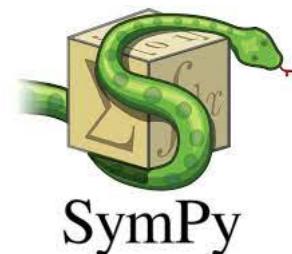




Universidade Federal do Pará - UFPA  
Instituto de Geociências - IG  
Faculdade de Geofísica - FAGEOF

## Introdução ao Pacote Sympy



Autor : *Marcelo Lucas Almeida*

[marcelolucasif@gmail.com](mailto:marcelolucasif@gmail.com)

# Sumário

<b>1</b>	<b>Introdução</b>	<b>2</b>
1.1	Objetivos da Apostila . . . . .	2
1.2	Instalação do SymPy e Jupyter Notebook . . . . .	3
1.2.1	Instalação no Windows . . . . .	3
1.2.2	Instalação no Linux . . . . .	4
1.2.3	Verificando a versão do Sympy . . . . .	4
<b>2</b>	<b>Começando com SymPy</b>	<b>6</b>
2.1	Criando Símbolos em SymPy . . . . .	6
2.2	Outras Funções e Métodos Úteis . . . . .	12
2.2.1	O Método evalf() . . . . .	12
2.2.2	A Função lambdify() . . . . .	13
2.2.3	Criando Igualdades em SymPy . . . . .	14
<b>3</b>	<b>Plotagem com SymPy</b>	<b>16</b>
3.1	Plote de uma função de uma variável . . . . .	16
3.2	Personalizando os Gráficos em Sympy . . . . .	18
3.3	Plote de Funções Paramétricas . . . . .	22
3.4	Plote de Funções Implícitas 2D . . . . .	24
3.5	Plote de Funções de Duas Variáveis . . . . .	27
3.5.1	Plote de Funções Paramétricas no Espaço . . . . .	29
3.5.2	Superfícies Paramétricas . . . . .	30
<b>4</b>	<b>Cálculo</b>	<b>32</b>
4.1	Limites . . . . .	33
4.2	Somatório e Produtório . . . . .	35
4.3	Derivadas . . . . .	39
4.3.1	Derivadas de Funções de uma variável . . . . .	39
4.3.2	Série de Taylor . . . . .	41
4.3.3	Série de Fourier . . . . .	42
4.3.4	Derivadas de Funções de várias variáveis . . . . .	46
4.4	Integrais . . . . .	47
4.4.1	Integrais de Funções de uma variável . . . . .	47
4.4.2	Integrais Impróprias . . . . .	48
4.4.3	Funções Especiais . . . . .	50
4.4.4	Integrais Duplas e Triplas . . . . .	54
<b>5</b>	<b>Resolvendo Equações em SymPy</b>	<b>55</b>
5.1	Equação Linear . . . . .	55
5.2	Sistema de Equações Lineares . . . . .	58
5.3	Resolvendo Equações Diferenciais . . . . .	59

# 1 Introdução

O *Sympy* é um pacote do *Python* voltado para **Matemática Simbólica**, que diz respeito ao uso de computadores para manipular equações matemática de forma 'exata', no sentido de resolver ou simplificar a equação de forma similar ao que faríamos se tivéssemos resolvendo à mão, em oposição à mera aproximações de quantidades numéricas específicas representadas por aqueles símbolos.

Um pacote que manipula expressões de forma simbólica é geralmente chamado de *CAS*(*computer algebra system*), no qual é capaz de resolver *equações lineares* e *não-lineares*, *cálculo de derivadas* e *integrais* e muito mais. O Sympy é um pacote extremamente rico para manipulação simbólica, que vai desde uma simples operação algébrica até expressões mais complicadas como cálculos de campos vetoriais.

## 1.1 Objetivos da Apostila

O foco principal dessa apostila é introduzir o leitor aos conceitos iniciais de computação simbólica, resolvendo problemas matemáticos através de exemplos usando o pacote SymPy. Os Pré-Requisitos que são necessários para entender essa apostila, são os conceitos de programação com *Python*<sup>1</sup>, como : *tipos de variáveis* (*int*, *float*, *str* ...), *listas*, *tuplas*, *dicionários*, *funções*, *blocos de decisões*, *blocos de repetições* e *alguns conceitos de tratamentos de exceções*.

Caso o leitor não tenha nenhum conhecimento prévio sobre a linguagem Python, recomendo fortemente que estude um pouco sobre, antes de iniciar essa apostila. Uma boa fonte de conhecimento é a sua documentação oficial, que pode ser encontrada nesse link (<https://docs.python.org/pt-br/3/tutorial/index.html>). Apesar da documentação do Python ser rica em detalhes, para alguns tópicos, a documentação não é tão didática para o iniciante. Para isso, sugiro alguns livros introdutórios como : [2], [1]. Para livros no caráter mais intermediário, os seguintes livros são interessantes : [3], [4].

Juntamente com apresentação dos comandos do pacote SymPy, iremos aprender sobre esse pacote através da resolução de problemas, com objetivo de fixar e aprender a programar usando a computação simbólica. O exemplos e toda a programação desse módulo do Python será feita no *Jupyter Notebook*<sup>2</sup>, os comandos são inseridos no que chamamos *células*, que são campos para digitação do código e aparecem com numerações que indicam a que célula os códigos pertencem. Por exemplo, *In/1* significa que estamos na célula de numero 1. Para executar o código de uma célula no Jupyter, apertamos SHIFT + ENTER na própria célula de comando.

---

<sup>1</sup>De preferencia o Python 3

<sup>2</sup>O Projeto Jupyter é uma organização sem fins lucrativos criada para "desenvolver software de código aberto, padrões abertos e serviços para computação interativa em dezenas de linguagens de programação". Originado do IPython em 2014, o Projeto Jupyter suporta ambientes de execução em dezenas de linguagens de programação.

## 1.2 Instalação do SymPy e Jupyter Notebook

Para instalar o pacote Sympy você deve ter em seu computador o *Python* instalado, assim como o *PIP*<sup>3</sup> ou o *Conda*<sup>4</sup>. Mostrarei como instalar esse pacote em duas distribuições, a saber, no *Windows* e no *Linux*.

### 1.2.1 Instalação no Windows

Para instalação via *Conda*, você deverá usar o seguinte comando no seu PowerShell :

- `conda install sympy`

Insira *y* quando for solicitado. Você receberá uma mensagem semelhante assim que a instalação for concluída

```
(base) PS C:\Users\Geeks> conda install sympy
Collecting package metadata (current_repodata.json): done
Solving environment: done

## Package Plan ##

environment location: C:\Users\Geeks\anaconda3

added / updated specs:
- sympy

The following NEW packages will be INSTALLED:

mpmath      pkgs/main/win-64::mpmath-1.2.1-py38haa95532_0
sympy       pkgs/main/win-64::sympy-1.8-py38haa95532_0

Proceed ([y]/n)? y
Preparing transaction: done
Verifying transaction: done
Executing transaction: done
```

Figura 1: imagem da instalação do sympy no windows via Conda

Certifique-se de seguir as práticas recomendadas para instalação usando conda como:

- Use um ambiente para instalação em vez do ambiente de base usando o comando abaixo:

- `conda create -n my-env`
- `conda activate my-env`

Para instalação via *PIP*, você pode estar usando o seguinte comando :

- `pip install sympy`

Você receberá uma mensagem semelhante assim que a instalação for concluída:

```
C:\Users\Geeks>pip install sympy
Collecting sympy
  Downloading sympy-1.8-py3-none-any.whl (6.1 MB)
    |████████| 6.1 MB 6.4 MB/s
Requirement already satisfied: mpmath>=0.19 in c:\users\geeks\anaconda3\lib\site-packages (from sympy) (1.2.1)
Installing collected packages: sympy
Successfully installed sympy-1.8
```

Figura 2: imagem da instalação do sympy no windows via PIP

### 1.2.2 Instalação no Linux

Para usuários de linux, o seguinte comando pode ser usado para instalação do Sympy em sua máquina :

- `sudo apt-get update`
- `sudo apt-get install python-sympy`

### 1.2.3 Verificando a versão do Sympy

Se o Sympy foi instalado corretamente em seu computador, você poderá verificar a sua versão usando os seguintes comandos no seu shell :

- `import sympy`
- `sympy.__version__`

---

<sup>3</sup>pacotes de instalação do Python, para instalar o pip no linux vá até <http://devfuria.com.br/linux/installando-pip/>. Para instalar no windows vá para <https://acervolima.com/como-instalar-o-pip-no-windows/>.

<sup>4</sup>Pacote de instalação que vem com a distribuição *Anaconda*, para baixar o Anaconda em seu computador entre no site <https://www.anaconda.com/>.

Uma vez que você tem pacote *Anaconda* instalado em seu computador, o editor Jupyter já vêm como instalado como padrão. Para executar o Jupyter, abra o terminal<sup>5</sup> e digite o seguinte comando :

A screenshot of a terminal window on a dark background. The text in the window is white. At the top left, it shows the user's name and host: 'marcelo\_lucas@mlucas:~\$'. Below that, the command 'jupyter notebook' is typed and followed by a new line character.

Figura 3: Comando para inicializar o Jupyter no terminal.

Assim, o jupyter abrirá no seu navegador padrão. E terá a seguinte cara :

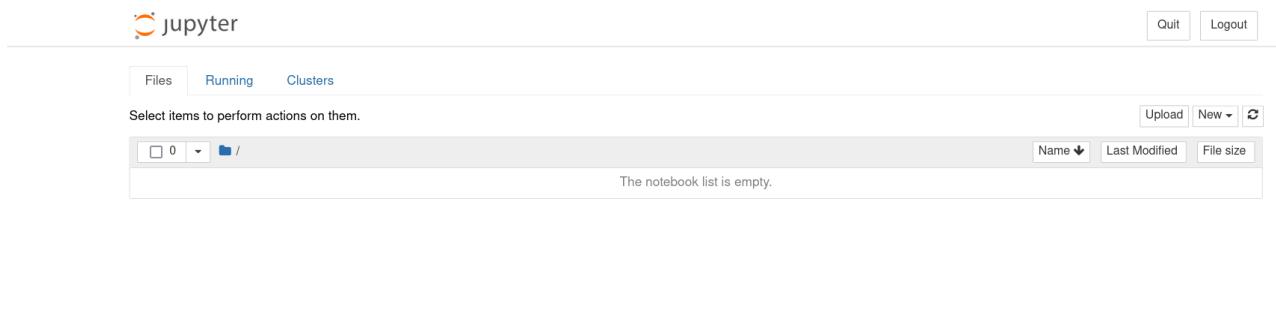


Figura 4: Tela inicial do Jupyter.

Para criar um script em Python, vá no superior direito e aperte em **New** e selecione a opção **Python 3**.

---

<sup>5</sup>O jupyter irá abrir no diretório atual que você se encontra, no Linux por default, o diretório terá o seguinte caminho : */home/pasta\_usuário*.

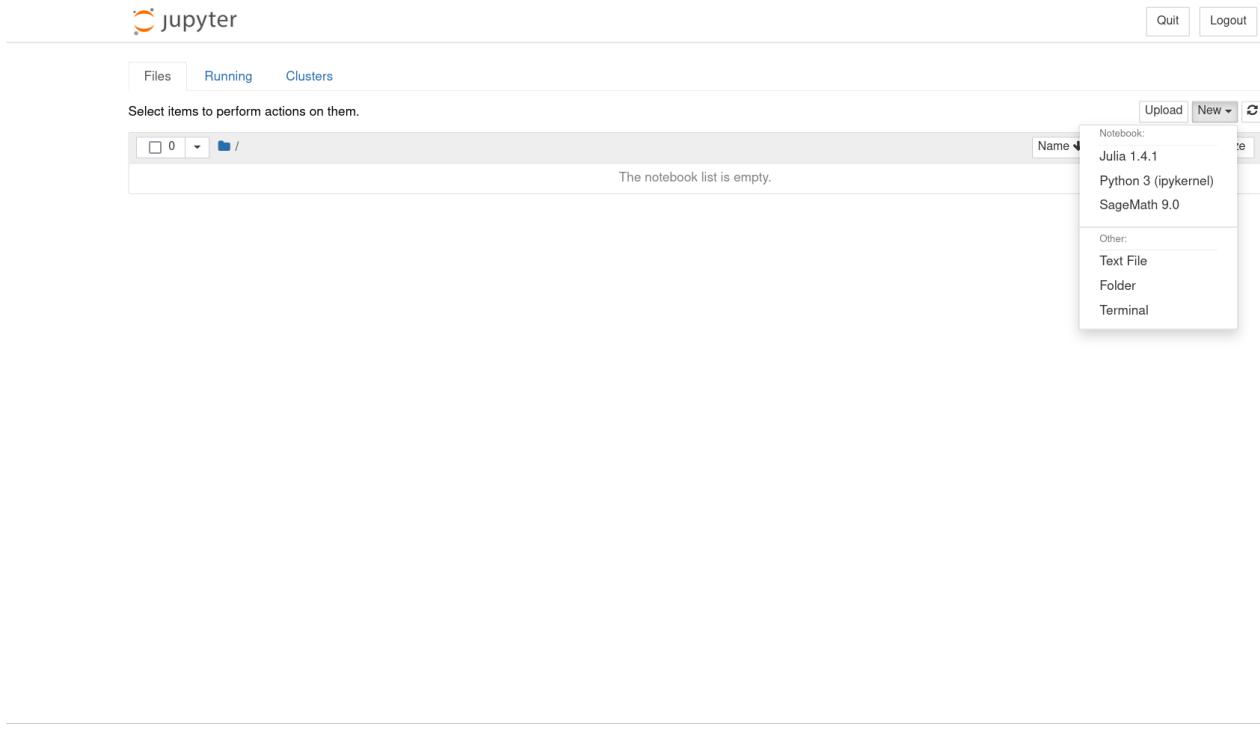


Figura 5: Selecionando a linguagem Python no Jupyter.

Pronto!, agora você poderá criar programas usando a linguagem Python.

## 2 Começando com Sympy

O *SymPy* é uma biblioteca de Python voltada para fazer cálculos simbólicos. O cálculo simbólico é diferente do cálculo numérico, pois o mesmo retorna um valor exato e não aproximado como no último caso, isto é, uma resposta analítica para um dado problema que se queira resolver(se houver analítica<sup>6</sup>). A vantagem de se usar as operações simbólicas ao invés da numéricas, encontra-se no fato de ajudar na elaboração e resolução de determinados problemas matemáticos, por exemplo, o cálculo simbólico pode ajudar a verificar se um dado processo de integração ou diferenciação estão corretos, isso ajuda o aluno a confirmar o resultado de um dado exercício. Sem mais delongas, vejamos como trabalhar com essa biblioteca poderosa para computação simbólica.

### 2.1 Criando Símbolos em SymPy

Uma vez que a biblioteca de SymPy encontra-se instalada em sua máquina, para usar todas as suas funcionalidades, devemos fazer a importação dessa biblioteca, em Python fazemos da seguinte forma :

- `import sympy`

<sup>6</sup>Muitos problemas de modelagem matemática, como por exemplo **equações diferenciais**, não possuem uma solução exata ou fechada, necessitando de uma solução aproximada(*numérica*).

Depois disso, todas as funcionalidades<sup>7</sup> Dessa biblioteca estarão disponíveis para ser usadas. Uma forma mais fácil de usar os comandos desse módulo é dando um 'apelido' ao nome *Sympy*, isso pode ser feito em Python usando a palavra *as* e em seguida atribuindo um nome que vc queira colocar. Por convenção, usamos o seguinte comando :

- `import sympy as sp`

Assim toda e qualquer função associada a biblioteca será chamada da seguinte forma `sp.function()`.

O objeto<sup>8</sup> principal do SymPy é o que chamamos de **variável simbólica**(ou valor simbólico), que representa um conjunto de símbolos. Através dessas variáveis que podemos realizar várias operações relacionadas à computação simbólica em Python. Para definir um valor simbólico usamos o seguinte comando `sp.symbols()`<sup>9</sup>, vejamos isso em um exemplo :

### Exemplo 2.1 : Definindo um valor simbólico

```
In [4]: import sympy as sp # importando a biblioteca do SymPy
x = sp.symbols('x') # criando a variavel simbolica 'x'
x
Out[4]: x
```

Figura 6: Criação de uma variável simbólica no SymPy.

No momento é importante falar de editores para programar, o exemplo acima, o código está escrito no editor chamado *Jupyter Notebook*, onde In[1] : indica a primeira *célula*(linha) desse editor. Para executar um comando nesse editor, pasta pressionar as teclas SHIFT + ENTER que a célula será executada. Se vc estiver em um console do IPYTHON, basta apertar ENTER em para cada comando, que será executado de imediato.

Voltando ao código do exemplo acima. Na primeira linha de comando `import sympy as sp` , importamos a biblioteca do SymPy e a referenciamos com o nome de `sp`. Na segunda linha, usamos a função `symbols('')` para definir uma variável simbólica "x". Perceba que entre parenteses passamos o símbolo entre aspas para a nossa variável simbólica que também é "x". Por quê passamos(ou tentamos) o mesmo símbolo para a variável simbólica?, fazemos isso, porque é perfeitamente válido passar um símbolo diferente para a variável que receberá esse valor, por exemplo, se fizéssemos `x = sp.symbols('a')`,

<sup>7</sup>Não todas na verdade, só as que se encontram no *namespace* da biblioteca do SymPy. Alguns comandos não poderão ser acessado devido a não pertencer à esse *namespace*, de tal forma, que só poderão ser acessados se importarmos um *submódulo* dessa biblioteca. Veremos isso no decorrer dessa apostila.

<sup>8</sup>Lembre que tudo em Python é um objeto, isso é válido também para bibliotecas externas da linguagem.

<sup>9</sup>Existe também o comando `sp.Symbol()`, no singular e começando com a letra maiúscula. A diferença entre esses dois comandos é que `symbols()` permite definir vários símbolos ao mesmo tempo.

nesse caso, a variável simbólica ” $x$ ” iria referenciar o valor simbólico ” $a$ ”, ou seja ,  $x \rightarrow a$ . Geralmente criamos um símbolo igual para as nossas variáveis simbólicas. Executando o exemplo acima, temos

A função `symbols()` é versátil pois ela pode definir vários símbolos ao mesmo tempo, isso pode ser feito da seguinte forma :

**Exemplo 2.2 : Definindo vários valores simbólicos**

```
In [2]: import sympy as sp
x , y = sp.symbols('x,y')
print(x)
print(y)
x
y
```

Figura 7: Criação de vários símbolos com SymPy.

Podemos passar como um segundo argumento para a função `symbols()`, uma característica sobre o símbolo em questão, como por exemplo, se o dado símbolo é um *Real*, *Complexo*, *Inteiro* e etc. Essas informações adicionais que podemos passar para os nossos símbolos criados é útil para o SymPy, assim em certas simplificações , o SymPy fará de forma mais correta possível, apesar de não ser algo obrigatório. Essas características são chamadas de *atributos* do valor simbólico. Vejamos um exemplo de como definir vários símbolos com características diferentes.

**Exemplo 2.3 : Definindo vários valores simbólicos com características diferentes**

```
In [3]: import sympy as sp
x = sp.symbols('x', real = True) # define 'x' como um valor real
i = sp.symbols('i', integer = True) # define 'i' como um valor inteiro
y = sp.symbols('y', complex = True )# define 'y' como um valor complexo
```

Figura 8: especificando atributos aos símbolos.

Especificar atributos à um dado valor simbólico, pode ajudar o SymPy a calcular e simplificar expressões algébricas de forma mais rápida e precisa. Abaixo se encontra uma tabela de atributos que podemos ser passado para a função `symbols()`.

