -- intervalo da EDO (t0, tf)

##method -- metodo de sua escolha como:
##    -RK45:Runge Kutta de quarta ordem com correcao de quinta
##    -RK23:RK de segunda ordem com correcao de terceira
##    -DOP854:Runge Kutta explicito de oitava ordem
##    -...

##t-eval=pontos de tempo desejado
##caso nao escreva nada, o proprio solver decidira
```

Beleza, mas o que quer dizer "sistema de EDO"? Pense na seguinte EDO:

$$y'' + y' + y = 0 \tag{27}$$

Podemos diminuir sua ordem:

$$\begin{aligned} y' &= z \\ y'' &= -(z + y) \end{aligned} \tag{28}$$

Agora, escrevamos cada item como elemento de uma lista: l[0]=y, l[1]=y' e l[2]=y". Nosso sistema acima fica:

$$\begin{aligned} l[1] &= z \\ l[2] &= -(l[1] + l[0]) \end{aligned} \tag{29}$$

Com isso em mente, vamos escrever um código em python que resolva e EDO acima:

```
from scipy.integrate import solve-ivp as ivp


ci=[1,0] #condicoes iniciais
t0=0 #tempo inicial
tf=10 #tempo final

def edo(t,y): #primeiro parametro deve ser o tempo
    return(y[1],-(y[0]+y[1]))

resposta=ivp.(edo,(t0,tf),ci)
t=resposta.t #retorna os pontos no tempo
y=resposta.y[0] #como e uma edo de segunda ordem
                #ele retornara uma matrix de len(t)
                # e n colunas (n e a ordem da EDO)
ydot=resposta.y[1]
```

Simples, não? Resolva e plote os exercícios abaixo para que as ideias fiquem claras(tente resolver e plotar os problemas acima sozinho pois só assim você criará intimidade com o pacote e terá certeza, mas não a ilusão de que aprendeu):

- $y'' + w^2 sen(y) = 0$
- $y' = cos(t)y$
- $y'' + Ay' + By = f(t)$
- $x' = \sigma(y - x);\ y' = -y - xz;\ z' = -\beta z + xy$

Vamos às respostas:

Primeiro problema:

```
from scipy.integrate import solve-ivp as ivp
import matplotlib.pyplot as plt
import numpy as np

#constantes
t0=0
tf=10
y0=5 #ci para funcao
y0dot=0.1 #condicao inicial para a derivada
w=2

#definir o sistema
def edo(t,y):
```

```
    return(y[1],-y[1]-(w**2)*np.sin(t))

solucao=ivp(edo,(t0,tf),[y0,y0dot])
#deixamos o solver escolher os valores de t entre t0 e tf

solucao2=ivp(edo,(t0,tf),[y0,y0dot],t_eval=np.linspace(t0,tf,2**10))
#acima, especificamos os valores de t entre t0 e tf

plt.plot(solucao.t,solucao.y[0],'r') #y1
plt.plot(solucao.t,solucao.y[1],'b') #y1dot

plt.plot(solucao2.t,10+solucao2.y[0],'y') #y1
plt.plot(solucao2.t,10+solucao2.y[1],'g') #y1dot




plt.legend(["y1(x)","y'1(x)","y2(x)","y'2(x)"],loc='upper right')

plt.savefig("plotsemdense.png")
plt.show()
```

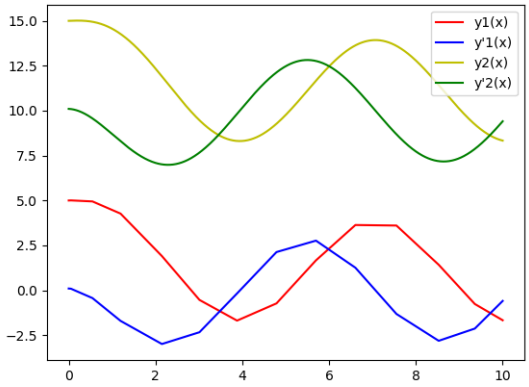


Fig. 18: Comparação entre pontos escolhidos pelo programa e pontos escolhidos pelo usuário.

Segundo problema:

```
from scipy.integrate import solve-ivp as ivp
import matplotlib.pyplot as plt
import numpy as np