Atributo	Descrição
<code>real</code>	<i>define um símbolo como sendo real</i>
<code>integer</code>	<i>define um símbolo como sendo inteiro</i>
<code>complex</code>	<i>define um símbolo como sendo complexo</i>
<code>is_positive</code>	<i>define um símbolo como sendo um valor positivo</i>
<code>is_negative</code>	<i>define um símbolo como sendo um valor negativo</i>

Uma vez que já sabemos definir variáveis simbólicas e acrescentar certas características à esses símbolos, vejamos como podemos usar esse símbolos em expressões algébricas com o SymPy.

#### Exemplo 2.4 : Expressões algébricas com o SymPy.

```
In [4]: import sympy as sp
sp.init_printing(use_latex= True)

x = sp.symbols('x', real = True)
a , b ,c = sp.symbols('a,b,c', integer = True )

expr1 = a*x**2 + b*x + c    # expressão do 2º grau
expr2 = -2*a*x**2 + 4*b*x   # segunda expressão

soma = expr1 + expr2        # soma as duas expressões
sub = expr1 - expr2        # subtração das duas expressões

print('A expressão 1 é : ', expr1)
print()
print('a expressão 2 é : ', expr2)
print()
print('A soma é : ', soma)
print()
print('A subtração é : ', sub)
```

A expressão 1 é :  $a*x^{**2} + b*x + c$   
a expressão 2 é :  $-2*a*x^{**2} + 4*b*x$   
A soma é :  $-a*x^{**2} + 5*b*x + c$   
A subtração é :  $3*a*x^{**2} - 3*b*x + c$

Figura 9: Cálculos com Expressões simbólicas.

Observe que no exemplo acima criamos os símbolos `x,a,b` e `c` como sendo valores reais, e através desses valores criamos expressões simbólicas que representam equações do segundo grau e atribuímos a certas variáveis. Depois realizamos a soma e subtração dessas expressões, de forma que o SymPy calcula de forma exata essas expressões algébricas.

É importante mencionar nesse momento que o SymPy tem uma vasta biblioteca para todas as funções matemáticas, como trigonométricas, exponenciais, logarítmicas, hiperbólicas e muitos mais. Abaixo se encontra uma tabela onde mostrar algumas funções mais importante que se encontra no SymPy. Para uma lista extensiva sobre todas as funções desse pacote, você pode consultar a documentação oficial do SymPy que se encontra em (<https://docs.sympy.org/latest/modules/functions/index.html>).

Função	Comando em SymPy
$\sin(x)$	<code>sympy.sin(x)</code>
$\cos(x)$	<code>sympy.cos(x)</code>
$\tan(x)$	<code>sympy.tan(x)</code>
$\sqrt{x}$	<code>sympy.sqrt(x)</code>
$e^x$	<code>sympy.exp(x)</code>
$\ln(x)$	<code>sympy.log(x)</code>
$\log_b(x)$	<code>sympy.log(x,b)</code>
$\cosh(x)$	<code>sympy.cosh(x)</code>
$\sinh(x)$	<code>sympy.sinh(x)</code>
$\tanh(x)$	<code>sympy.tanh(x)</code>
$\sin^{-1}(x)$	<code>sympy.asin(x)</code>
$\cos^{-1}(x)$	<code>sympy.acos(x)</code>
$\tan^{-1}(x)$	<code>sympy.atan(x)</code>

Agora que sabemos como criar símbolos em SymPy e como usar funções matemáticas, o próximo passo é aprender a manipular essas expressões simbólicas com o SymPy. Uma atividade muito corriqueira que você vai se deparar quando estiver trabalhando com expressões é a *substituição* de uma expressão simbólica por um valor ou outra expressão simbólica e a *simplificação* de expressões simbólicas. Vejamos como fazer ambas as coisas com o SymPy.

Para substituir para de uma expressão por outra no SymPy, usa-se o método<sup>10</sup> `subs()`, onde passamos um *dicionário* para esse método. No dicionário passamos um par *chave*-*valor*, onde nesse caso a *chave* será o o valor da expressão a ser substituído e o *valor* será a nova expressão a ser colocada. Para isso ficar mais claro, vejamos um exemplo de substituição em SymPy.

**Exemplo 2.5 :** Suponha que temos a seguinte função  $f(x) = (x^2 + 2)^3 + 2x$ , e queremos substituir a expressão  $x^2 + 2$  por  $e^{2x} + \cos(2x)$ , isso pode ser feito da seguinte forma :

```
In [5]: import sympy as sp
x = sp.symbols('x' , real = True)
fun = (x**2 + 2)**3 + 2*x      # função f(x)
fun

Out[5]: 2x + (x2 + 2)3

In [6]: fun = fun.subs({x**2 + 2 : sp.exp(2*x) + sp.cos(2*x)}) # nova função
fun

Out[6]: 2x + (e2x + cos(2x))3
```

Figura 10: Usando o método `subs()` para substituir uma expressão.

<sup>10</sup>Um método é uma função aplicada à um determinado tipo de objeto, assim, diferentes objetos terão diferente métodos. Um método difere das funções normais criada pelo usuário pois, a chamamos sobre o dado objeto, usando a seguinte sintaxe de ponto : *objeto.nome\_do\_método()*.

Só uma pequena observação sobre o exemplo acima : dividimos o código em duas células separadas no Jupyter, isso foi feito para que a saída de cada código possa ser renderizada no estilo L<sup>A</sup>T<sub>E</sub>X, isso é feito no Jupyter através de uma biblioteca do JavaScript que já vêm implementada por default no Jupyter. Se fosse usada a função interna `print()` do Python, saída não seria renderizada dessa forma. Compare a saída do exemplo acima com o exemplo 2.4.

Um outro outro artifício que usamos com frequência quando estamos manipulando expressões simbólica, é a simplificação dessas expressões ao longo dos cálculos. O SymPy possui uma função geral para fazer simplificações chamada de `sympy.simplify()`<sup>11</sup>, onde passamo a expressão simbólica que queremos simplificar para o argumento dessa função. Vejamos um exemplo sobre isso.

**Exemplo 2.6 :** Suponha que temos a seguinte função  $f(x) = \frac{(2x+3)(10x-5)}{2x-1}$ , e queremos simplificar essa expressão, fazemos da seguinte forma :

```
In [7]: import sympy as sp
x = sp.symbols('x', real = True )
# criando a função :
fun = ((2*x + 3)*(10*x - 5))/(2*x - 1)
# mostrando a função :
fun
Out[7]: (2x + 3)(10x - 5)
                    2x - 1

In [8]: # simplificando a expressão usando a função simplify()
fun = sp.simplify(fun)
# mostrando a simplificação :
fun
Out[8]: 10x + 15
```

Figura 11: Usando a função `simplify()` para simplificar uma expressão.

SymPy faz simplificações mais complicadas que a mostrada acima, vejamos mais um exemplo usando agora expressões trigonométricas.

**Exemplo 2.7 :** Suponha agora que temos a seguinte função  $f(x) = \frac{1 - \sin^2(x)}{\tan^2(x) + 1}$ , e queremos simplificar essa expressão, fazemos da seguinte forma :

---

<sup>11</sup>Existem outras funções para simplificações mais específicas, que irá depender da forma de sua expressão simbólica, como funções de simplificações mais rápidas para funções racionais por exemplo. Para uma descrição mais detalhada, consulte <https://docs.sympy.org/latest/tutorials/intro-tutorial/simplification.html>.

```
In [9]: import sympy as sp
x = sp.symbols('x', real = True)

# criando a função :
fun = (1 - sp.sin(x)**2)/(sp.tan(x)**2 + 1)

# mostrando a função :
fun

Out[9]: 
$$\frac{1 - \sin^2(x)}{\tan^2(x) + 1}$$


In [10]: # simplificando :
fun = sp.simplify(fun)

# mostrando a função simplificada
fun

Out[10]: 
$$\cos^4(x)$$

```

Figura 12: Usando a função `simplify()` para simplificar uma expressão.

## 2.2 Outras Funções e Métodos Úteis

Nessa seção veremos como usar alguns outros método e funções que estão implementadas no SymPy, que podem aparecer frequentemente na resolução de problemas. Até agora vimos como criar e manipular expressões simbólicas, mas se estivermos interessados em calcular o valor numérico de uma expressão simbólica?, para isso usamos uma combinação do método `subs()` que aprendemos em seções anteriores e o método `evalf()`.

### 2.2.1 O Método `evalf()`

Quando estivermos interessados em saber o valor numérico de um expressão simbólica, podemos usar o método `evalf()`. Esse método recebe um *argumento opcional* que indica a precisão do valor numérico que se quer obter, onde é passado um valor *inteiro* para esse método, indicando o tamanho da resposta numérica. Para que isso fique mais claro, vejamos um exemplo onde podemos usar esse método.

**Exemplo 2.8 :** Imagine que temos a seguinte expressão simbólica  $f(x) = \frac{\sin(2x) + \cos(2x)}{e^{3x}} + 2x^2$ , e queremos avaliar essa expressão em  $x = \pi$  com uma precisão de 4 casas decimais, isso pode ser feito da seguinte forma :

```
In [11]: import sympy as sp
x = sp.symbols('x', real = True)
fun = (sp.sin(2*x) + sp.cos(2*x))/(sp.exp(3*x)) + 2*x**2
# mostrando a função :
fun

Out[11]: 
$$2x^2 + (\sin(2x) + \cos(2x))e^{-3x}$$


In [12]: # avaliando a função em pi :
valor = fun.subs({x : sp.pi}).evalf(6)    # tamanho do valor numérico
valor

Out[12]: 19.7393
```

Figura 13: Usando o método `evalf()` para obter um valor numérico.

Agora se quisermos calcular uma expressão simbólica para vários valores ao mesmo tempo?, ou seja, calcular a expressão para um *vetor* de valores?. Poderíamos usar um laço sobre o dado vetor e usar os comandos aprendidos nas seções anteriores para calcular a função em cada ponto. Só que existe um pequeno problema nisso, dependendo do tamanho do vetor, usar repetições pode deixar o programa mais lento, para evitar isso em problemas de grande porte, o SymPy tem uma função interna chamada de `lambdify()`, que pega uma expressão simbólica e a transforma em uma função *vetorizada*<sup>12</sup>, similarmente a funções da biblioteca do *Numpy*.<sup>13</sup> Vejamos como fazer isso de duas formas.

### 2.2.2 A Função `lambdify()`

Essa função nos permite calcular o valor de uma dada expressão simbólica de forma *vetorizada*, isto é, receber um vetor como argumento e retornar um vetor de valores calculados para cada valor de entrada. Esse processo é útil, pois, exclui a necessidade de criação de laços para calcular uma dada expressão para cada componente de um vetor. Antes de começar a colocar a mão na massa, é necessário instalar o pacote de computação numérica *Numpy* em seu computador, para que a função `lambdify()` possa funcionar corretamente para um vetor. Para instalar esse pacote, vá até o terminal e digite o seguinte comando :

- `pip install numpy`

Como não abordaremos esse pacote nessa apostila, o único comando que iremos usar é o `numpy.linspace(x_i,x_f,num)`. Onde `x_i` indica um valor inicial do vetor, `x_f` o valor final do vetor e `num` a quantidade de elementos que contém entre `x_i` e `x_f`. Por convenção a biblioteca do numpy é importada da seguinte forma : `import numpy as np`. Agora vejamos um exemplo usando a função `lambdify()`.

**Exemplo 2.9 :** Suponha que temos a seguinte função  $f(x) = \sqrt{x} + e^{2x}$  e queremos calcular o valor de  $f(x)$  para cada valor em  $\vec{v} = [1, 2, 3, \dots, 10]^T$ . Programa em Python para isso é :

---

<sup>12</sup>Um função *vetorizada* é uma função que recebe um vetor de valores e retorna um outro vetor de valores, onde cada valor de retorno é obtido substituindo um valor de entrada na expressão da função.

<sup>13</sup>*Numpy* é uma biblioteca do Python voltada para computação numérica e científica. O principal objeto dessa biblioteca é um arranjo homogêneo Multidimensional que é usado para representar vetores e matrizes. As funções que atuam sobre esses objetos, são aplicadas sobre cada elemento desses vetores e matrizes, ou seja, são funções *vetorizadas*.

Nessa apostila não estudaremos sobre essa biblioteca mas se o leitor estiver interessado em aprender mais sobre o Numpy, uma excelente fonte de conhecimento é a sua documentação oficial que se encontra no seguinte endereço [https://numpy.org/doc/1.23/user/absolute\\_beginners.html](https://numpy.org/doc/1.23/user/absolute_beginners.html).

```
In [13]: import sympy as sp
import numpy as np

x = sp.symbols('x', real = True )
fun = sp.sqrt(x) + sp.exp(2*x)

vetx = np.linspace(1,10,10)

# vetorizando a função simbolica :
fun_vet = sp.lambdify(x, fun,'numpy')

# calculando a função para cada valor na lista :
valores = fun_vet(vetx)

valores
```

Out[13]: array([8.38905610e+00, 5.60123636e+01, 4.05160844e+02, 2.98295799e+03,
 2.20287019e+04, 1.62757241e+05, 1.20260693e+06, 8.88611335e+06,
 6.56599721e+07, 4.85165199e+08])

Figura 14: Uso da função `lambdify()`.

### 2.2.3 Criando Igualdades em SymPy

Quando estamos trabalhando com matemática simbólica, frequentemente nos deparamos com o problema de resolver uma equação ou um sistema de equações lineares, afim de encontrar as expressões para as variáveis do problema. Todos esses casos utilizam igualdades nas expressões e embora não vamos aprender a resolver esses agora, apenas em seções posteriores, aprenderemos como criar igualdades simbólicas no SymPy. Para construir uma igualdade usamos a função `Eq()` do SymPy, onde passamos como primeiro argumento a expressão que se encontrará ao lado esquerdo do sinal de `=`, e o segundo argumento a expressão que ficará ao lado direito desse sinal.

Vejamos um exemplo básico da criação de igualdades em SymPy.

**Exemplo 2.10 :** Crie a seguinte igualdade em SymPy :  $\operatorname{tg}^2(x) + 1 = \sec^2(x)$

```
In [14]: import sympy as sp

x = sp.symbols('x', real = True)

expr1 = sp.tan(x)**2 + 1 # expressão do lado esquerdo

expr2 = sp.sec(x)         # expressão do lado direito

equacao = sp.Eq(expr1,expr2)

equacao
```

Out[14]:  $\tan^2(x) + 1 = \sec(x)$

Figura 15: Criando uma equação simples em SymPy.

Agora vejamos como criar um sistema simples  $2 \times 2$  em SymPy. Nesse caso, cada equação será um elemento de uma lista em Python, como temos duas equações, então a lista terá dois elementos.

**Exemplo 2.11 :** Crie o seguinte sistema de equações lineares em SymPy :

$$5x + 2y = 10$$

$$3x - y = 0$$

Isso pode ser feito da seguinte maneira no SymPy :

```
In [15]: import sympy as sp
x = sp.symbols('x', real = True)
y = sp.symbols('y', real = True )
eq1 = sp.Eq(5*x + 2*y, 10)      # primeira equação
eq2 = sp.Eq(3*x - y, 0 )        # segunda equação
sistema = [eq1, eq2]
sistema
Out[15]: [5x + 2y = 10, 3x - y = 0]
```

Figura 16: Criando um sistema equações lineares simples em SymPy.

### 3 Plotagem com SymPy

Nessa seção veremos como fazer gráfico de funções em SymPy. A classe de plotagem do SymPy encontra-se no submódulo `sympy.plotting`, onde encontramos as seguinte funções para esboçar gráficos :

- `plot()` : Plota gráficos de linha 2D.
- `plot_parametric()` : Plota gráficos 2D paramétricos.
- `plot_implicit()` : Faz plotagem 2D implícitas e de regiões.
- `plot3d()` : Plota gráficos de funções de 2 variáveis.
- `plot3d_parametric_line()` : Plota gráficos de linhas 3D, definidas por um parâmetro.
- `plot3d_parametric_surface()` :Plota gráficos de superfície paramétrica 3D.

Antes de começarmos a colocar a mão na massa, é importante ressaltar que a classe de plotagem do Sympy não é a melhor para construir plotes bem específicos, isso porquê, o Sympy foi criado para resolver problemas matemáticos no geral, então a classe de plotes não é tão geral quanto o pacote *Matplotlib*<sup>14</sup> por exemplo. Sendo assim, não fique frustrado se não conseguir fazer plotagem da forma que você deseja, nesse caso, tente estudar o pacote mencionado anteriormente no seguinte link (<https://matplotlib.org/>). Uma outra coisa importante que seguiremos que faremos com frequência ao longo dos próximos códigos que escreveremos é importar apenas funções específicas de uma dado módulo ou biblioteca do SymPy. Isso pode ser feito da seguinte forma : `from nome_modulo import func1, func2 ,..., funcn` . Onde `func1, func2 ,..., funcn` são as funções que pertencem ao módulo '`nome_modulo`'.

Para usar um *sub-módulo* de um dado módulo, usamos a seguinte sintaxe em Python/SymPy : `from modulo.name_submodulo import func1, func2 ,..., funcn`. Construir scripts dessa forma ajuda a economizar memória do computador e aumentar o seu desempenho também.

Dito tudo isso, vamos começar a realmente a construir gráficos com o SymPy.

#### 3.1 Plote de uma função de uma variável

Para fazer o plote de uma função de apenas uma variável  $f(x)$ , usamos o comando `plot()`, onde passamos como argumento dessa a função que queremos plotar(ela deve ser uma função simbólica!). Por *default*, o comando `plot()` realiza o gráfico da função desejada no intervalo de  $[-1, 1]$ , mas podemos mudar o *range* do plot, passando uma tupla da seguinte forma : `(var,valor_inicial,valor_final)`. Onde '`var`' representa a *variável independente da função*, '`valor_inicial`' a extremidade esquerda do intervalo e '`valor_final`' a extremidade direita do intervalo.

---

<sup>14</sup>É um pacote de Python voltado para criação de gráficos mais gerais possíveis, sendo muito usado por cientistas de várias áreas.

Para entendermos como criar gráficos usando SymPy, vamos fazer o gráfico da função  $f(x) = x^2$  em  $[-10, 10]$ .

**Exemplo 3.1** Plotando a função  $f(x) = x^2$  em  $[-10, 10]$  com Sympy.

```
In [1]: '''
Descrição : plotando um gráfico simples da
função f(x) = x² em [-10,10]

'''

from sympy import symbols
from sympy.plotting import plot

# criando a função a ser plotada :
x = symbols('x', real = True )
fun = x**2

# plotando essa função :
plot(fun,(x,-10,10))
```

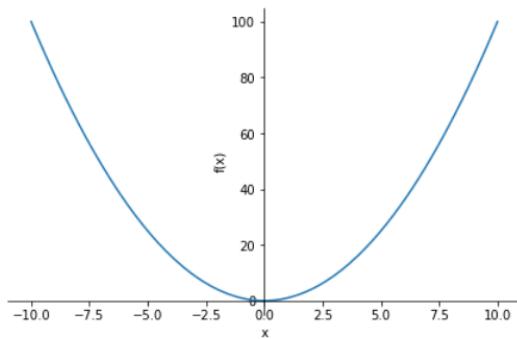
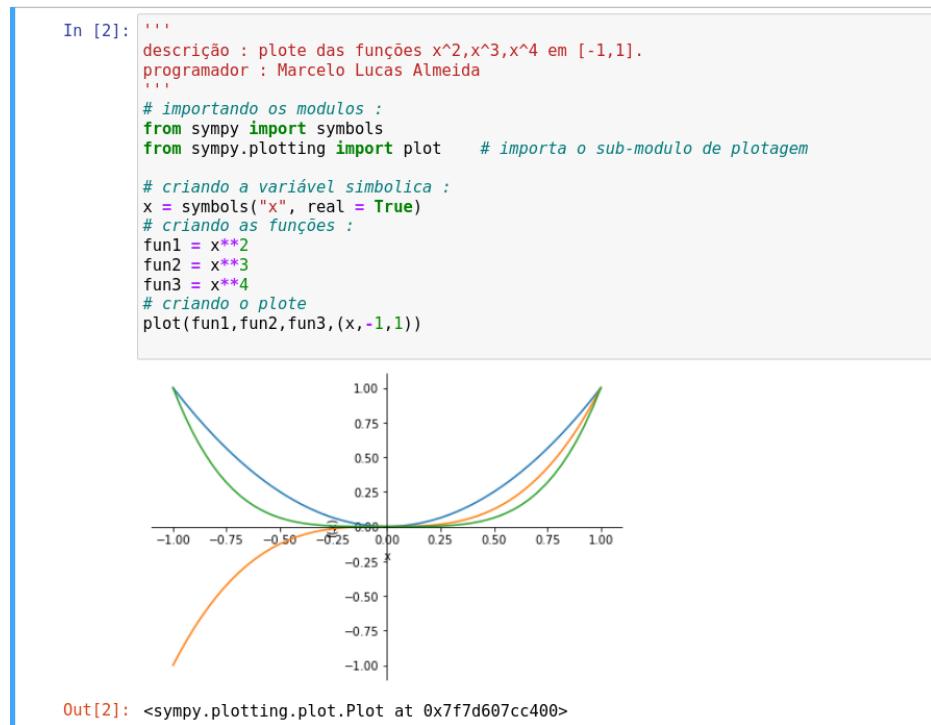


Figura 17: plote da função  $f(x) = x^2$ .

Podemos também fazer vários plots de funções em uma mesma janela gráfica, apenas passando cada função separada por vírgula para `plot()`. Vejamos um exemplo disso.

**Exemplo 3.2** Plote das funções  $x^2, x^3, x^4$  em uma mesma janela gráfica

Figura 18: plote das funções  $x^2, x^3, x^4$ .

Perceba que o Sympy muda a cor automaticamente dos gráficos, isso pode ser feito de forma manual e veremos isso na próxima seção, onde aprenderemos a personalizar nossos gráficos.

### 3.2 Personalizando os Gráficos em Sympy

Nessa seção aprenderemos a personalizar nossos gráficos no Sympy, a fim de torná-los mais atrativos para assim, entendermos melhor o comportamento das funções que estão sendo plotadas. Como sabemos o comando `plot()` do módulo `sympy.plotting`, faz a plotagem de funções de 1 variável, ou seja, plota as funções no *plano cartesiano*. Podemos passar alguns argumentos para essa função, que modificará a estética dos nossos gráficos. Por exemplo, podemos passar um argumento que mudará a cor da nossa curva, ou um argumento que colocará uma legenda. Abaixo encontra-se alguns argumentos que podemos passar para a função `plot()`.

- **title** : esse argumento insere o título do gráfico (*string*)
- **xlabel** : faz a nomeação do eixo horizontal do gráfico (*string*)
- **ylabel** : faz a nomeação do eixo vertical do gráfico (*string*)
- **legend** : coloca a legenda no gráfico (*booleano*)
- **show** : permite que o gráfico seja mostrado assim que executado o comando de plotagem, por *default* é definido como `True` (*booleano*)
- **axis** : permite mostrar os eixos cartesianos (*booleano*)
- **xlim** : permite controlar o tamanho dos valores no eixo horizontal (*tupla de valores*)

- **ylim** : permite controlar o tamanho dos valores no eixo vertical (*tupla de valores*)
- **size** : permite definir o tamanho da figura (*tupla de valores*)
- **line\_color** : define a cor do gráfico (*string ou tupla de valores*)

Através desses argumentos podemos melhorar nossos gráficos. Vejamos como criar o gráfico da função  $f(x) = \sin(x)$  em  $[-\pi, \pi]$ , modificando a sua cor para verde, criando rótulos para os eixos, colocando um título e criando uma legenda.

**Exemplo 3.3** *Plete da função  $f(x) = \sin(x)$  em  $[-\pi, \pi]$ .*

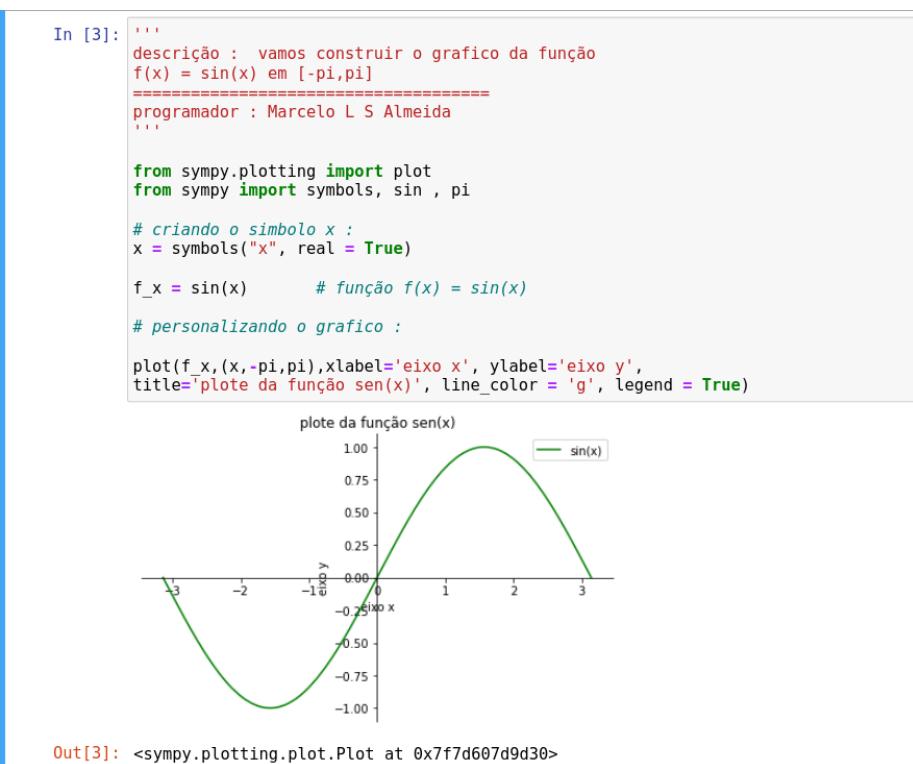
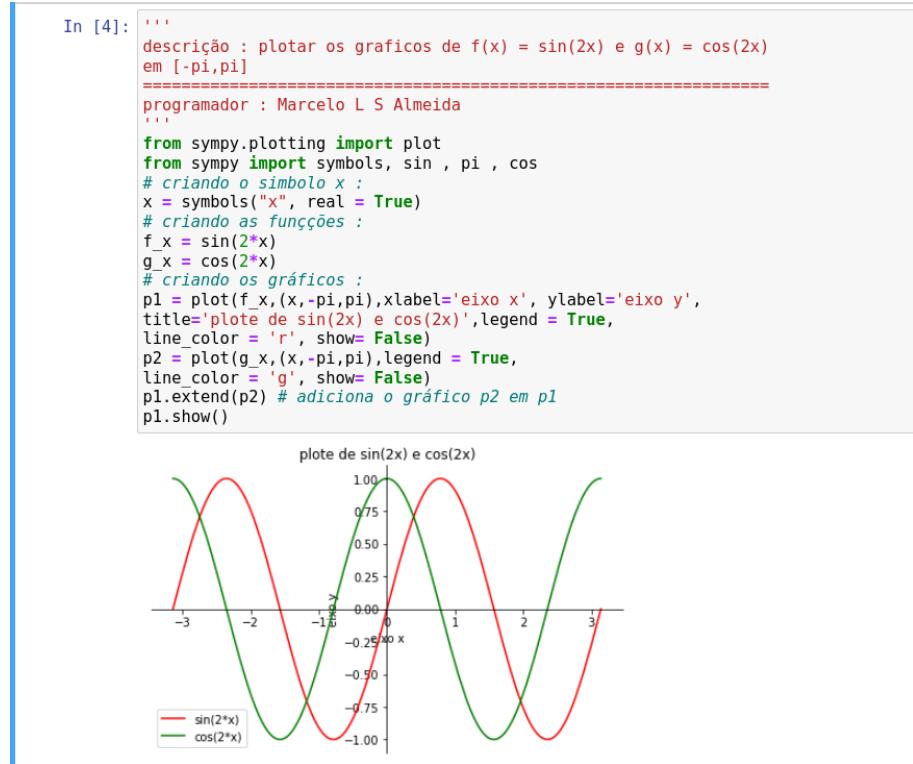


Figura 19: pote da função  $\sin(x)$

Quando queremos definir características para vários gráficos em uma mesma janela de plotagem(ou seja, usando um único comando `plot()`), os argumentos passados para essa função se aplicaram de forma idênticas aos plots das funções. Para definir os parâmetros de forma correta, teremos que usar outra abordagem, usando o **método extend()**, que adiciona um gráfico ao um outro gráfico já existente. Dessa forma podemos criar dois plots que tenham as suas características desejadas, e depois basta usar esse método para unir esse dois gráficos em um só gráfico. Para entender melhor como isso funciona, vamos fazer o pote das funções  $f(x) = \sin(2x)$  e  $g(x) = \cos(2x)$  em  $[-\pi, \pi]$ , e vamos personalizar cada um individualmente.

**Exemplo 3.4** *Plete de duas funções usando o método `extend()`.*

Nesse momento, alguns comentários sobre o script acima são válidos. Em primeiro, tudo em Python é um *objeto*, sendo assim, podemos atribuir um *objeto de plot* para uma variável, como fizemos em `p1 = plot(...)` e `p2 = plot(...)`. Fazer isso, garante que

Figura 20: plote das funções  $\sin(2x)$  e  $\cos(2x)$ 

ambos os gráficos terão estilos diferentes, como por exemplo, a cor e a legenda. Como ambos os gráficos vão ser plotados em um mesmo *range*, não é necessário escrever em cada comando `plot` os comandos para inserir os rótulos dos eixos, sendo necessário escrever apenas em um. Outra coisa importante é o comando `show = False` em ambos os plots, isso faz com que Sympy não faça o gráfico da função assim que encontrar o comando `plot()`, sendo necessário usar `.show()` na variável que contém ambos os gráficos, aí sim é mostrado os gráficos.

Agora que já temos as ferramentas para criar gráficos mais elaborados, vejamos um exemplo mais interessante usando o que acabamos de aprender. Vamos plotar o gráfico da função  $f(x) = e^x$  e os seus 4 termos em *série de Taylor*<sup>15</sup> centrada em  $x_0 = 0$ <sup>16</sup>. Para isso a fórmula da série de Taylor é dada por :

$$T(x) = \sum_{k=0}^{\infty} \frac{f^{(k)}(x_0)(x - x_0)^k}{k!}$$

Para a função exponencial, a série de taylor em  $x_0 = 0$  fica :

$$e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!}$$

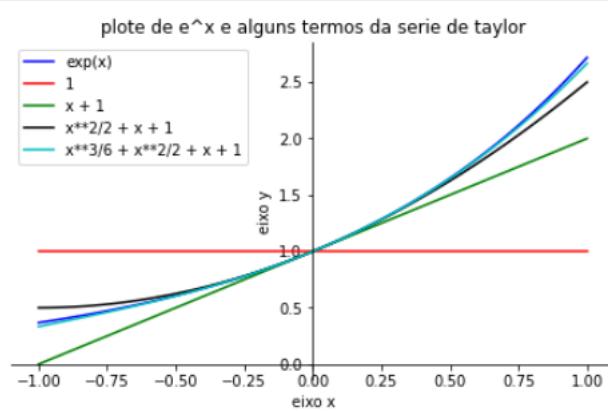
Então o código para plotar a função e os seus 4 termos da série, fica :

**Exemplo 3.5** Série de Taylor de  $f(x) = e^x$  e seus 4 termos.

<sup>15</sup>Para um melhor entendimento matemático do assunto consulte as referências : [6],[7], [8].

<sup>16</sup>Também chamada de *Série de Maclaurin*.

```
In [7]: """
Descrição : vamos plotar o gráfico da função f(x) = e^x e
o seus quatros termos da série de taylor em x0 = 0 em [-1,1]
=====
programador : Marcelo L S Almeida
"""
from sympy.plotting import plot
from sympy import symbols , exp
import math as mt
# criando os simbolos :
x = symbols("x", real = True)
k = symbols("k", integer = True)
# criando as funções :
funcao = exp(x)    # função exponencial
expr1 = (x**k) # função que representa o numero da serie de taylor de exp(x)
expr2 = k # denominador da expressão da serie de taylor de exp(x)
# criando uma lista :
lista = [0,1,2,3] # termos da série
lista_cor = ['r','g','k','c'] # lista de cores
# criando o plote que receberá os termos da serie :
p = plot(funcao,(x,-1,1), xlabel='eixo x', ylabel='eixo y',
title='plote de e^x e alguns termos da serie de taylor',
line_color = 'b', legend = True, show=False)
# criando um laço para plotar :
valor = 0 # acumula as funções
for i in range(len(lista)):
    valor += expr1.subs({k:lista[i]})/mt.factorial(expr2.subs({k:lista[i]}))
    p1 = plot(valor,(x,-1,1), line_color = lista_cor[i],
    legend = True, show=False)
    p.extend(p1)
p.show()
```

Figura 21: Série de Taylor de  $e^x$  com 4 termos (código).Figura 22: Série de Taylor de  $e^x$  com 4 termos (imagem).

### 3.3 Plote de Funções Paramétricas

Nesse seção veremos como plotar curvas definidas por *equações paramétrica*. Uma curva paramétrica é uma função  $\gamma : I \rightarrow \mathbb{R}^2$  ou  $\gamma : I \rightarrow \mathbb{R}^3$ , com  $I \subset \mathbb{R}$ . O que diferencia uma curva *parametrizada* de um gráfico de uma *função*, é que as coordenadas do ponto no plano ou no espaço dependem de uma terceira variável, chamada de *parâmetro*. Assim uma curva paramétrica tem o seguinte aspecto :  $\gamma(t) = (x(t), y(t))$  para uma curva em  $\mathbb{R}^2$  e  $\gamma(t) = (x(t), y(t), z(t))$  para uma curva em  $\mathbb{R}^3$ .

Para plotar uma curva parametrizada no plano, usamos a função `plot_parametric()`, onde passamos as funções  $x(t)$  e  $y(t)$  e o intervalo do parâmetro  $t$ . Como um primeiro exemplo, vamos plotar a curva paramétrica definida pela seguinte função :

$$\gamma(t) = (\sin(t), \sin(t)\cos(t))$$

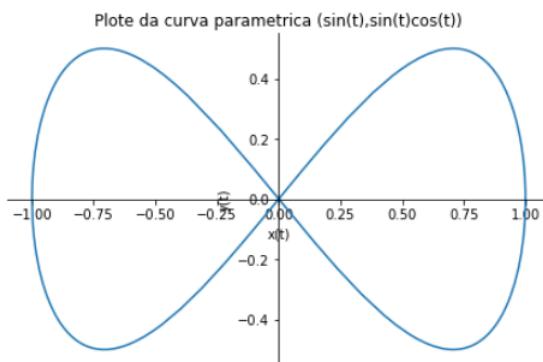
Com  $t \in [0, 2\pi]$ .

**Exemplo 3.6** *Plote da função paramétrica  $\gamma(t) = (\sin(t), \sin(t)\cos(t))$ , com  $t \in [0, 2\pi]$ .*

```
In [1]: '''
Descrição : plote da curva parametrica de equações : (sin(t),sin(t)cos(t))
com t em [0,2pi]
=====
Programador : Marcelo Almeida
'''

from sympy.plotting import plot_parametric
from sympy import symbols, pi, sin, cos

# criando as funções :
t = symbols('t', real = True ) # parametro
x_t = sin(t) # x(t)
y_t = sin(t)*cos(t) # y(t)
# plote da curva parametrizada
plot_parametric(x_t,y_t,(t,0,2*pi), xlabel='x(t)', ylabel='y(t)',
title = 'Plote da curva parametrica (sin(t),sin(t)cos(t))')
```



Out[1]: <sympy.plotting.plot.Plot at 0x7f81256d8550>

Figura 23: Plote da função paramétrica :  $\gamma(t) = (\sin(t), \sin(t)\cos(t))$ .

A criação de uma curva paramétrica no plano é simples, passamos primeiro  $x(t) = \sin(t)$  , depois  $y(t) = \cos(t)$  e em seguida passamos o intervalo do parâmetro  $t \in [0, 2\pi]$ .

Um outra curva paramétrica interessante é a *Nó de Trevo*, cuja as equações paramétricas são :

$$\gamma(t) = (\sin(t) + 2 \sin(2t), \cos(t) - 2 \cos(2t))$$

Para  $t \in [0, 2\pi]$ .

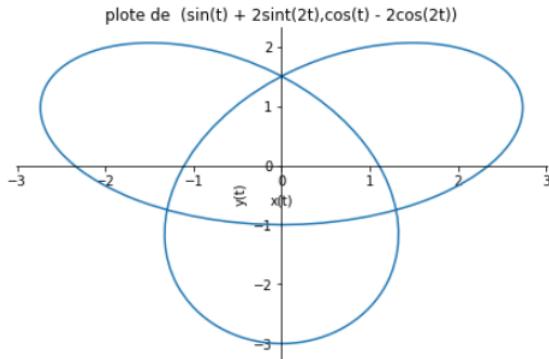
**Exemplo 3.7** Plote da função paramétrica  $\gamma(t) = (\sin(t) + 2 \sin(2t), \cos(t) - 2 \cos(2t))$ , com  $t \in [0, 2\pi]$ .

```
In [2]: """
Descrição : plote da nó de trevo : (sin(t) + 2sin(2t),cos(t) - 2cos(2t))
com t em [0,2pi]
=====
Programador : Marcelo Almeida
"""

from sympy.plotting import plot_parametric
from sympy import symbols, pi, sin, cos

# criando as funções :
t = symbols('t', real = True)
x_t = sin(t) + 2*sin(2*t)      # x(t)
y_t = cos(t) - 2*cos(2*t)      # y(t)

# plote dessa função :
plot_parametric(x_t,y_t,(t,0,2*pi), xlabel='x(t)', ylabel='y(t)',
title='plote de (sin(t) + 2sin(2t),cos(t) - 2cos(2t))')
```



Out[2]: <sympy.plotting.plot.Plot at 0x7f81161b7d60>

Figura 24: Plote da função paramétrica :  $\gamma(t) = (\sin(t) + 2 \sin(2t), \cos(t) - 2 \cos(2t))$ .

E para finalizar, vamos plotar a curva paramétrica definida pelas seguintes equações :

$$\begin{aligned} x(t) &= \sin(t)(e^{\cos(t)} - 2 \cos(4t) - \sin^5(t/12)) \\ y(t) &= \cos(t)(e^{\cos(t)} - 2 \cos(4t) - \sin^5(t/12)) \end{aligned}$$

Com  $t \in [0, 100]$ . Essa curva paramétrica é chamada de *Curva Borboleta*.

**Exemplo 3.8** *Plote da Curva Borboleta.*

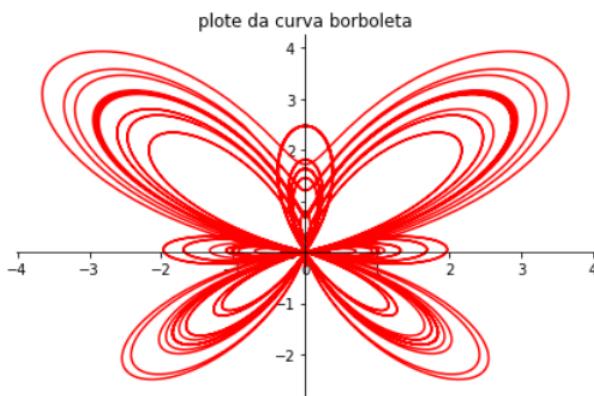
```
In [1]: '''
Descrição : plote da curva borboleta em t = [0,100]
=====
Programador : Marcelo Almeida
'''

from sympy.plotting import plot_parametric
from sympy import symbols, pi, sin, cos, exp

# criando as funções :
t = symbols('t', real = True )
x_t = sin(t)*(exp(cos(t)) - 2*cos(4*t) - sin(t/12)**5)
y_t = cos(t)*(exp(cos(t)) - 2*cos(4*t) - sin(t/12)**5)

# plote da função parametrica :

plot_parametric(x_t,y_t,(t,0,100), title='plote da curva borboleta', line_color = 'r')
```



Out[1]: <sympy.plotting.plot.Plot at 0x7f2c2025eb50>

Figura 25: Plote da Curva Borboleta.