#constante
y0=10 #ci
t0=0
tf=10

def edo(t,y):
```

```
    return(np.cos(t)*y)

solucao=ivp(edo,(t0,tf),[y0],t_eval=np.arange(t0,tf,10**-3))

plt.plot(solucao.t,solucao.y[0],"r")
plt.grid(True)
plt.legend(["y(x)"],loc="upper right")
plt.xlim(t0,tf)
plt.title("Solução")

plt.ylabel("y(t) (m)")
plt.xlabel("t (s)")
plt.savefig("segundoivp.png")
plt.show()
```

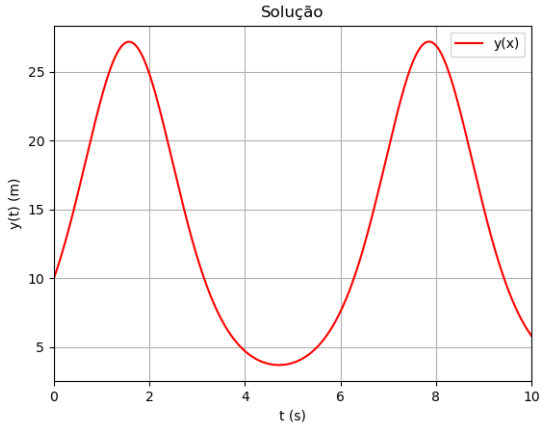


Fig. 19: Sulução da EDO $y' = ycos(x)$

Terceiro problema:

```
from scipy.integrate import solve-ivp as ivp
import matplotlib.pyplot as plt
import numpy as np

#constante
y0=10 #ci
y0dot=0
t0=0
tf=50

A=7
B=9
amp=15 #vamos usar f(t) = amp*cos(t)

def edo(t,y):
    return(y[1],-A*y[1]-B*y[0]+amp*np.cos(t))
```

```
solucao=ivp(edo,(t0,tf),[y0,y0dot],t_eval=np.arange(t0,tf,10**-4))

plt.plot(solucao.t,solucao.y[0],"r")
plt.plot(solucao.t,solucao.y[1],"b--")

plt.grid(True)
plt.legend(["y(x)","y'(x)"],loc="upper right")
plt.xlabel("t (s)")
plt.savefig("terceiroivp.png")
plt.show()
```

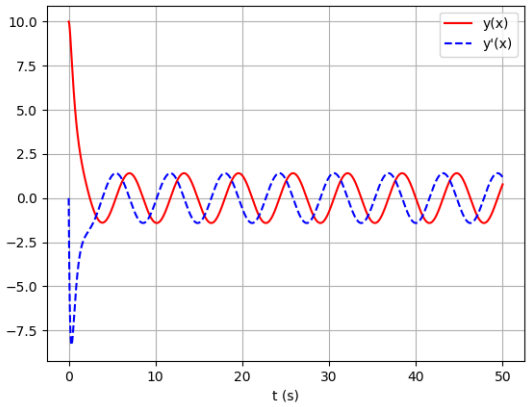


Fig. 20: Gráfico da solução da EDO acima, tanto a função quanto sua derivada no tempo delimitado.

Quarto e mais elegantes problema, um conjunto de três edo's acopladas em primeira ordem:

```
from scipy.integrate import solve-ivp as ivp
import matplotlib.pyplot as plt
import numpy as np

sigma=10 #constante
beta=8/3 #constante
ro=28 #constante

ti=0 #tempo inicial
tf=50 #tempo final

tempo=[ti,tf] #vetor de tempo ro ivp
intervalo=np.arange(0,50,0.05) #tempo desejado
cis=[0.1,0.1,0.1] #vetor de condicoes iniciais


def edo(t, v):
    return [sigma*(v[1]-v[0]), ro*v[0]-v[1]-v[0]*v[2],
            -beta*v[2]+v[1]*v[0]]
```

```
sol=ivp(edo,tempo,cis,method="Radau")#,method="RK45",interval)

#sol.y[0] -- x
#sol.y[1] -- y
#sol.y[2] -- z

x=sol.y[0]
y=sol.y[1]
z=sol.y[2]

plt.plot(sol.t,x,"r")
plt.plot(sol.t,y-50,"m") #repare que essa soma translada o grafico
plt.plot(sol.t,z+10,"y") #faço isso para melhor visualização

plt.grid(True)
plt.xlim(ti,tf)
plt.legend(["x(t)","y(t)","z(t)"],loc="upper left")
plt.xlabel("Tempo (s)")