### 3.4 Plote de Funções Implícitas 2D

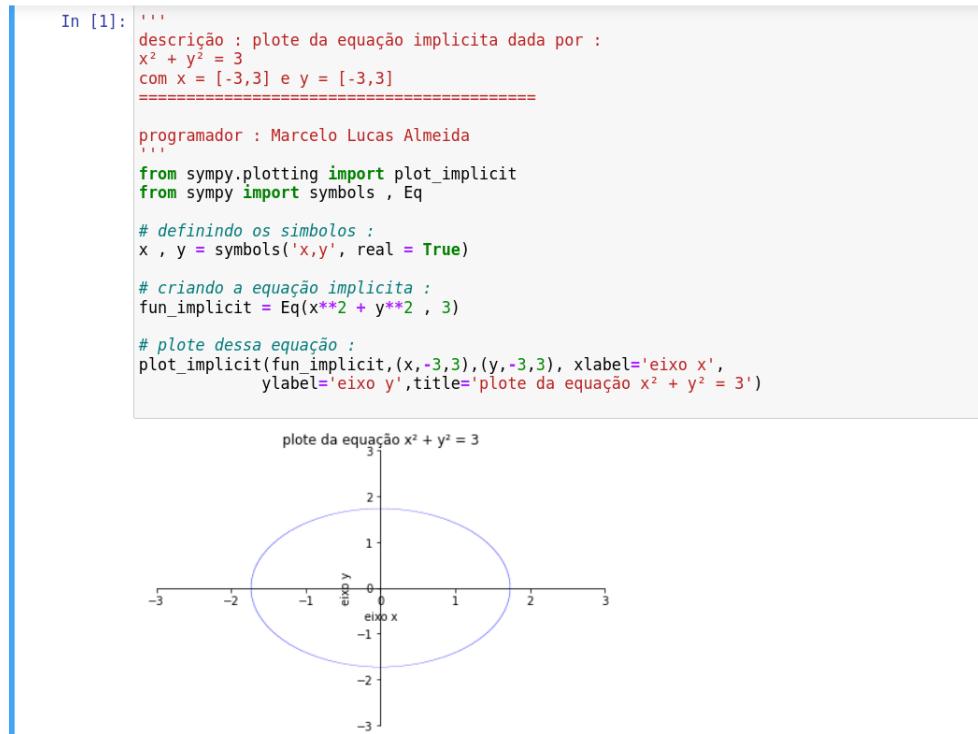
Nesse seção aprenderemos a como fazer plote de funções implícitas, que são úteis quando não conseguimos encontrar uma relação como  $y = f(x)$ <sup>17</sup>. Para fazer gráfico equações implícitas, usamos o `plot_implicit()` do módulo `sympy.plotting`.

Passamos como primeiro argumento dessa função a equação implícita que queremos plotar e logo após os intervalos no *eixo x* e *eixo y*. Vale ressaltar que os comandos para personalização do gráfico funciona também nessa função. Para entendermos melhor como usar essa função, vejamos um exemplo simples abaixo :

**Exemplo 3.9** *Plote da Equação implícita dada por  $x^2 + y^2 = 3$ , com  $x \in [-3,3]$  e  $y \in [-3,3]$ .*

---

<sup>17</sup>Esse tipo de equação está na forma *explicita*.

Figura 26: Plote da equação implícita  $x^2 + y^2 = 3$ .

Mais fácil que isso não há!. Para termos mais afinidade com essa função, vejamos mais alguns exemplos. Podemos plotar gráficos de inequações também com essa função, para isso podemos usar os *relacionais* que o Python possui. Para recordarmos deles, a tabela abaixo especifica cada um deles.

Operador Relacional	Significado
$>$	maior que
$\geq$	maior ou igual a
$<$	menor que
$\leq$	menor ou igual a
$==$	igual a
$\neq$	diferente de

Tabela 1: Tabela de Operadores Relacionais em Python

Agora que sabemos o significado de cada operador relacional, podemos plotar gráficos mais interessantes com a função `plot_implicit()`. Por exemplo vamos plotar o gráfico dado pela seguinte inequação  $y > x^2$ .

**Exemplo 3.10** Plote da Inequação dada por  $y > x^2$ .

```
In [3]: '''
Descrição : plote da inequação dada por y > x²
=====
programador : Marcelo Lucas Almeida
'''
from sympy.plotting import plot_implicit
from sympy import symbols

# criando os símbolos :
x , y = symbols('x,y', real = True)

# plotando a inequação :
plot_implicit(y-x**2, xlabel='eixo x', ylabel='eixo y', title='plote da inequação y > x²')
```

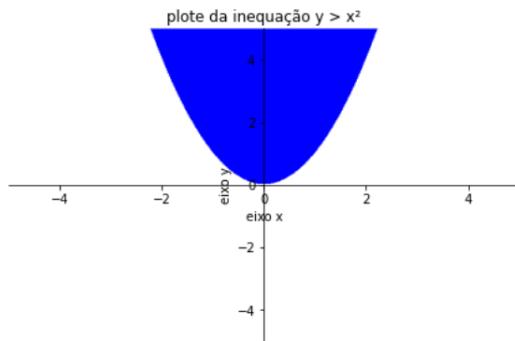


Figura 27: Plote da inequação  $y > x^2$ .

Ainda podemos plotar inequações com mais de uma sentença, por exemplo  $y > x$  e  $y > -x$ , para isso devemos usar a função `And()` do SymPy. Vejamos como fica esse plote.

**Exemplo 3.11** Plote da Inequação dada por  $y > x$  e  $y > -x$ .

```
In [2]: '''
Descrição : plote da inequação dada por y > x e y > -x
=====
programador : Marcelo Lucas Almeida
'''
from sympy.plotting import plot_implicit
from sympy import symbols, And

# criando os símbolos :
x , y = symbols('x,y', real = True)

# plotando a inequação :
plot_implicit(And(y > x , y > -x), title='plote da inequação y > x e y > -x')
```

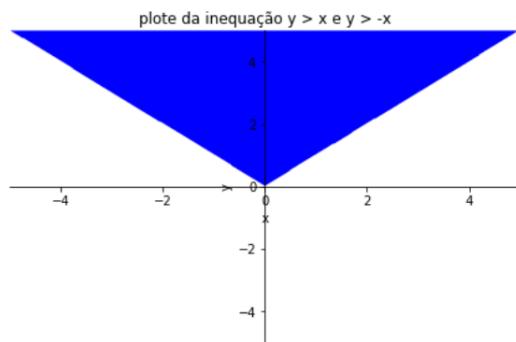


Figura 28: Plote da inequação  $y > x$  e  $y > -x$ .

### 3.5 Plote de Funções de Duas Variáveis

Nessa seção aprenderemos a fazer o gráfico de funções de duas variáveis  $z = f(x, y)$ , usando o comando `plot3d()` da biblioteca `plotting` do SymPy. Para construir o gráfico de uma função  $f(x, y)$  basta apenas passar a função e as informações relativo à formatação do gráfico. Para entender como funcionar essa função, vamos plotar o gráfico de  $f(x, y) = \cos(y) \sin(x)$  em  $[-6, 6] \times [-6, 6]$ .

**Exemplo 3.12** *Plote da função  $f(x, y) = \cos(y) \sin(x)$  em  $[-6, 6] \times [-6, 6]$ .*

```
In [1]: '''
Descrição : plote da função f(x,y) = cos(y)sin(x) em [-6,6]x[-6,6]
=====
Programador : Marcelo Lucas Almeida

'''
from sympy import symbols, sin, exp, cos
from sympy.plotting import plot3d
x, y = symbols('x,y', real = True)

# criando a função f(x,y)
fun_xy = cos(y)*sin(x)

# criando o plote da função
plot3d(fun_xy,(x,-6,6),(y,-6,6))
```

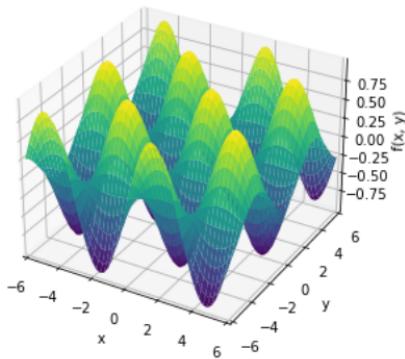


Figura 29: Plote da função  $f(x, y) = \cos(y) \sin(x)$ .

Vemos que criar gráficos de funções de duas variáveis é simples de se fazer com o SymPy. Para fixarmos ainda mais sobre como fazer plote, vejamos como criar o gráfico da seguinte função  $f(x, y) = (4x^2 + y^2)e^{-x^2-y^2}$  em  $[-2, 2] \times [-2, 2]$ .

**Exemplo 3.13** Plote da função  $f(x, y) = (4x^2 + y^2)e^{-x^2-y^2}$  em  $[-2, 2] \times [-2, 2]$ .

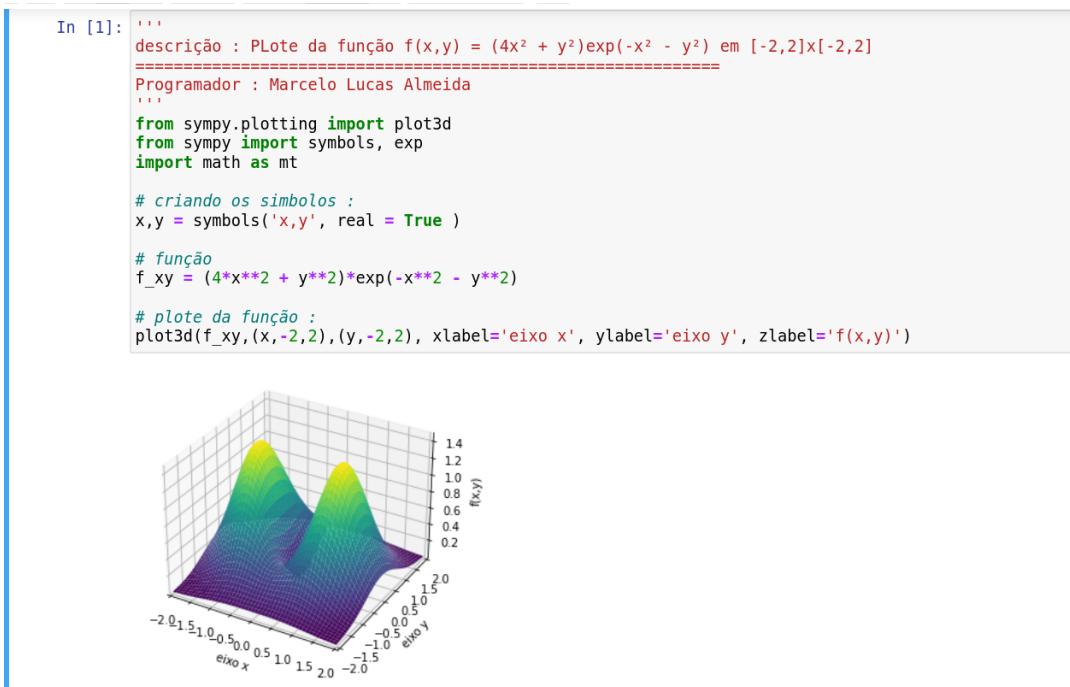


Figura 30: Plote da função  $f(x, y) = (4x^2 + y^2)e^{-x^2-y^2}$ .

Para finalizar essa parte de plote de superfície, vamos fazer os plotes dos seguintes *paraboloides*  $f(x, y) = x^2 + y^2$  e  $g(x, y) = -(x^2 + y^2)$  no mesmo gráfico e na mesma escala, que nesse caso será em  $[-10, 10] \times [-10, 10]$ .

**Exemplo 3.14** Plote das funções  $f(x, y) = x^2 + y^2$  e  $g(x, y) = -(x^2 + y^2)$  em  $[-10, 10] \times [-10, 10]$ .

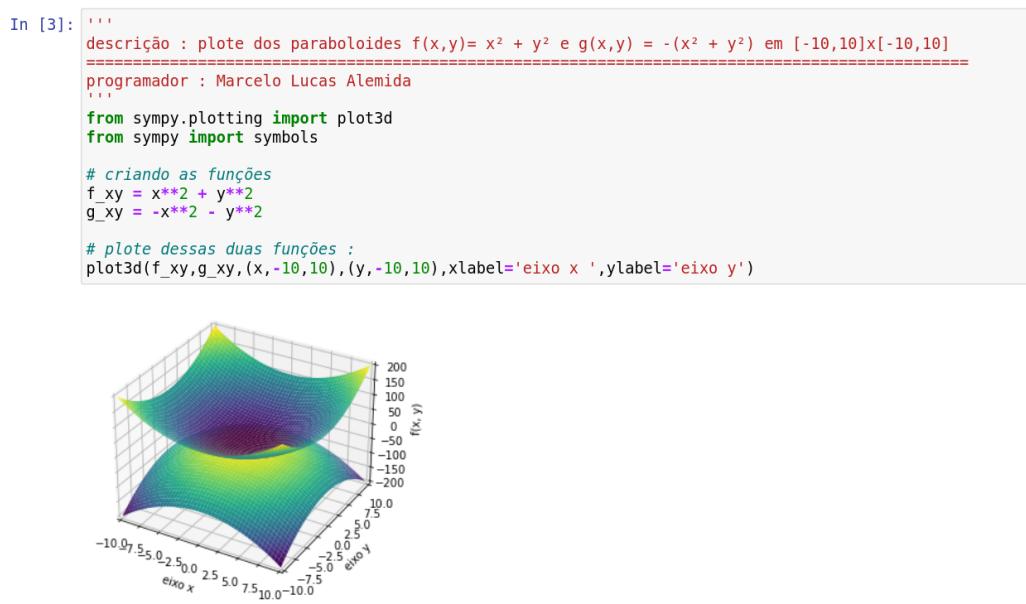


Figura 31: Plote das funções  $f(x, y) = x^2 + y^2$  e  $g(x, y) = -(x^2 + y^2)$ .

### 3.5.1 Plote de Funções Paramétricas no Espaço

Já vimos como criar gráficos de equações paramétricas no plano ( $\mathbb{R}^2$ ), agora vejamos como fazer isso no espaço ( $\mathbb{R}^3$ ). Agora a nossa curva parametrizada terá uma terceira componente que será uma função do parâmetro em questão, assim, nossa curva será  $\gamma = (x(t), y(t), z(t))$ . Para construir o gráfico dessa curva parametrizada, vamos usar a função `plot3d_parametric_line()` do módulo `plotting` do SymPy.

Como primeiro exemplo, vamos traçar a curva paramétrica dada pelas seguintes equações :  $\gamma = (\sqrt{t} \cos(4t), \sqrt{t} \sin(4t), \sqrt{t})$ , onde  $t$  é o nosso parâmetro e varia em  $[0, 2\pi]$ .

**Exemplo 3.15** *Plote da equação paramétrica  $\gamma = (\sqrt{t} \cos(4t), \sqrt{t} \sin(4t), \sqrt{t})$ , com  $t \in [0, 2\pi]$ .*

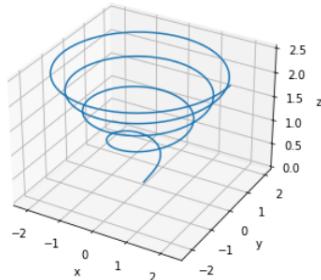
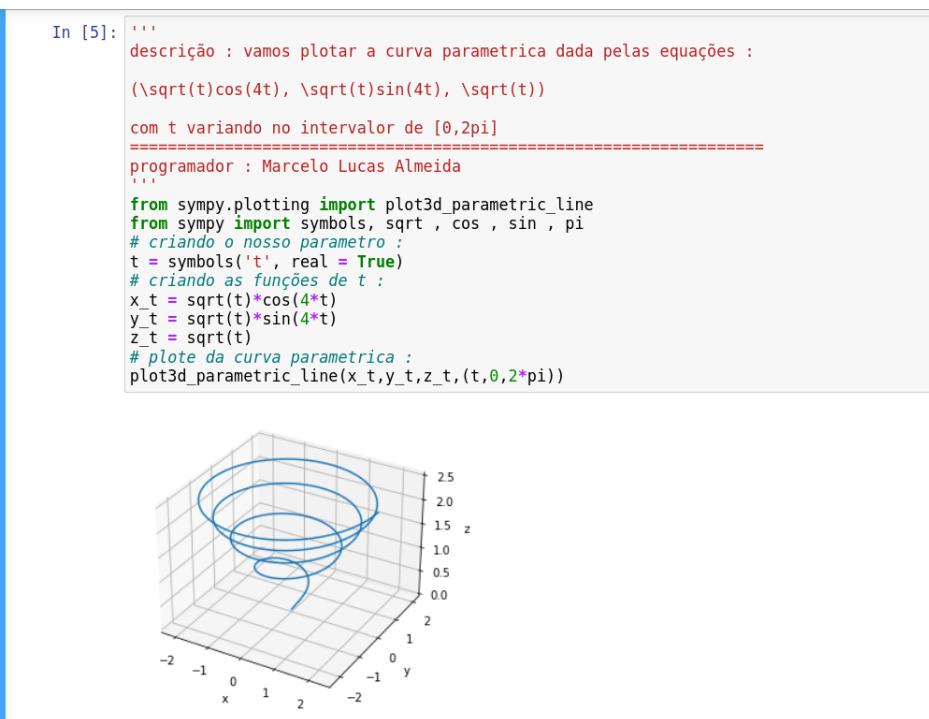


Figura 32: Plote da curva paramétrica  $\gamma = (\sqrt{t} \cos(4t), \sqrt{t} \sin(4t), \sqrt{t})$ .

Os parâmetros que usamos para personalizar nossos gráficos, também podem ser passados para a função `plot3d_parametric_line()`. Essa função pode plotar os mais diversos gráficos de equações paramétricas no espaço tridimensional. Vejamos um exemplos de uma curva paramétrica mais elaborada dada pelas seguintes equações :

$$\gamma = \left( e^{\frac{-|t|}{10}} \sin(5|t|), e^{\frac{-|t|}{10}} \cos(5|t|), t \right)$$

Com  $t$  variando no intervalo de  $[-5, 5]$ .

**Exemplo 3.16** Plote da equação paramétrica  $\gamma = \left( e^{\frac{-|t|}{10}} \sin(5|t|), e^{\frac{-|t|}{10}} \cos(5|t|), t \right)$ , com  $t \in [-5, 5]$ .

```
In [7]: '''
Descrição : vamos plotar a curva paramétrica dada pelas equações :
(e^{|-x|/10}sin(5|x|), e^{|-x|/10}cos(5|t|), t)

com t variando no intervalo de [-5,5]
=====
programador : Marcelo Lucas Almeida
'''

from sympy.plotting import plot3d_parametric_line
from sympy import symbols, exp , sin , cos

# criando o parametro :
t = symbols('t', real = True)
# criando as funções de t :
x_t = exp(-abs(t)/10)*sin(5*abs(t))
y_t = exp(-abs(t)/10)*cos(5*abs(t))
z_t = t
#plota da equação paramétrica :
plot3d_parametric_line(x_t,y_t,z_t,(t,-5,5))
```

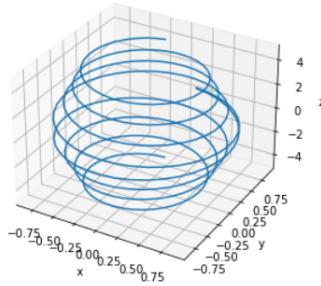


Figura 33: Plote da curva paramétrica  $\gamma = \left( e^{\frac{-|t|}{10}} \sin(5|t|), e^{\frac{-|t|}{10}} \cos(5|t|), t \right)$ .

### 3.5.2 Superfícies Paramétricas

Nesse seção abordaremos as chamadas *superfícies paramétricas*, que podem ser plotadas no SymPy através do comando `plot3d_parametric_surface()` do módulo `plotting`. Antes de começarmos a realizar os plotes, vamos relembrar o que é uma superfície paramétrica. Seja  $\mathbb{D}$  uma região do plano  $UV$  e :

$$\vec{r} = \vec{r}(u, v) = x(u, v)\vec{i} + y(u, v)\vec{j} + z(u, v)\vec{k}$$

Uma função vetorial de  $\mathbb{D}$  em  $\mathbb{R}^3$ . Quando  $(u, v)$  variam em  $\mathbb{D}$ , os pontos da imagem  $(x, y, z)$  com

$$x = x(u, v) \quad y = y(u, v) \quad z = z(u, v)$$

Descrevem a superfície  $S$ , chamada de *Superfície Paramétrica*. Vejamos um exemplo com SymPy.

**Exemplo 3.17** Plote da Superfície Paramétrica de equação vetorial :  
 $\vec{r}(u, v) = (2 \sin(v) \cos(u), 2 \sin(v) \sin(u), 2 \cos(v))$ , com  $u \in [0, 2\pi]$  e  $v \in [0, \pi]$ .

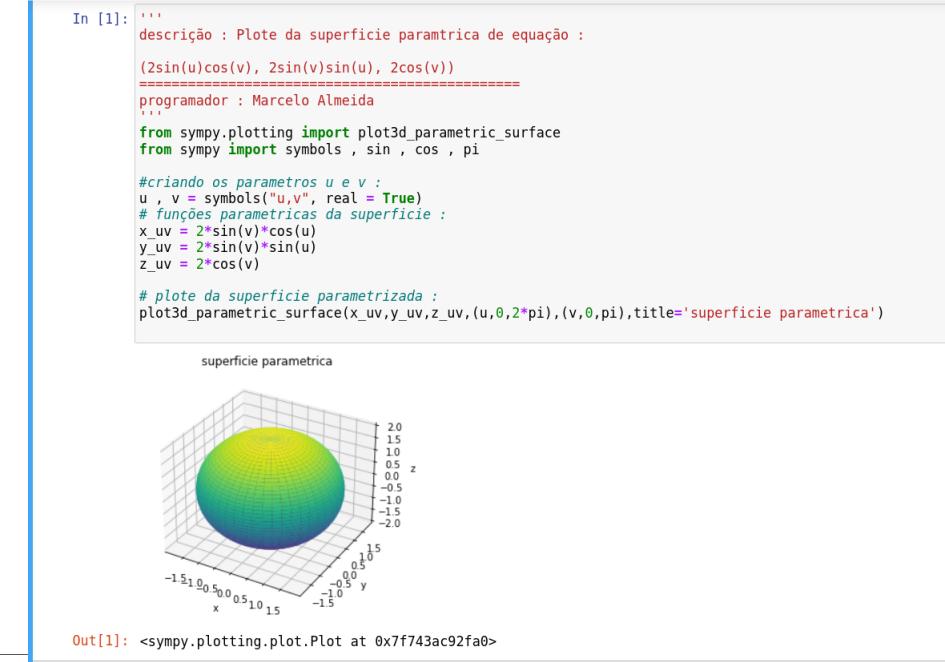


Figura 34: Plote da superfície paramétrica  $(2 \sin(v) \cos(u), 2 \sin(v) \sin(u), 2 \cos(v))$ .

Como podemos observar, trata-se de uma esfera. vamos plotar a superfície chamada de *Hiperbolóide de uma folha* de equações paramétricas dada por :  $(\cos(u) - v \sin(u), \sin(u) + v \cos(u), v)$ , com  $u \in [0, 2\pi]$  e  $v \in [-2, 2]$ .

**Exemplo 3.18** Plote do Hiperbolóide de equações paramétricas : $(\cos(u) - v \sin(u), \sin(u) + v \cos(u), v)$ .

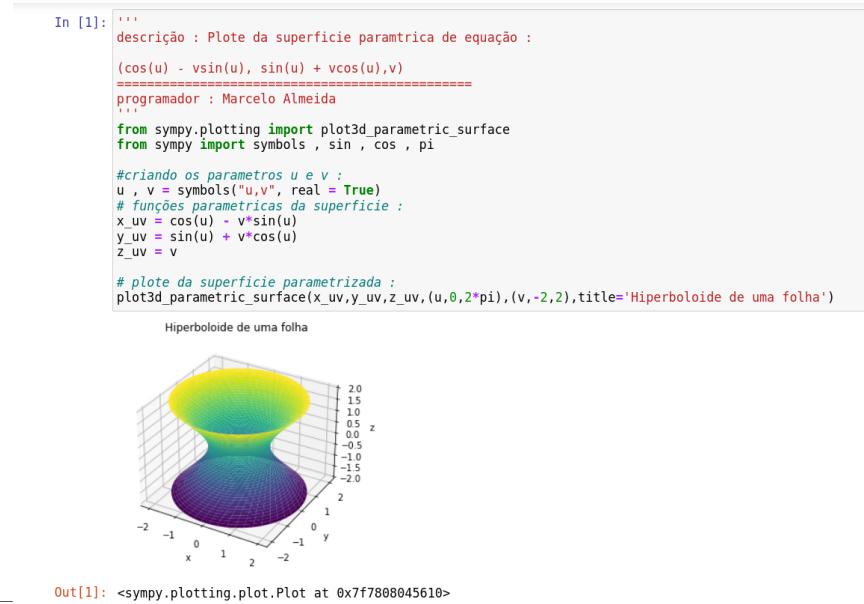


Figura 35: Plote da superfície paramétrica  $(\cos(u) - v \sin(u), \sin(u) + v \cos(u), v)$ .

## 4 Cálculo

Nesse seção abordaremos conceitos relativos ao ramo da matemática pura e aplicada que é uma base de extrema importância para qualquer cientista da área das ciências exatas, que é **Cálculo**. Esse ramo da matemática é dividido em duas partes que possuem uma relação muito bela.<sup>18</sup> A primeira parte trata da *taxa de variação instantânea* de uma função, que geometricamente corresponde a encontrar a inclinação da *reta tangente* ao gráfico da função em um determinado ponto no domínio, as essas problemas usamos o **Cálculo Diferencial**, cuja principal ferramenta matemática é calcular derivadas de uma dada função. A figura 36 mostra essa ideia.

A segunda parte trata do problema de encontrar o valor da área compreendida entre o gráfico de uma função  $f(x)$  definida em algum intervalo  $I$  no eixo  $x$ . Isso pode ser visto na figura 37 logo abaixo.

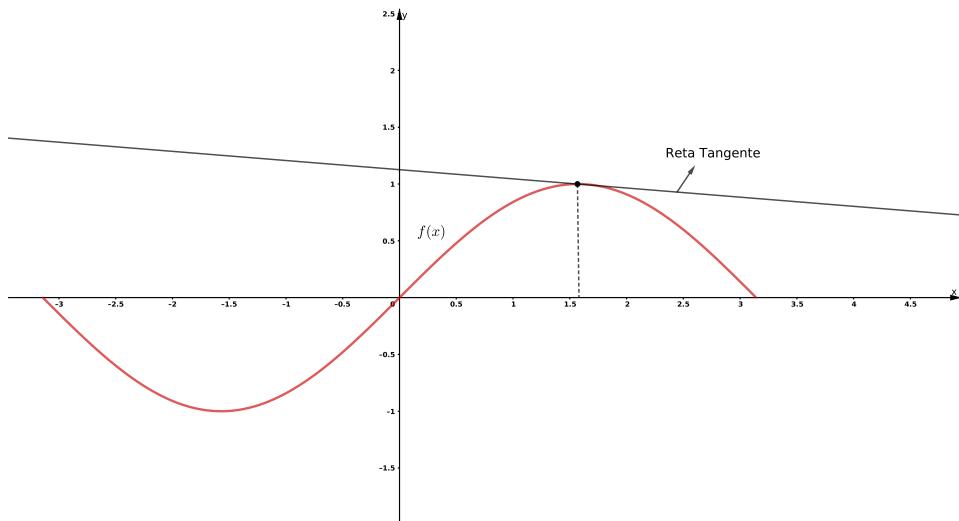


Figura 36: Problema comum do cálculo diferencial

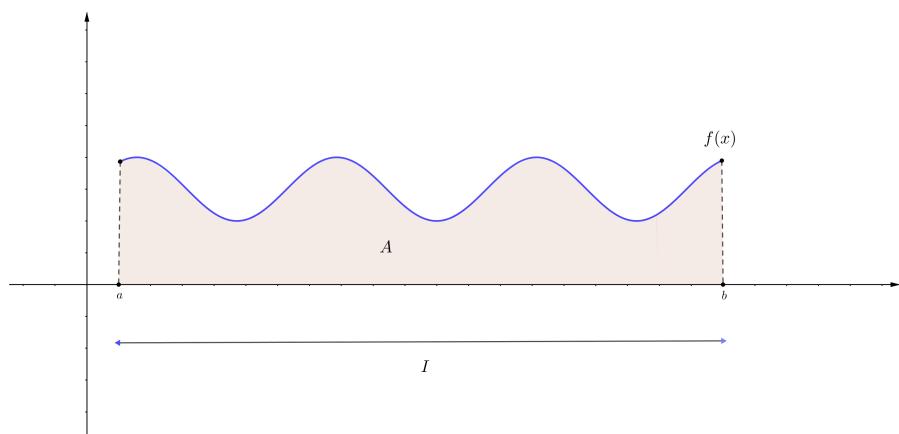


Figura 37: Problema comum do cálculo Integral

<sup>18</sup>Dada pelo *Teorema fundamental do Cálculo*.

Veremos através do Sympy como calcular *limites*, *Derivadas*, *Integrais* e muito mais .

## 4.1 Limites

Os conceitos primordiais do cálculo estão relacionados com o conceito de *limite de uma função*. De forma simples, o limite de uma função nos diz para qual valor a função tende a se aproximar a medida que variável independente se aproxima de um determinado valor em seu domínio. Ou seja, o limite de uma função representa um comportamento funcional. Se o limite de uma função  $f(x)$  existe e é  $L$ , a medida que  $x \rightarrow a$ , então escrevemos :

$$\lim_{x \rightarrow a} f(x) = L$$

No SymPy limites de funções podem ser calculadas facilmente usando duas funções para isso. A primeira é a função `Limit()`, que retorna uma expressão simbólica não avaliada do limite a princípio, mas que pode ser avaliada usando o método `doit()`. A segunda função é `limit()`, que calcula diretamente o valor do limite da função. Ambas as funções recebem 4 parâmetros, onde o primeiro parâmetro corresponde a expressão simbólica que representa a função, o segundo parâmetro é o símbolo correspondente à variável independente, no terceiro parâmetro passamos o ponto para o qual a variável tende e no quarto parâmetro(opcional) passamos uma *string* indicando a direção onde o limite deve ser calculado, isto é , "+" para limite à direita e "-" para calcular o limite à esquerda, se for omitido por padrão será usado "+".

Para ficar mais claro todos esses conceitos vamos verificar o cálculo do seguinte limite em SymPy :  $\lim_{x \rightarrow 0^+} \frac{\sin(x)}{x}$  , que é conhecido como o *limite fundamental trigonométrico*. Usaremos em primeiro lugar a função `Limit()` para criar a expressão simbólica e depois avaliaremos a expressão com o método `doit()`.

**Exemplo 4.1** Avaliando o limite fundamental trigonométrico com a função `Limit()`.

```
In [1]: '''
Descrição : avaliar o limite fundamental trigonométrico
usando a função Limit
=====
Programador : Marcelo Lucas Almeida
'''

from sympy import symbols, init_printing, sin, Limit
init_printing(use_latex=True) # saída formatada em latex
x = symbols('x', real=True) # símbolo / variável independente
fun_x = sin(x)/x # função f(x)

Limit(fun_x,x,0)

Out[1]: lim (sin(x))
           x->0+
```

```
In [2]: # calculando a expressão com o método .doit()
print(f'O valor do limite é = {Limit(fun_x,x,0).doit()}')

O valor do limite é = 1
```

Figura 38: Limite usando `Limit()`.

Como podemos perceber no código do exemplo anterior, na primeira célula (In[1]) definimos a expressão simbólica que representa o limite que queremos calcular, na segunda célula (In[2]) calculamos esse limite usando o método `doit()`, que como sabemos do cálculo esse limite está correto. Esse resultado seria obtido de mesma forma usando a função `limit()`. Como é de se esperar o SymPy tem o poder de calcular limites mais complexos e assim, facilitando a nossa vida dentro dos cálculos. Vejamos um exemplo um pouco mais elaborado.

**Exemplo 4.2** Calcule o seguinte limite no SymPy :

$$\lim_{h \rightarrow 0} \frac{(4+h)^3 - 64}{h}$$

O código é :

```
In [3]: """
Descrição : calculo de limite
=====
Programador : Marcelo Lucas Almeida
"""
from sympy import symbols, init_printing, Limit
init_printing(use_latex=True)
h = symbols('h', real = True) # criando o símbolo h
fun_h = Limit( ((4 + h)**3) - 64)/h, 0
fun_h

Out[3]:  $\lim_{h \rightarrow 0^+} \left( \frac{(h+4)^3 - 64}{h} \right)$ 

In [4]: # avaliando o limite :
print(f"o valor do limite é = {fun_h.doit()}")
o valor do limite é = 48
```

$$\text{Figura 39: Limite de } \lim_{h \rightarrow 0} \frac{(4+h)^3 - 64}{h}.$$

Mas uma vez o SymPy calcula o valor do limite de forma precisa e rápida. Também podemos calcular limites no infinito, onde  $x \rightarrow \pm\infty$ . Para calcular esses tipos de limites só precisamos usar o símbolo de infinito no SymPy, que pode ser usado usando o seguinte comando `sympy.oo`. Fora isso, o restante é similar ao que fizemos nos exemplos anteriores. Vejamos um exemplo usando limites no infinito.

**Exemplo 4.3** Calcule o seguinte limite no SymPy :

$$\lim_{x \rightarrow \infty} \left( 1 + \frac{1}{x} \right)^x$$

O código é :

```
In [6]: '''
Descrição : calculo de limite no infinito
=====
Programador : Marcelo Lucas Almeida
'''

from sympy import symbols, init_printing, Limit, oo

x = symbols('x', real = True)

fun = Limit((1 + 1/x)**x, x, oo)
fun
```

Out[6]:  $\lim_{x \rightarrow \infty} \left(1 + \frac{1}{x}\right)^x$

```
In [7]: # avaliando esse limite
fun.doit()
```

Out[7]:  $e$

$$\text{Figura 40: Limite de } \lim_{x \rightarrow \infty} \left(1 + \frac{1}{x}\right)^x.$$

Na qual esse resultado é o famoso *Número de Euler* ( $e \approx 2,718281$ ).

## 4.2 Somatório e Produtório

Um outro tópico muito importante dentro da matemática aplicada, principalmente na área da *análise numérica* ou *método numéricos*, são as *séries numéricas* que estão intimamente relacionadas ao conceito de *somatório*. O conceito de representar funções complicadas como soma de termos finitos(*séries finitas*) ou como soma de infinitos termos(*séries infinitas*) é de fundamental importância para problemas que usam a modelagem computacional ou numérica. Podemos construir *séries finitas* ou *séries infinitas* em SymPy usando a função `Sum()`. Essa função recebe dois parâmetros de entrada, sendo o primeiro argumento a fórmula geral do  $n$ -ésimo termo da sequência  $a_n$ , e segundo termos é uma *tupla* de valores especificando *limite inferior do somatório* e o *limite superior do somatório*. No caso do limite superior ser  $\infty$ , teremos uma série infinitas. SymPy abusa da computação simbólica para estimar a soma da série se for possível (*convergente*), caso a série seja *divergente*, SymPy retornará  $+\infty$  ou  $-\infty$ .

Como primeiro exemplo, vamos construir uma *série finita* para mostrar a seguinte relação

$$\sum_{k=1}^N k = \frac{N(N+1)}{2}$$

O somatório acima representa a soma dos  $N$  primeiros números inteiros que está relacionado ao *números triangulares*. Vamos construir esse somatório e avaliar para  $N = 100$ . Isso pode ser feito da seguinte forma em SymPy.

**Exemplo 4.4** *Construindo um somatório e calculando a sua soma:*

*O código é :*

```
In [1]: '''
    descrição : construindo um somatório dos N primeiros
    números inteiros e avaliando para N = 100.
=====
programador : Marcelo Almeida
'''
from sympy import symbols, Sum
# criando os simbolos :
k = symbols('k', integer = True, positive = True)
N = symbols('N', integer = True, positive = True)
# criando o somatorio :
somatorio = Sum(k,(k,1,N))
somatorio
```

---

```
Out[1]: 
$$\sum_{k=1}^N k$$

```

```
In [2]: # encontrando a soma sem avaliar numericamente :
somatorio.doit().simplify()
```

```
Out[2]: 
$$\frac{N(N + 1)}{2}$$

```

```
In [3]: # avaliando a soma em N = 100
somatorio.doit().subs({N:100})
```

```
Out[3]: 5050
```

Figura 41: Criando um Somatório.

Em In[1] construímos o somatório dos  $N$  primeiros números inteiros. Já em In[2] avaliamos o somatório usando o método `doit()` intercambiado com o método `simplify()`, essa combinação faz com que o SymPy avalie o somatório e depois simplifique a resposta analítica, que por sinal é a fórmula para a soma dos  $N$  inteiros positivos. Por fim, em In[3] avaliamos o somatório para  $N = 100$ , ou seja, queremos saber quanto vale a soma dos 100 primeiros números inteiros positivos, que nesse caso vale 5050.

Para criar uma *série infinita* seguimos a mesma ideia apresentada acima, mas passamos como limite superior do somatório o valor  $\infty$ , assim indicamos ao SymPy que se trata de uma soma de infinitos termos de uma dada expressão. Como foi dito no início da seção, SymPy não soma infinitos termos para a série, mas utiliza de meios analíticos para encontrar a fórmula(se houver) que representa a *sequência de somas parciais*  $s_n$ . Para ilustrar esse fato, vamos calcular com SymPy a seguinte série e verificar o seu resultado :

$$\sum_{n=1}^{\infty} \frac{1}{n^2} = \frac{\pi^2}{6}$$

Podemos construir fazendo da seguinte forma

**Exemplo 4.5** *Construindo a série e calculando a sua soma:*

O código é :

```
In [4]: '''
    descrição : construindo a serie de 1/n^2 e
    verificar a sua soma
=====
programador : Marcelo Almeida
'''

from sympy import symbols, Sum , oo
# criando os simbolos :
n = symbols('n', integer = True, positive = True)

# criando a serie :
serie = Sum(1/n**2,(n,1,oo))

serie
```

```
Out[4]: 
$$\sum_{n=1}^{\infty} \frac{1}{n^2}$$

```

---

```
In [5]: # calculando a sua soma :

serie.doit()
```

```
Out[5]: 
$$\frac{\pi^2}{6}$$

```

---

$$\text{Figura 42: Série } \sum_{n=1}^{\infty} \frac{1}{n^2}.$$

Mas se uma dada não tiver uma soma?, ou seja, a série é divergente, o que SymPy faz?. Para esses casos o SymPy sabe avaliar e expressar a resposta de forma elegante. Como exemplo vamos calcular a soma da *série harmônica*, que sabemos que é uma série divergente. A série é dada por :

$$\sum_{n=1}^{\infty} \frac{1}{n} = 1 + \frac{1}{2} + \frac{1}{3} + \dots \frac{1}{n} + \dots$$

O código a seguir mostrar que essa série diverge :

**Exemplo 4.6** *Construindo a série e calculando a sua soma:*

*O código é :*

```
In [6]: '''
    descrição : construindo a serie 1/n e verificando a
    sua divergencia
=====
programador : Marcelo Almeida
'''

from sympy import symbols, Sum , oo
# criando os simbolos :
n = symbols('n', integer = True, positive = True)

# criando a serie :
serie = Sum(1/n,(n,1,oo))

serie
```

```
Out[6]: 
$$\sum_{n=1}^{\infty} \frac{1}{n}$$

```

---

```
In [7]: # calculando a sua soma :

serie.doit()
```

```
Out[7]: 
$$\infty$$

```

---

$$\text{Figura 43: Série harmônica.}$$

Podemos construir e avaliar séries mais complicadas, como é o caso da *série de Taylor* de  $\sin(x)$  em  $x_0 = 0$ , dada por :

$$\sin(x) = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n+1}}{(2n+1)!}$$

Iremos aprender a construir a série de Taylor de várias funções na próxima seção de forma mais rápida, mas ilustraremos a construção da mesma usando o função `Sum()`.