plt.savefig("quartoivp.png")
plt.show()
```

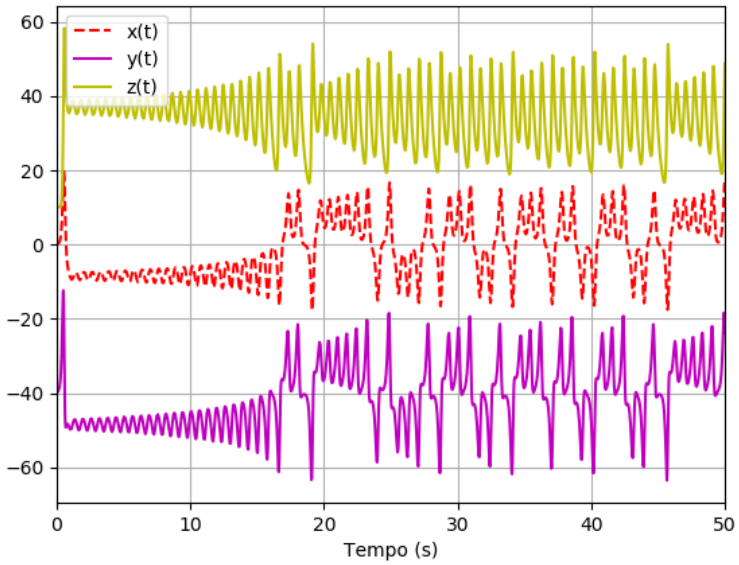


Fig. 21: Gráfico da solução obtida via "ivp"para as edo's acopladas.

Recomendo que você resolva as EDO's dos capítulos anteriores e compare os resultados (tempo gasto e precisão obtida) já obtidos via implementação própria(Verlet, Runge Kutta e Euler).

15.2 *Integração*

Como último tema da apostila, falarei sobre funções do SciPy para realizar integração de funções com um estrutura otimizada e completas, como sempre, peço que você leia a documentação e analise os exemplos do pacote em https://docs.scipy.org/doc/scipy/reference/tutorial/integrate.html.

As funções de integração podem ser divididas em dois tipos, um grupo delas recebe uma array e integra a partir de valores numéricos já estabelecidos (trapz, simps) já o segundo grupo recebe um uma função previamente definida e integra a partir dela (quad, dblquad e tplquad).

Primeiro façamos um código para o primeiro grupo:

```
import numpy as np
import scipy.integrate as inte

n=10**3 #numero de pontos
M=np.zeros((n,2)) #uma coluna para x e outra para f(x)

x=np.linspace(0,2*np.pi,n)
y=np.sin(x)**2

A=inte.trapz(y,x)
B=inte.simps(y,x)

#resutado de A: 3.141592653589793
#resutado de B: 3.1415926950552224
#resultado analitico: 3.141592653589793
```

Para o segundo grupo de métodos:

```
import numpy as np
import scipy.integrate as inte

def func(x): #funcao a ser integrada
    return(np.sin(x)**2)

def func2(x,t):
    return(np.sin(x)**2+np.sin(t)**2)

def func3(x,t,z):
    return(np.exp(np.cos(x*t*z)**2))

A=inte.quad(func,0,5) #integra de 0 a 5
B=inte.dblquad(func2,0,5,0,1) #integra x de 0 a 5
                              #integra t de 0 a 1
C=inte.tplquad(func3,0,1,0,np.pi,0,10) #x e [0,1]
                                       #t e [0,np.pi]
                                       #z e [0,10]

#Os resultados saem como Integra, Erro Absoluto
#resutado de A: 2.6360052777223424, 3.746763545689801e-13
```

```
#resutado de B: 3.9993834941902406, 3.0904877090678896e-13
#resultado analitico: 58.472731836273354, 8.535902199307306e-07
```

O código acima expõe quase tudo que você precisará, passarei alguns exercícios para fixação das ideias apresentadas (quaisquer dúvidas podem ser solucionadas em https://docs.scipy.org/doc/scipy/reference/tutorial/integrate.html):

- $\int_0^\infty e^{-x^2} dx$;
- $\int_{y=0}^{\frac{1}{2}} \int_{x=0}^{1-2y} sen(xy) dxdy$;
- $\int_0^1 \int_0^1 \int_0^1 xyzdxdydz$ (compare com Monte Carlo);

Aqui acaba nossa trajetória, espero que agora você seja capaz de resolver quaisquer problemas envolvendo equações diferenciais, integrais diversas.

Parte IV. Apêndice

Instalação do Spyder

Spyder é um ambiente de desenvolvimento que possui todos os pacotes utilizados ao longo da apostila. Trata-se de um programa leve (usável caso seu PC tenha mais de 2gb de RAM), extremamente intuitivo e gratuito.

As principais IDE's (ambientes de desenvolvimento) em Python são apresentadas no link https://hackr.io/blog/best-python-ide, embora existam várias opções, utilizaremos o Spyder (simples, direto e leve).

A COMO INSTALAR NO UBUNTU/DEBIAN

https://www.youtube.com/watch?v=3olPAMnh3Kc

B COMO INSTALAR NO WINDOWS

https://www.youtube.com/watch?v=DiuUHoVyxsY

C PRIMEIROS PASSOS COM O SPYDER (O MESMO PARA TODOS OS SO)

https://www.youtube.com/watch?v=a1P_9fGrfnU