**Exemplo 4.7** A série de Taylor de  $\sin(x)$ :

O código é :

```
In [8]: ...
descrição : Criando a serie de taylor se sin(x)
=====
programador : Marcelo Almeida

...
from sympy import symbols, Sum , oo, factorial
# criando os simbolos :
n = symbols('n', integer = True, positive = True)
x = symbols('x', real = True)
# criando expressão :

expr = (((-1)**n)*x**((2*n + 1)))/(factorial(2*n + 1))
serie = Sum(expr,(n,0,oo))

serie

Out[8]: 
$$\sum_{n=0}^{\infty} \frac{(-1)^n x^{2n+1}}{(2n+1)!}$$


In [9]: # avaliando :
serie.doit()

Out[9]: 
$$\sin(x)$$

```

Figura 44: Série de Taylor de  $\sin(x)$ .

Uma outra ferramenta importante é o *produtório*, que pode ser calculado em SymPy usando a função `Product()`. Essa função é similar a função usada para construir somatórios, então não precisarei explicar como deve-se passar os argumentos. Para um exemplo usado o produtório, vamos representar a seguinte identidade :

$$\cos(x) = \prod_{k=1}^{\infty} \left( 1 - \frac{4x^2}{\pi^2(2k-1)^2} \right)$$

que é a representação para  $\cos(x)$ .

então o exemplo que implementa essa identidade para  $x = 0$  e compara com o valor  $\cos(0)$  é :

**Exemplo 4.8** A representação de  $\cos(x)$  por um produtório infinito em  $x = 0$  :

O código é :

```
In [10]: """
Descrição : implementar o produtorio que
aproxima a função coseno em um valor x
=====
Programador : Marcelo Almeida
"""

from sympy import Product, cos, symbols, init_printing, \
pi, oo

# criando os simbolos :
x = symbols('x', real = True)
k = symbols('k', integer = True, positive = True)

# criando a função :
fun = 1 - ((4*x**2)/((pi**2)*(2*k - 1)))

produtorio = Product(fun, (k, 1, oo))
produtorio

Out[10]: 
$$\prod_{k=1}^{\infty} \left( -\frac{4x^2}{\pi^2 \cdot (2k-1)} + 1 \right)$$

```

---

```
In [11]: # avaliando em x = 0 e N = 10 :
valor_appx = produtorio.subs({x:0}).doit()

print('O valor de coseno em x = 0 é = ', cos(0))
print('O valor pelo produtório é = ', valor_appx)

0 valor de coseno em x = 0 é = 1
0 valor pelo produtório é = 1
```

Figura 45:  $\cos(x)$  e seu produtório infinito.

## 4.3 Derivadas

### 4.3.1 Derivadas de Funções de uma variável

Agora vejamos como calcular *derivadas* em SymPy, uma ferramenta de extrema importância dentro de qualquer ciências exatas e tem muita aplicação dentro da física. Assim como no limites de funções, podemos calcular derivada usando duas funções para isso, uma que calcula de forma direta bastando passar a expressão simbólica e uma outra função que cria a expressão da derivada sem avaliar. As funções usadas em SymPy para isso é a função `diff()` e `Derivative()`. Vamos testar isso calculando a derivada da função  $f(x) = \cos(x)$ .

**Exemplo 4.9** Cálculo da derivada de  $f(x) = \cos(x)$  :

```
In [1]: """
Descrição : calcula da derivada de f(x) = cos(x)
=====
Programador : Marcelo Almeida
"""

from sympy import symbols, diff, cos, init_printing, Derivative
init_printing(use_latex=True) # usando latex

x = symbols('x', real = True)
f_x = cos(x)

derivada = Derivative(f_x,x)
derivada

Out[1]: 
$$\frac{d}{dx} \cos(x)$$

```

---

```
In [2]: # mostrando a derivada avaliando com o metodo .doit()
print('A derivada é :')
derivada.doit()
A derivada é :

Out[2]: -sin(x)
```

Figura 46: Derivada da função  $f(x) = \cos(x)$ .

Podemos observar do código acima que o resultado está correto. Nos exemplos a seguir usaremos a função `diff()`, que calcula a derivada da expressão simbólica sem usar o método `doit()`. O SymPy não fica limitado apenas à derivadas de funções simples e nem de funções de uma única variável, a função `diff()` é muito poderosa. Para vermos mais de sua funcionalidade, vamos calcular a derivada da função  $f(x) = e^{2x} \cos(-x)$ .

**Exemplo 4.10** Cálculo da derivada de  $f(x) = e^{2x} \cos(-x)$  :

```
In [3]: '''
Descrição : cálculo da derivada da função f(x) = exp(2x)cos(-x)
=====
Programador : Marcelo Almeida
'''

from sympy import symbols, sin, cos, exp, diff, init_printing, \
    Function, Derivative

init_printing(use_latex=True) # uso do latex

# criando o valor simbólico :
x = symbols('x', real = True)

fun = exp(2*x)*cos(-x)

derivada2 = diff(fun,x,2)

derivada2

Out[3]: (-4 sin(x) + 3 cos(x)) e^{2x}
```

Figura 47: Derivada da função  $f(x) = e^{2x} \cos(-x)$ .

É possível também calcular derivadas de ordem superiores, como derivada de ordem 2, 3 e etc. Para fazer isso passamos a ordem da derivada (inteiro) após a variável independente como : `diff(fun,x,nº da ordem)`.

Como exemplo vamos calcular a derivada de ordem 2 da função  $f(x) = e^{-x} \sin(2x)$  .

**Exemplo 4.11** Cálculo da derivada de  $f(x) = e^{-x} \sin(2x)$  :

```
In [4]: '''
Descrição : calcular a segunda derivada da função f(x) = exp(-x)sin(2x)
=====
Programador : Marcelo Almeida
'''

from sympy import symbols, diff, exp, sin, init_printing
init_printing(use_latex=True)

# criando o simbolo :
x = symbols('x', real = True)

fun = exp(-x)*sin(2*x)

# calculando a segunda derivada :
derivada = diff(fun,x,2)

derivada

Out[4]: -(3 sin(2x) + 4 cos(2x)) e^{-x}
```

Figura 48: Segunda Derivada da função  $f(x) = e^{-x} \sin(2x)$ .

### 4.3.2 Série de Taylor

Certamente um tópico dentro do cálculo que é extremamente importante para qualquer modelagem usando o computador, é a ideia de representar funções mais complicadas como *funções trigonométricas*, *funções logarítmicas* ou composições de ambas em somas infinitas de funções polinomiais, essa ideia é chave para computação numérica em todas as escalas. Essa artifício foi criado graças ao matemático inglês *Brook Taylor*, que mais tarde ficou conhecida como *Série Taylor*. A série de Taylor é :

$$S(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)(x-a)^n}{n!} = f(a) + f'(a)(x-a) + \frac{f''(a)(x-a)^2}{2!} + \dots + \frac{f^{(n)}(a)(x-a)^n}{n!} + \dots$$

Sendo  $f(x)$  uma função diferencial até ordem  $n+1$  em um intervalo  $\mathbf{I}$ .<sup>19</sup> Se o limite superior do somatório for finito, isto é ,  $N < \infty$ , teremos os chamados *Polinômios de Taylor*, e por fim, se  $a = 0$  teremos a *Série de Maclaurin*.

Para representar qualquer expressão simbólica de SymPy como uma série de Taylor, usamos a função `series()`, onde passamos os seguintes argumentos :

- `fun` : Expressão simbólica que representa a função no qual se quer encontrar a série de Taylor.
- `var` : variável independente da função
- `x0` : valor numérico de onde calcular a série de Taylor, por default  $x0 = 0$
- `n` : argumento que indica a quantidade de termos da série.

Como exemplo, vamos representar a função  $f(x) = \cos(x)$  em série de Taylor em torno de  $a = 0$  com SymPy.

**Exemplo 4.12** A representação de  $\cos(x)$  em série de Taylor em  $x = 0$  :

O código é :

```
In [12]: '''
descrição : implementar a serie de taylor de f(x) = cos(x)
em torno de a = 0.
=====
programador : Marcelo Almeida
'''
from sympy import symbols, series, init_printing, cos
init_printing(use_latex=True)

# criando o simbolo :
x = symbols('x', real = True)

# criando a função :
fun = cos(x)

# representação em serie de taylor :
serie_taylor = series(fun,x,x0=0)

serie_taylor
Out[12]: 1 - x2/2 + x4/24 + O(x6)
```

Figura 49:  $\cos(x)$  em série de Taylor.

<sup>19</sup>Ou seja, a função  $f(x)$  deve ser classe  $C^{n+1}(\mathbf{I})$

Onde percebemos que a resposta encontra-se certa, uma vez que a série de Taylor de  $\cos(x)$  admite apenas termos não nulos para valores pares de  $n$ . Por default, SymPy representa a série de Taylor de uma função até uma dada ordem, que no exemplo anterior foi até a quarta ordem e despreza ordem mais altas(por isso o termo  $O(x^6)$ ). Para fins de avaliação numérica ou plote da Série de Taylor, precisamos eliminar o termo  $O(x^6)$  da equação, para isso usamos o método `.removeO()` no resultado obtido e assim podemos realizar os cálculos necessários.

Como um exemplo da aplicação da série de Taylor, vamos construir o gráfico de  $f(x) = \cos(x)$  e alguns termos de sua série em  $[-\pi, \pi]$ .

**Exemplo 4.13** Plote do  $\cos(x)$  e alguns termos de sua série:

O código é :

```
In [13]: """
Descrição : construção do grafico de cos(x)
e alguns termos de sua serie de taylor
=====
programador : Marcelo Almeida
"""

from sympy import symbols, init_printing, cos, series, pi
from sympy.plotting import plot
x = symbols('x', real = True)
fun = cos(x)

# termos da serie de taylor de cos(x) :
fun1 = series(fun,x,x0=0,n=1).removeO() # um termo
fun3 = series(fun,x,x0=0,n=3).removeO() # tres termos
fun5 = series(fun,x,x0=0,n=5).removeO() # cinco termos

# plote da função e seus termos
plot(fun,fun1,fun3,fun5,xlim=(-pi,pi),ylim=(-1,pi), xlabel='eixo x ', ylabel='eixo y', legend = True)
```

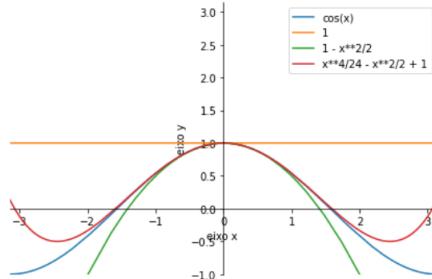


Figura 50: Plote da função  $\cos(x)$  e seus cinco termos em série de Taylor.

### 4.3.3 Série de Fourier

Uma outra série muito importante é a chamada *Série de Fourier*, em homenagem ao matemático e advogado francês *Joseph Fourier*. A série de Fourier é muito usada nos estudos de *equações diferenciais ordinárias e parcias*, assim como na parte de *análise de sinais e sistemas*. A ideia por trás dessas série é a representação de uma função *periódica*<sup>20</sup> em termos de somas de funções trigonométricas, a saber, *senos* e *cossenos*.

A série de Fourier é expressa da seguinte forma :

---

<sup>20</sup>Uma função  $f(t)$  é dita periódica se :

$$f(t + T) = f(t)$$

Para o menor  $T$  inteiro positivo.

$$S(t) = \frac{a_0}{2} + \sum_{n=1}^{\infty} (a_n \cos(n\omega_0 t) + b_n \sin(n\omega_0 t))$$

Onde também podemos escrever da seguinte forma :

$$S(t) = C_0 + \sum_{n=1}^{\infty} C_n \cos(n\omega_0 t - \theta_n)$$

Onde  $\omega_0 = \frac{2\pi}{T}$ , e os termos  $a_0$ ,  $a_n$  e  $b_n$  são dados por :

$$\begin{aligned} a_0 &= \frac{2}{T} \int_{-\frac{T}{2}}^{\frac{T}{2}} f(t) dt \\ a_n &= \frac{2}{T} \int_{-\frac{T}{2}}^{\frac{T}{2}} f(t) \cos(n\omega_0 t) dt, \quad n = 0, 1, 2, \dots, \\ b_n &= \frac{2}{T} \int_{-\frac{T}{2}}^{\frac{T}{2}} f(t) \sin(n\omega_0 t) dt, \quad n = 0, 1, 2, \dots, \end{aligned}$$

Para representar uma série de Fourier de uma dada função  $f(x)$  em SymPy, usamos a função `fourier_series()`. Onde passamos como argumento a função a ser representada em uma série de fourier. Nesse momento é interessante falar de como construir *funções não definidas* em SymPy, isto é, função que não possuem uma lei de formação conhecida. Para fazer isso em SymPy, usamos o comando `Function()` e passamos uma string que representará a função desconhecida de qualquer variável independente. Vamos usar essa ideia para construir uma série de fourier genérica de uma função qualquer  $f(x)$ . O seguinte exemplo ilustra esse fato :

**Exemplo 4.14** Série de Fourier de uma função qualquer  $f(x)$ :

O código é :

```
In [1]: '''
Descrição : criando a série de fourier de uma função
desconhecida f(x)
=====
Programador : Marcelo Almeida
'''
from sympy import symbols, Function, fourier_series, init_printing
init_printing(use_latex=True)

# criando a variável independente x :
x = symbols('x', real = True)
f = Function('f') # cria uma função indefinida

# montando a serie de fourier :
serie_f = fourier_series(f(x))
serie_f
```

Out[1]:

$$\frac{\sin(x) \int_{-\pi}^{\pi} f(x) \sin(x) dx}{\pi} + \frac{\sin(2x) \int_{-\pi}^{\pi} f(x) \sin(2x) dx}{\pi} + \frac{\cos(x) \int_{-\pi}^{\pi} f(x) \cos(x) dx}{\pi} + \frac{\cos(2x) \int_{-\pi}^{\pi} f(x) \cos(2x) dx}{\pi} + \frac{\int_{-\pi}^{\pi} f(x) dx}{2\pi} + \dots$$

Figura 51: Série de Fourier de  $f(x)$ .

Observe que o SymPy faz a representação da série de fourier de  $f(x)$  de forma mais geral possível, ou seja , abrindo vários termos do somatório da própria série. Podemos facilmente obter os termos  $a_n, b_n$  e o  $a_0$  da seguinte forma :

**Exemplo 4.15** Obtendo os termos da Série de Fourier de uma função qualquer  $f(x)$ :

O código é :

```
In [2]: # obtendo o an
serie_f.an

Out[2]: 
$$\left[ \frac{\cos(x) \int_{-\pi}^{\pi} f(x) \cos(x) dx}{\pi}, \frac{\cos(2x) \int_{-\pi}^{\pi} f(x) \cos(2x) dx}{\pi}, \frac{\cos(3x) \int_{-\pi}^{\pi} f(x) \cos(3x) dx}{\pi}, \frac{\cos(4x) \int_{-\pi}^{\pi} f(x) \cos(4x) dx}{\pi}, \dots \right]$$


In [3]: # obtendo o bn :
serie_f.bn

Out[3]: 
$$\left[ \frac{\sin(x) \int_{-\pi}^{\pi} f(x) \sin(x) dx}{\pi}, \frac{\sin(2x) \int_{-\pi}^{\pi} f(x) \sin(2x) dx}{\pi}, \frac{\sin(3x) \int_{-\pi}^{\pi} f(x) \sin(3x) dx}{\pi}, \frac{\sin(4x) \int_{-\pi}^{\pi} f(x) \sin(4x) dx}{\pi}, \dots \right]$$


In [4]: # obtendo a0
serie_f.a0

Out[4]: 
$$\frac{\int_{-\pi}^{\pi} f(x) dx}{2\pi}$$

```

Figura 52: Termos da série de Fourier de  $f(x)$ .

Agora que sabemos um pouco de como criar uma série de fourier, vamos construir essa série para a função  $f(x) = x$  em  $[-\pi], \pi$  e apresentar os cinco primeiros termos usando o método `truncate(n)`, onde  $n$  é um valor inteiro. O exemplo abaixo ilustra esse processo.

**Exemplo 4.16** Obtendo os cinco primeiros termos da série de fourier de  $f(x) = x$  em  $[-\pi, \pi]$  :

O código é :

```
In [5]: '''
Descrição : obtendo os cinco termos da
serie de fourier de f(x) = x
=====
programador : Marcelo Almeida
'''
from sympy import symbols, fourier_series, init_printing, pi
init_printing(use_latex=True)

x = symbols('x', real = True)
fun = x

f_series = fourier_series(fun,(x,-pi,pi)).truncate(5)
f_series

Out[5]: 
$$2 \sin(x) - \sin(2x) + \frac{2 \sin(3x)}{3} - \frac{\sin(4x)}{2} + \frac{2 \sin(5x)}{5}$$

```

Figura 53: 5 Termos da série de Fourier de  $f(x) = x$ .

E para finalizar, vamos plotar o gráfico dessa função com cada termos de sua série de fourier e analisar como essa poderosa ferramenta é boa para aproximar funções a medida que adicionamos mais e mais termos de sua série.

**Exemplo 4.17** Plote da série de fourier de  $f(x) = x$  em  $[-\pi, \pi]$  e alguns termos :

O código é :

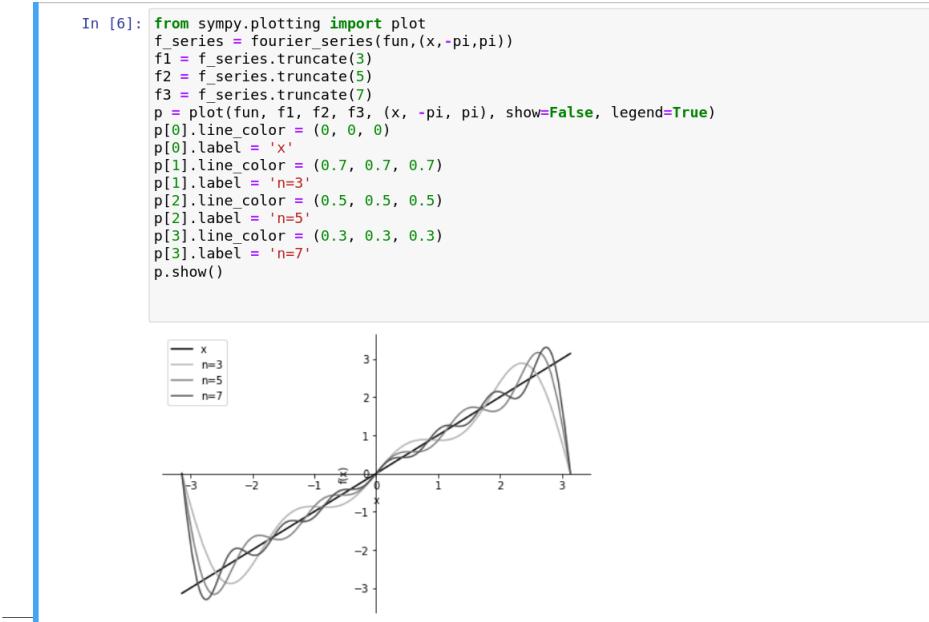


Figura 54: Alguns Termos da série de Fourier de  $f(x) = x$ .

#### 4.3.4 Derivadas de Funções de várias variáveis

Para calcular a derivada de uma função de mais de uma variável, digamos  $z = f(x, y)$ , devemos criar uma função que dependa das duas variáveis simbólicas  $x$  e  $y$ . Com isso, podemos calcular expressões como  $\frac{\partial z}{\partial x}$ ,  $\frac{\partial z}{\partial y}$  e derivadas mistas  $\frac{\partial^2 z}{\partial x \partial y}$ .

A função `diff()` é programada para calcular derivadas parciais também e no caso de derivadas mistas, a ordem que as variáveis são passadas para a função `diff()` é a mesma ordem definida na expressão da derivada. Como exemplo, vamos calcular  $\frac{\partial z}{\partial x}$ ,  $\frac{\partial z}{\partial y}$  e  $\frac{\partial^2 z}{\partial x \partial y}$  da seguinte função  $f(x, y) = e^{xy} \cos(xy)$ .

**Exemplo 4.18** Cálculo das derivadas parciais de  $f(x, y) = e^{xy} \cos(xy)$  :

O código é :

```
In [1]: '''
Descrição : calculo das derivadas parciais de f(x,y) = exp(xy)cos(xy)
=====
programador : Marcelo Almeida
'''

from sympy import symbols, diff, exp, cos, init_printing
init_printing(use_latex= True)
# criando simbólos :
x = symbols('x', real = True) # variavel x
y = symbols('y', real = True) # variavel y
# criando a função z = f(x,y) :
fun_xy = exp(x*y)*cos(x*y)
# derivada em relação a x :
diff_x = diff(fun_xy,x)
# derivada em relação a y :
diff_y = diff(fun_xy, y)
# derivada mista f_yx :
diff_yx = diff(fun_xy,y,x)
```

Figura 55: Código das Derivadas.

O resultado das Derivadas é :

```
In [2]: print("derivada em relação a x : ")
diff_x
derivada em relação a x :
Out[2]: -yexy sin (xy) + yexy cos (xy)

In [3]: print("derivada em relação a y : ")
diff_y
derivada em relação a y :
Out[3]: -xexy sin (xy) + xexy cos (xy)

In [4]: print("derivada em relação a y e x : ")
diff_yx
derivada em relação a y e x :
Out[4]: (-2xy sin (xy) - sin (xy) + cos (xy)) exy
```

Figura 56: Cálculo das Derivadas.

## 4.4 Integrais

### 4.4.1 Integrais de Funções de uma variável

SymPy mostra-se eficiente para calcular integrais de funções de uma ou várias variáveis. Assim como derivadas e limites, podemos calcular integrais de duas formas com a função `integrate()` e a função `Integral()`. A primeira calcula a integral de uma função simbólica automaticamente e a segunda cria uma um símbolo não avaliado da integral da função, que poderá ser avaliada posteriormente com o método `doit()`. É possível calcular integrais indefinidas(*primitivas*), onde é passada a função e a variável independente para uma das duas funções. Para calcular a integral definida, passamos a função a ser integrada e uma *tupla* de três valores : como primeiro elemento passamos a variável independente, no segundo elemento passamos o limite inferior da integral e como terceiro elemento colocamos o limite superior da integral, ou seja, teremos uma tupla formada por  $(x, x_i, x_f)$ .

Como primeiro exemplo vamos calcular a seguinte integral abaixo :

**Exemplo 4.19** *Cálculo da seguinte integral indefinida :*

$$\int e^x \sin(x) dx$$

O código é :

```
In [5]: '''
Descrição : calculo da primitiva de f(x) = exp(x)sin(x)
=====
Programador : Marcelo Almeida
'''
from sympy import symbols, exp, sin, Integral
x = symbols('x', real = True)
fun_x = exp(x)*sin(x)
integral_f = Integral(fun_x,x)
integral_f
```

```
Out[5]:  $\int e^x \sin(x) dx$ 
```

```
In [6]: # calculando a integral com o metodo .doit() :
integral_f.doit()
```

```
Out[6]: 
$$\frac{e^x \sin(x)}{2} - \frac{e^x \cos(x)}{2}$$

```

Figura 57: Cálculo da seguinte integral  $\int e^x \sin(x) dx$  .

Onde na célula In[5] criamos uma integral não avaliada com a função `Integral()` e na célula In[6] avaliamos essa integral com o método `.doit()`. Perceba também que o SymPy não inclui uma constante de integração após avaliar uma integral indefinida. Para contornar esse problema, podemos criar um símbolo que represente a constante de integração e somar esse símbolo à expressão que representa o valor da primitiva.

Agora vejamos como calcular uma integral definida no próximo exemplo.

**Exemplo 4.20** *Cálculo da seguinte integral definida :*

$$\int_{-\pi/2}^{\pi/2} e^x \cos(x) dx$$

O código é :

```
In [1]: '''
Descrição : cálculo da integral definida
=====
Programador : Marcelo Almeida
'''

from sympy import symbols, integrate, exp, cos, pi, init_printing
init_printing(use_latex=True)
x = symbols('x', real=True)
fun = exp(x)*cos(x) # função

# cálculo da integral definida :

valor_integral = integrate(fun, (x, -pi/2, pi/2))

valor_integral
```

Out[1]:  $\frac{1}{2e^{\frac{\pi}{2}}} + \frac{e^{\frac{\pi}{2}}}{2}$

Figura 58: Cálculo da seguinte integral definida  $\int_{-\pi/2}^{\pi/2} e^x \cos(x) dx$ .

Onde vemos claramente a resposta certa. Certamente as funções que são usadas para calcular integrais são muito mais poderosas do que foi mostrado no exemplos acimas. Antes de prosseguir com mais exemplos de integração, quando as funções `integrate()` e `Integral()` não conseguem resolver para um determinado tipo de expressão simbólica, é retornado uma integral não-valiada da expressão. Isso acontece pois existem funções que não possuem primitivas elementares/analíticas, onde nesse caso é recorrente o uso de método numéricos para encontrar um valor aproximado para integral.

#### 4.4.2 Integrais Impróprias

SymPy também pode computar *integrais impróprias*<sup>21</sup>. Para calcular esse tipos de integrais em SymPy, fazemos uso do simbolo  $\infty$  que pode ser importado como `oo`. Como

---

<sup>21</sup>Engana-se quem acha que uma intregral imprópria é aquela onde somente os limites de integração são infinitos. Podemos classificar as integrais imprópria em três tipos basicamente :

- **Tipo I** : Quando um ou os dois limites de integração são infinitos ( $\pm\infty$ ) mas o integrando é contínuo em todo o intervalo de integração
- **Tipo II** : Quando os limites de integração são finitos mas o integrando possui algum tipo de descontinuidade nesse intervalo de integração.
- **Tipo III** : É uma mistura dos tipos I e II .

exemplo, vamos calcular simbolicamente a seguinte integral imprópria.

$$\int_0^\infty \frac{\sin^3(x)}{x^3} dx$$

**Exemplo 4.21** *Cálculo da integral imprópria :*

*O código é :*

```
In [2]: '''
Descrição : calculando uma integral imprópria
=====
programador : Marcelo Almeida
'''
from sympy import symbols, integrate, oo, sin, Integral, exp
x = symbols('x', real = True)
fun = sin(x)**3 / x**3
fun = Integral(fun, (x, 0, oo))
fun

Out[2]: ∫ sin³(x) dx
         0           x³

In [3]: # calculando o seu valor com o método .doit()
fun.doit()

Out[3]: 3π
        —
          8
```

Figura 59: Cálculo da integral imprópria.

Como pode-se observar o SymPy resolve integrais impróprias de forma rápida e analítica. Para ficarmos mais familiarizados na resolução de tipos de integrais, vejamos mais um exemplo interessante de integração imprópria.

**Exemplo 4.22** *Calcule a seguinte integral imprópria em SymPy :*

$$\int_0^\infty e^{-x^2} dx$$

*O código é :*

```
In [3]: '''
Descrição : cálculo de uma integral imprópria
=====
Programador : Marcelo Almeida
'''

from sympy import symbols, integrate, oo, sin, Integral, exp
x = symbols('x', real = True)
fun = exp(-x**2)
expr = Integral(fun, (x, 0, oo))
expr

Out[3]: 
$$\int_0^{\infty} e^{-x^2} dx$$


In [4]: # avaliando a integral
expr.doit()

Out[4]: 
$$\frac{\sqrt{\pi}}{2}$$

```

Figura 60: Cálculo da integral imprópria.

#### 4.4.3 Funções Especiais

Dentro da matemática aplicada existem certos tipos de funções que são definidas por um tipo especial de integral, podendo ou não depender de um ou mais parâmetros dentro da integral. Essas funções recebem o nome de funções especiais<sup>22</sup>.

Como essas funções possuem larga aplicações em diferentes campos da matemática e suas formulações dependem de integrais impróprias ou não, conhecer e aprender a calcular essas funções em SymPy será de grande valia para o aluno.

Dentre as *funções especiais* mais conhecidas temos : **função gama**, **função beta**, **A Função Erro de Gauss** e **A Função Exponencial Integral** .

- **Função Gama :**

Um tipo especial de função que surge muito na resolução de *problemas de contorno* é a função gama, que é definida por :

$$\Gamma(x) = \int_0^{\infty} t^{x-1} e^{-t} dt$$

Cuja a sua *convergência* requer que  $x > 0$ . A relação de recorrência :

$$\Gamma(x + 1) = x\Gamma(x)$$

Que pode ser obtida usando integração por partes.

---

<sup>22</sup>Para um aprofundamento das funções especiais, consulte a referências [9]

Com os conceitos aprendidos sobre integração nas seções anteriores, vamos criar a função gama em SymPy e calcula-lá em  $x = 2$ . O exemplo a seguir ilustra esse processo :

**Exemplo 4.23** Criando e calculando a função  $\Gamma(x)$  em  $x = 2$  com SymPy :

O código é :

```
In [5]: '''
Descrição : criando a função gamma em sympy e calculando em x = 2
=====
programador : Marcelo Almeida
'''
from sympy import symbols, exp, init_printing, Integral, oo
x = symbols('x', real = True, positive = True) # definindo x como valor positivo
t = symbols('t', real = True )
fun_gamma = t**(x-1)*exp(-t)
expr = Integral(fun_gamma,(t,0,oo))
# mostrando a integral não avaliada
expr
Out[5]: 
$$\int_0^{\infty} t^{x-1} e^{-t} dt$$


In [6]: # valiando essa integral, temos :
expr.doit()
Out[6]: 
$$\Gamma(x)$$


In [7]: # calculando a função gamma em x = 2 :
expr.subs({x : 2}).doit()
Out[7]: 1
```

Figura 61: Função  $\Gamma(x)$ .

- **Função Beta :**

Uma outra função especial bem importante que surge em vários problemas da modelagem matemática é a função beta, que foi estudada pela primeira vez pelo matemáticos Euler e Legendre. Sua definição é :

$$B(m, n) = \int_0^1 x^{m-1} (1-x)^{n-1} dx$$

e tem convergência para  $m > 0, n > 0$ . A função beta está associada a função gama por :

$$B(m, n) = \frac{\Gamma(m)\Gamma(n)}{\Gamma(m+n)}$$

Agora vamos criar um programa que implemente a função beta e avalie a mesma em  $m = 2$  e  $n = 1$ .

**Exemplo 4.24** Criando e calculando a função  $B(m, n)$  em  $m = 2$  e  $n = 1$  com SymPy :

O código é :

```
In [8]: '''
Descrição : criando a função beta em sympy e avaliando em m=2 e n = 1
=====
programador : Marcelo Almeida
'''

from sympy import symbols, init_printing, Integral
init_printing(use_latex=True)
x = symbols('x', real = True)
m, n = symbols('m,n', real = True , positive = True)
fun_beta = x**(m - 1)*(1 - x)**(n - 1)
expr = Integral(fun_beta, (x,0,1))
expr
```

Out[8]:  $\int_0^1 x^{m-1}(1-x)^{n-1} dx$

```
In [9]: # avaliando a integral :
expr.doit()
```

Out[9]: 
$$\frac{\Gamma(m) {}_2F_1\left(\begin{matrix} m, 1-n \\ m+1 \end{matrix} \middle| 1\right)}{\Gamma(m+1)}$$

```
In [10]: # calculando a função beta para m = 2 e n = 1 .
expr.subs({m : 2 , n : 1}).doit()
```

Out[10]:  $\frac{1}{2}$

Figura 62: Função  $B(m, n)$ .

- **A Função Erro de Gauss :**

A função erro de gauss, desempenha um papel relevante em muitas aplicações envolvendo *Transformadas integrais*, como a *Transformada de Fourier* e a *Transformada de Laplace*<sup>23</sup>. A função erro, denotada por :

$$erf(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-u^2} du$$

E a função erro complementar se define em termos da função erro por :

$$erfc(x) = 1 - erf(x)$$

---

<sup>23</sup>Veja [11] e [10].

De posse desse conhecimento, vamos implementar a função erro de Gauss e vamos avaliar em  $x = 0$ . O próximo exemplo mostra a implementação desse código.

**Exemplo 4.25** Criando e calculando a função erro  $\text{erf}(x)$  em  $x = 0$  com SymPy :

O código é :

```
In [11]: from sympy import symbols, sqrt, Integral, exp, init_printing, \
pi
init_printing(use_latex=True)
x = symbols('x', real = True)
u = symbols('u', real = True)
fun_erro = (2/sqrt(pi))*Integral(exp(-u**2),(u,0,x))

fun_erro
Out[11]: 
$$\frac{2 \int_0^x e^{-u^2} du}{\sqrt{\pi}}$$


In [12]: # avaliando essa integral
fun_erro.doit()

Out[12]: erf(x)

In [13]: # calculando a função erro em x = 0
fun_erro.subs({x:0}).doit()

Out[13]: 0
```

Figura 63: Função  $\text{erf}(x)$ .

- **A Função Exponencial Integral :**

A função exponencial integral é definida<sup>24</sup> por

$$Ei(x) = \int_{-x}^{\infty} \frac{e^{-t}}{t} dt$$

Como  $\frac{1}{t}$  diverge para  $t = 0$ , então essa integral deve ser interpretada no sentido do *valor principal de Cauchy*<sup>25</sup>. Vamos criar essa função em SymPy e avaliar também.

**Exemplo 4.26** Criando e calculando a função exponencial integral com SymPy :

O código é :

---

<sup>24</sup>Uma das várias formas alternativas como essa função é definida .

<sup>25</sup>Consulte [12], [13] e [14] .

```
In [1]: '''
Descrição : Vamos criar a função exponencial integral no SymPy
=====
programador : Marcelo Almeida
'''

from sympy import symbols, exp, Integral, oo
x = symbols('x', real = True)
t = symbols('t', real = True)

Ei = Integral(exp(-t)/t, (t, -x, oo))
Ei

Out[1]: 
$$\int_{-x}^{\infty} \frac{e^{-t}}{t} dt$$


In [2]: # avaliando essa integral :

Ei.doit()

Out[2]: 
$$-Ei(-xe^{i\pi})$$

```

Figura 64: Função  $Ei(x)$ .

#### 4.4.4 Integrais Duplas e Triplos

Uma parte importante do cálculo de varias variáveis é calcular as integrais dessas funções de mais de uma variável. Para realizar o cálculo de integrais duplas e triplas em SymPy, utilizamos ainda da função `integrate()` ou `Integral()`, mas só que agora passamos a função de 2 ou 3 variáveis e a ordem da variáveis de integração. Por exemplo, para calcular  $\iint f(x, y) dx dy$ , passamos a função  $f(x, y)$  e logo após passamos a variável  $x$  e  $y$ . Como exemplo de integração dupla, vamos calcular a seguinte integral dupla da função  $f(x, y) = x^2 + y^2$  e plotar seu gráfico juntamente com a sua primitiva.

**Exemplo 4.27** Calculando a seguinte integral dupla  $\iint x^2 + y^2 dx dy$  e plotando seu gráfico

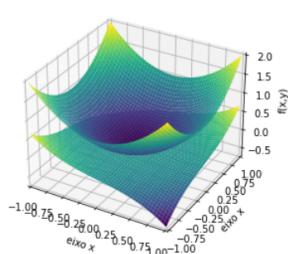
O código é :

```
In [7]: '''
Descrição : calculando uma integral dupla
=====
programador : Marcelo Almeida
'''

from sympy import symbols, integrate, init_printing
from sympy.plotting import plot3d
x, y = symbols('x,y', real = True)
fun_xy = x**2 + y**2
# primitiva da função :
primitiva = integrate(fun_xy,x,y)
primitiva

Out[7]: 
$$\frac{x^3 y}{3} + \frac{x y^3}{3}$$


In [8]: # plote das duas funções :
plot3d(fun_xy,primitiva,(x,-1,1),(y,-1,1), title='plote da função  $x^2 + y^2$  e sua primitiva',
xlabel='eixo x', ylabel='eixo x', zlabel='f(x,y)')
```

Figura 65: Plote da função  $f(x, y) = x^2 + y^2$  e sua primitiva.

Para calcular uma integral tripla segue a mesma ideia da integração dupla. Então vamos calcular a seguinte integral tripla em SymPy :

$$\iiint (x^2 + y^2 + z^2) dx dy dz$$

**Exemplo 4.28** Calculando a seguinte integral tripla  $\iiint (x^2 + y^2 + z^2) dx dy dz$  :

O código é :

```
In [15]: """
Descrição : cálculo de uma integral tripla
=====
Programador : Marcelo Almeida
"""

from sympy import symbols, integrate, exp, Integral
# criando os símbolos :
x, y, z = symbols('x,y,z', real = True)
# criando a função :
fun_xyz = x**2 + y**2 + z**2
# calculando a sua primitiva :
primitiva = Integral(fun_xyz,x,y,z)
primitiva

Out[15]: \iiint (x2 + y2 + z2) dx dy dz
```

```
In [16]: # avaliando a integral :
primitiva.doit()

Out[16]: xyz3/3 + z (x3y/3 + xy3/3)
```

Figura 66: Cálculo de  $\iiint (x^2 + y^2 + z^2) dx dy dz$ .

## 5 Resolvendo Equações em SymPy

### 5.1 Equação Linear

Chegamos em um tópico muito rotineiro dentro da matemática elementar, que é a resolução de equações lineares e não-lineares. Equações lineares são mais simples de serem resolvidas que as equações não-lineares<sup>26</sup>, veremos que resolver esses tipos de problema usando o pacote SymPy, que funciona de forma satisfatória na maioria dos casos. Começaremos essa seção com exemplos simples e que o leitor já está acostumado no seu curso de matemática elementar, trata-se de resolver a seguinte equação do segundo grau  $x^2 - 1 = 0$ . Ou seja, queremos encontrar os dois valores de  $x$ <sup>27</sup>, chamados de *raízes* que satisfazem a equação dada, em outra palavra, queremos encontrar valores numéricos de  $x$  que torne a equação igual à zero. Por se tratar de uma equação do segundo graus

<sup>26</sup>Quando se trata de resolução de equações não-lineares ou os sistemas de equações não-lineares, são poucas as equações que possuem solução fechadas. Na maioria dos casos temos que recorrer aos métodos numéricos para encontrar uma solução aproximada.

<sup>27</sup>O teorema fundamental da álgebra nos diz que uma equação de grau  $n$  terá no máximo  $n$  raízes.

incompleta, sua solução é extremamente simples e manipulando a equação chegamos ao resultado que os valores de  $x$  são :  $x_1 = -1$  e  $x_2 = 1$ .

Agora que sabemos a solução da equação, vamos verificar como fazer isso usando o SymPy e comparar as respostas, que deve ser idêntica a resposta que obtivemos por meios analíticos. A função que resolve equações e sistemas de equações em SymPy é a função `solve()`, ela não é a única, porque dependendo da natureza de sua equação (*polinomial, trigonométrica*) existem funções específicas para resolver a dada equação, mas a função `solve()` é mais geral de todas<sup>28</sup>. Para usar esse solucionador devemos passar no mínimo dois parâmetros para essa função, como primeiro parâmetro passamos a equação a ser resolvida, usando a função `Eq()` que aprendemos em seções anteriores e o segundo parâmetro passamos a variável no qual queremos resolver a equação. Dito isso, esse problema em SymPy fica da seguinte forma :

**Exemplo 5.1** Resolvendo a equação  $x^2 - 1 = 0$  com SymPy

O código é :

```
In [1]: """
Descrição : vamos resolver a equação x^2 - 1 = 0
usando o SymPy
=====
programador : Marcelo Almeida
"""

from sympy import symbols, Eq, solve
# criando a variável independente :
x = symbols('x', real = True)
# criando a equação a ser resolvida :
equacao = Eq(x**2 - 1, 0)
# resolvendo a equação para x :
sol = solve(equacao,x)
sol

Out[1]: [-1, 1]

In [2]: # retornando a solução como um dicionário :
sol2 = solve(equacao,x, dict = True)
sol2

Out[2]: [{x: -1}, {x: 1}]
```

Figura 67: Solução da equação  $x^2 - 1 = 0$ .

Como podemos observar a solução que o SymPy forneceu a resposta correta. Observe que na linha de código `In[1]` criamos a equação e resolvemos para  $x$ , e a variável `sol` recebe a solução em forma de uma *lista* do Python em ordem crescente de valores das raízes. Já na linha de código `In[2]`, usamos o parâmetro opcional da função `solve()` para retornar a solução como uma lista de *Dicionários*, dessa forma fica mais fácil visualizar a solução, principalmente quando falarmos de sistemas de equações. Começamos com um exemplo bem trivial usando a função `solve()`, podemos obter a solução geral da equação  $ax^2 + bx + c = 0$ , que é conhecida do leitor. Isso pode ser feito com o seguinte código :

---

<sup>28</sup>Isso possui uma pequena desvantagem no tempo de execução, que dependendo da equação a ser resolvida, pode demorar mais tempo para resolvê-la, do que usando um solucionador específico.

**Exemplo 5.2** Resolvendo uma equação do segundo grau  $ax^2 + bx + c = 0$  com SymPy

O código é :

```
In [3]: '''
Descrição : obtendo a solução geral da equação ax^2 + bx + c = 0 ,
em termos dos coeficientes a , b e c
=====
programador : Marcelo Almeida
'''

from sympy import symbols, Eq, solve, init_printing
init_printing(use_latex= True)

x = symbols('x', real = True) # variável independente
a,b,c = symbols('a,b,c', real = True ) # coeficientes da equação

# criando a equação do 2 grau :
equacao = Eq(a*x**2 + b*x + c , 0)

# resolvendo :
sol = solve(equacao,x, dict = True)
sol
```

Out[3]:  $\left[ \left\{ x : \frac{-b - \sqrt{-4ac + b^2}}{2a} \right\}, \left\{ x : \frac{-b + \sqrt{-4ac + b^2}}{2a} \right\} \right]$

Figura 68: Solução da equação  $ax^2 + bx + c = 0$ .

Podemos ir mais além e verificar qual a solução da equação  $\sin(x) - 1 = 0$ , isso é feito de forma simples no SymPy

**Exemplo 5.3** Resolvendo a equação  $\sin(x) - 1 = 0$  com SymPy

O código é :

```
In [4]: '''
Descrição : Vamos resolver a equação sin(x) -1 = 0
=====
programador : Marcelo Almeida
'''

from sympy import symbols, Eq, solve, init_printing, sin

init_printing(use_latex= True)
x = symbols('x', real = True) # variável independente

# criando a equação :
equacao = Eq(sin(x) -1 , 0)

sol = solve(equacao,x,dict = True)
sol
```

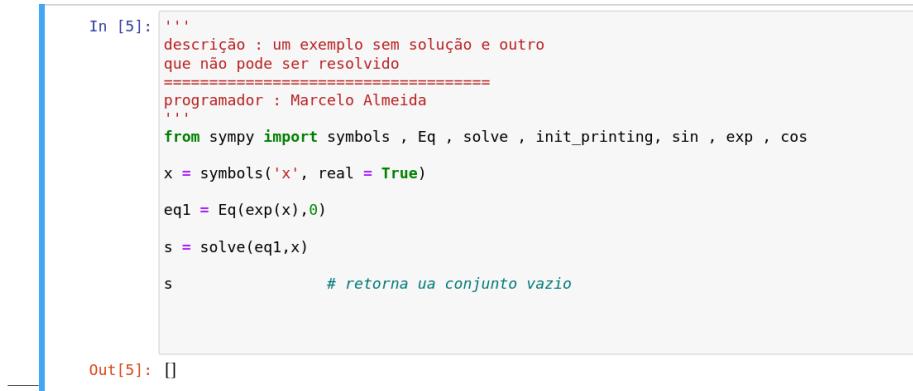
Out[4]:  $\left[ \left\{ x : \frac{\pi}{2} \right\} \right]$

Figura 69: Solução da equação  $\sin(x) - 1 = 0$ .

Certamente pode ocorrer de não existir uma solução da equação ou não pode ser possível encontrar uma solução por meios analíticos, nesses casos, o SymPy lida de forma elegante, vejamos um exemplo.

**Exemplo 5.4** Um sistema sem solução.

O código é :



```
In [5]: """
Descrição : um exemplo sem solução e outro
que não pode ser resolvido
=====
programador : Marcelo Almeida
"""

from sympy import symbols, Eq, solve, init_printing, sin, exp, cos
x = symbols('x', real = True)
eq1 = Eq(exp(x), 0)
s = solve(eq1,x)
s # retorna ua conjunto vazio

Out[5]: []
```

Figura 70: Sistema sem Solução.

## 5.2 Sistema de Equações Lineares

A função `solve()` também pode ser usada para resolver um *sistema de equações lineares*, onde passamos cada equação do sistema como uma *lista* de Python e o como seu segundo argumento uma outra *lista* contendo as variáveis da equação. Um sistema de equações lineares  $m \times n$  é representado por :

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \dots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \dots + a_{2n}x_n &= b_2 \\ \dots &= \dots \\ a_{m1}x_1 + a_{m2}x_2 + a_{m3}x_3 + \dots + a_{mn}x_n &= b_n \end{aligned}$$

Para resolver um sistema de equações lineares como o mostrado acima, usamos a função `solve()` mas com uma sintaxe um pouco diferente de como vimos até o momento. Como primeiro argumento da função, passamos uma *lista* contendo todas as equações do sistemas e como segundo argumento, passamos outra *lista* mas com as variáveis do sistema. Para ficar mais claro, vamos implementar e resolver o seguinte sistema de equações lineares abaixo.

**Exemplo 5.5** Resolvendo o seguinte sistema de equações :

$$\begin{cases} x + y + z = 6 \\ x + 2y + 2z = 9 \\ 2x + y + 3z = 11 \end{cases}$$

O código é :

Claro que o leitor pode facilmente verificar que se trata da solução sistema, substituindo os valores  $x, y$  e  $z$  nas 3 equações para reduzi-las nos valores do lado direito da equação. Claro que o SymPy não se limita a exemplos simples como vimos até agora, para não delongar muito e tornar essa seção extensa, encorajo o leitor a explorar mais exemplos de resolução de sistema de equações usando o SymPy.<sup>29</sup>

---

<sup>29</sup>É importante falar que a função `solve()` também pode ser usada para resolver sistemas de equações não-lineares.

```
In [6]: '''
Descrição : Resolvendo um sistema de equações
=====
Programador : Marcelo Almeida
'''

from sympy import symbols, solve, init_printing, Eq
init_printing(use_latex=True)
x,y,z = symbols('x,y,z', real = True)
# criando as equações :
eq1, eq2, eq3 = Eq(x+y+z,6) , Eq(x+2*y +2*z,9) , Eq(2*x +y +3*z ,11)
# resolvendo o sistema para x,y e z :
sol_eq = solve([eq1,eq2,eq3],[x,y,z], dict=True)
sol_eq
Out[6]: [{x : 3, y : 2, z : 1}]

In [7]: # mostrando os valores das soluções :
for chave,valor in sol_eq[0].items():
    print(f'{chave} = {valor} ')

```

x = 3  
y = 2  
z = 1

Figura 71: Sistema de Equações.

### 5.3 Resolvendo Equações Diferenciais

Nesse tópico veremos como usar o SymPy para resolver algumas equações diferenciais simples, como o estudo das equações diferenciais é um mundo a parte dentro da matemática, vamos nos limitar apenas nas suas soluções usando o SymPy, deixando todo o formalismo matemático em segundo plano<sup>30</sup>. Vale lembrar que em cursos introdutórios sobre equações diferenciais, o leitor estudou a seguinte equação diferencial :

$$a_n(x) \frac{d^n y}{dx^n} + a_{n-1}(x) \frac{d^{n-1}}{dx^{n-1}} + \dots + a_2(x) \frac{d^2 y}{dx^2} + a_1(x) \frac{dy}{dx} + a_0 y(x) = g(x)$$

É uma equação diferencial ordinária de *n*-ésima ordem com *coeficientes não-constantes* ( $a_n, a_{n-1}, \dots, a_0$ ), *linear* e *não-homogênea*<sup>31</sup>. Uma solução da equação diferencial é uma função  $y(x) = \phi(x)$  definida em algum *intervalo I*, que satisfaz a equação, ou seja, reduz a mesma em uma identidade. Começaremos pela equação linear mais simples, que é dada por :

$$a_1(x) \frac{dy}{dx} + a_0 y(x) = g(x)$$

Que é uma *equação diferencial ordinária de 1 ordem linear*. Comumente resolvemos uma equação diferencial linear para a sua maior derivada na equação, ou seja, deixamos

<sup>30</sup>Existem dois tipos de equações diferenciais : *Ordinárias* e *Parciais*.

- *Equações Diferenciais Ordinárias* : são aquelas onde temos funções de uma única variável e suas derivadas ordinárias
- *Equações Diferenciais Parciais* : são aquelas onde temos uma ou mais funções de duas ou mais variáveis juntamente com as suas derivadas parciais .

<sup>31</sup>Em algumas literaturas esse padrão é chamado de *forma canônica*

o termos  $\frac{dy}{dx}$  'sozinho', para isso vamos multiplicar a equação por  $\frac{1}{a_1(x)}$ , obtemos :

$$\frac{dy}{dx} + q(x)y(x) = r(x)$$

Onde :  $q(x) = \frac{a_0(x)}{a_1(x)}$  e  $r(x) = \frac{g(x)}{a_1(x)}$ .

Para resolver equações diferenciais em SymPy, devemos usar a função `dsolve()`, onde passamos essencialmente 3 parâmetros para função, são eles :

- Passamos como primeiro argumento a equação diferencial propriamente dita.
- Passamos como segundo argumento a função desconhecida para qual queremos resolver a EDO.
- Como terceiro argumento(opcional) passamos as condições iniciais ou de contorno se houver através do argumento nomeado `ics=`, que é passado como um *dicionário*.

Para ficar mais claro como devemos proceder para resolver uma EDO usando essa função, vejamos um exemplo elementar para resolver uma equação diferencial ordinária bem simples :  $\frac{dy}{dx} + y = \sin(x)$ .

Do curso de equações diferenciais sabemos que a solução, portanto  $y(x)$ , que procuramos é  $\frac{\sin(x) - \cos(x)}{2} + Ce^{-x}$ .

Agora vejamos como podemos implementar isso no Sympy e encontrar a mesma solução.

**Exemplo 5.6** : Resolvendo a seguinte EDO :

$$\frac{dy}{dx} + y = \sin(x)$$

O código é :

Então verificamos que a solução dada pelo SymPy encontra-se correta, caso ainda tenha dúvidas sobre essa solução, substitua na equação e verifique que a mesma se reduz à uma identidade. Bom, já sabemos como encontrar uma solução geral de uma EDO (nesse caso, uma solução à uma parâmetro), mas em muitos problemas aplicados temos uma condição do sistema, que é chamada de *condição inicial*. Para resolver um *problema de valor inicial* dada por :

$$\frac{dy}{dx} + q(x)y(x) = r(x)$$

```
In [1]: '''
Descrição : Resolvendo uma equação diferencial com SymPy
=====
programador : Marcelo Almeida
'''
from sympy import symbols, init_printing, sin, exp, Function, \
Eq, dsolve, diff

# criando a variável independente e a função desconhecida
x = symbols('x', real = True )
y_x = Function('y')(x)

# criando a equação diferencial :
eq_edo = Eq(diff(y_x,x) + y_x , sin(x))

# mostrando a EDO :
eq_edo

Out[1]:  $y(x) + \frac{d}{dx}y(x) = \sin(x)$ 

In [2]: # solucionando aedo :
sol_edo = dsolve(eq_edo, y_x)

Out[2]:  $y(x) = C_1e^{-x} + \frac{\sin(x)}{2} - \frac{\cos(x)}{2}$ 
```

Figura 72: Solução da EDO .

Sujeita à  $y(x_0) = y_0$ , devemos passar essa condição ao parâmetro `ics` da função `dsolve()`. Vamos resolver o mesmo problema do exemplo anterior, mas agora impondo a condição  $y(0) = \frac{\pi}{2}$  e vamos seu gráfico. Isso em SymPy fica :

**Exemplo 5.7** : Resolvendo a seguinte EDO :

$$\frac{dy}{dx} + y = \sin(x) \text{ com } y(0) = \frac{\pi}{2}$$

O código é :

```
In [7]: '''
Descrição : Resolvendo uma equação diferencial com um
problema de valor inicial com SymPy
=====
programador : Marcelo Almeida
'''
from sympy import symbols, init_printing, sin, exp, Function, \
Eq, dsolve, diff, pi
from sympy.plotting import plot
# criando a variável independente e a função desconhecida
x = symbols('x', real = True )
y_x = Function('y')(x)
# criando a equação diferencial :
eq_edo = Eq(diff(y_x,x) + y_x , sin(x))
# solucionando o problema de valor inicial
sol_ed = dsolve(eq_edo,y_x, ics={y_x.subs({x:0}) : pi/2})

# mostrando a solução
sol_ed

Out[7]:  $y(x) = \frac{\sin(x)}{2} - \frac{\cos(x)}{2} + \left(\frac{1}{2} + \frac{\pi}{2}\right)e^{-x}$ 
```

```
In [8]: # plotando seu grafico em [0,pi/2]
plot(sol_ed.rhs,(x,0,pi/2),x_label='eixo x',y_label='eixo y',title='pote da solução da EDO')
```

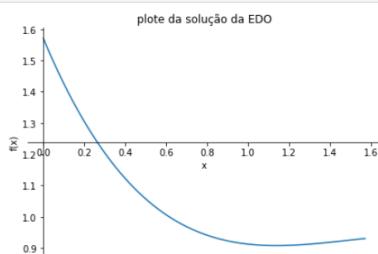


Figura 73: Solução da EDO com problema de valor inicial .

Só uma observação na parte da plotagem da solução da EDO, para selecionarmos apenas o lado direito da equação, para fins de avaliação e de plote como nesse caso, devemos usar o *atributo .rhs* na solução obtida (*sol\_ed*), dessa forma o SymPy entende que queremos apenas os termos do lado direito da equação e assim podemos fazer o plote da solução.

Podemos resolver EDOS de ordem superiores com a função `dsolve()`, cuja a ideia segue a mesma para EDOS de 1 ordem como visto nos exemplos anteriores. Vamos resolver mais um exemplo de EDO, mas agora vejamos como resolver um *problema de valor de contorno de 2 ordem*. Seja a equação diferencial ordinária elíptica de 2 ordem não homogênea :

$$-\frac{d^2u}{dx^2} + u = -6x^2 + 20x + 12 - 3 \sin\left(\frac{3\pi x}{2}\right) - \frac{27\pi^2}{4} \sin\left(\frac{3\pi x}{2}\right), \quad 0 < x < 1$$

Com as condições de contorno do tipo *Dirichlet* homogêneas :  $u(0) = u(1) = 1$ . Para solucionarmos essa EDO fazemos o seguinte código no SymPy :

**Exemplo 5.8** : Resolvendo a seguinte EDO de 2 ordem não-homogênea com condições de contorno :

O código é :

```
In [10]: """
Descrição : solução de uma EDO de 2 ordem
não-homogênea
-----
Programador : Marcelo Almeida
"""
from sympy import symbols, init_printing, sin, exp, Function, \
Eq, dsolve, diff, pi
from sympy.plotting import plot
init_printing(use_latex=True)
# criando a variável independente, função e a função do lado direito
x = symbols('x', real=True)
u_x = Function('u')(x)
h_x = -6*x**2 + 20*x + 13 - 3*sin((3*pi*x)/2) - ((27*pi**2)/(4))*sin((3*pi*x)/2)
# criando a EDO :
edo_eq = Eq(-diff(u_x,x,2) + u_x,h_x)

# solucionando o problema de fronteira :
sol_edo = dsolve(edo_eq,u_x,ics={u_x.subs({x:0}):0, u_x.subs({x:1}):0 })
sol_edo = sol_edo.simplify() # simplificando a solução
sol_edo
```

```
Out[10]: 
$$u(x) = \frac{\left(1-e^2\right) \left(-6 x^2+20 x-3 \sin \left(\frac{3 \pi x}{2}\right)+1\right) e^x-\left(1-18 e\right) e^{2 x}+e \left(-18+e\right) e^{-x}}{1-e^2}$$

```

```
In [12]: # plote dessa função em [0,1]
plot(sol_edo.rhs,(x,0,1),x_label='x',y_label='y',title='plote da solução da EDO')
```

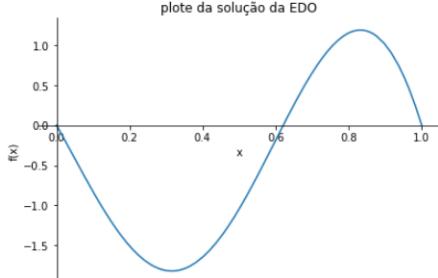


Figura 74: Solução da EDO com condições de contorno .

## References

- [1] SWEIGART, al. **Automatize Tarefas Maçantes com Python:Programação Prática Para Verdadeiros Iniciantes.** Editora Novatec, 2015.
- [2] MATTHES, Eric. **Curso Intensivo de Python: Uma Introdução Prática e Baseada em Projetos à Programação .** Editora Novatec, 2016.
- [3] MCKINNEY, Wes. **Python para Análise de Dados.** Editora Novatec, 2018.
- [4] RAMALHO, Luciano. **Python Fluente .** Editora Novatec, 2015.
- [5] SAHA, Amit. **Doing Math with Python: Use Programming to Explore Algebra, Statistics, Calculus, and More!.** No Starch Press, 2015.
- [6] STEWART, James. **Cálculo - Volume 2.** Editora Cengage, 2013
- [7] ANTON; BIVENS; DAVIS. **Cálculo - Volume 2.** Editora Bookman, 2014.
- [8] MUNEM; FOULIS. **Cálculo - Volume 2.** Editora LTC , 1982.
- [9] BASSALO ; CATTANI. **Elementos de Física Matemática , Vol. 1 .** Editora Livraria da Física , 2010.
- [10] BUTKOV. **Física Matemática .** Editora LTC , 1988.
- [11] ZILL ; CULLEN. **Equações Diferenciais , Vol. 1 .** Editora Pearson , 2001.
- [12] ESTRADA ; RICARDO ; KANWAL; RAM. **Singular Integral Equation.** Editora Birkhauser , 2000.
- [13] RAM P. KANWAL .**Linear Integral Equations.** Editora Boston: Birkhäuser, 1996.
- [14] HSU, Hwein .**Outline of Fourier Analysis.** Nova Iorque: Associated Educational Services Corp, 1967.