Criação e Formatação de Gráficos com o python

Tiago de Paula

Python apareceu pela primeira vez na década de 90 como uma tentativa de uma linguagem de programação com uma sintaxe simplificada e que favorecia a legibilidade do código, enforçando construções explícitas e menos complicadas. Na época, a linguagem cresceu apenas em um nicho próprio, mas começou a deslanchar em popularidade na década seguinte com a segunda versão da linguagem. Atualmente, Python é uma das linguagens de programação mais utilizadas, já na sua terceira iteração, e é mantido como código aberto por uma comunidade própria de desenvolvedores.

Uma das áreas em que Python mais tem destaque na atualiadade é com ánalise de dados, sendo comparável a linguagens mais tradicionais da área, como R. No entanto, boa parte do crescimento na área se deve às bibliotecas e às ferramentas criadas pela comunidade, em especial ao ecossistema SciPy. Isso envolve os pacotes próprios do SciPy e projetos como NumPy, Matplotlib e Pandas, que serão utilizadas neste material. Além dessas bibliotecas, pode ser interessante utilizar a biblioteca Seaborn e a ferramenta de *notebooks* do Jupyter.

O número de ferramentas e maneiras diferentes para se fazer qualquer coisa em Python é absurdamente grande, no entanto, nesse material serão exploradas apenas as funcionalidades importantes no curso de Física Experimental 3 (F 329). A divisão das seções é feita para começar com as técnicas mais básicas (**??** e **??**), seguida das intermediárias (**??** e **??**) e, por fim, as mais específicas (**??**, **??**, **??** e **??**).

0 Configurações Básicas

0.1 Bibliotecas de Python

Para os gráficos desse material, as bibliotecas Pandas e Matplotlib serão necessárias em todos os exemplos, enquanto a NumPy e alguns pacotes do SciPy aperecerão em boa parte deles. Essas bibliotecas serão usadas aqui com nomes reduzidos, como no código **??**.

Código 1: Importando as bibliotecas principais

```
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt

# -- configurações e o resto do código -- #
```

0.2 Opções de Formatação

Todos os gráficos serão feitos com a interface pyplot do Matplotlib , o que ajuda muito na confecção das figuras. Essa biblioteca de gráficos vem com uma imensa variedade de estilos que podem ser combinados na hora de criar os seus gráficos.

Em todos os exemplos deste material serão usados os estilos `'seaborn-v0_8-whitegrid'`, `'seaborn-v0_8-paper'` e `'seaborn-v0_8-muted'`, nessa ordem, além de algumas configurações adicionais. Tudo está presente no código **??**.

Código 2: Exemplo de customização de formatação

```
# -- bibliotecas -- #

config = {
    'axes.spines.right': False,
    'axes.spines.top':   False,
    'axes.edgecolor':  '.4',
    'axes.labelcolor': '.0',
    'axes.titlesize': 'large',
    'axes.labelsize': 'medium',
    'figure.autolayout': True,
    'figure.figsize': (4.5, 3.5),
    'font.family': ['serif'],
    'font.size': 10.0,
    'grid.linestyle': '--',
    'legend.facecolor': '.9',
    'legend.frameon': True,
    'savefig.transparent': True,
    'text.color': '.0',
    'xtick.labelsize': 'small',
    'ytick.labelsize': 'small',
}
plt.style.use(['seaborn-v0_8-whitegrid', 'seaborn-v0_8-paper', 'seaborn-v0_8-muted', config])

# -- resto do código -- #
```

0.3 Importando Dados

Em `Python` , a melhor forma de se representar os dados para análise gráfica costuma ser com as ferramentas do `Pandas` . Nessa biblioteca os dados são tratados na forma de um `DataFrame` , um tipo de representação em tabelas com as abstrações tradicionais de linhas e colunas. Essa tabela pode ser construída manualmente ou gerada de um arquivo de texto ou binário seguindo algumas especificações. As colunas do `DataFrame` são acessadas por nome, similar ao acesso por chave em um dicionáro.

No caso dos dados de um gerenciador de tabelas, como o `Excel`, o `Google Planilhas` ou o `LibreOffice Calc`, é preciso organizar os valores em colunas, como na figura **??**, em uma folha de planilha própria com apenas estes dados. É recomendável, também, colocar estes dados no começo da folha, começando na primeira linha e na primeira coluna, e deixar a primeira linha dos dados com o nome da coluna que será usado para acesso no `DataFrame` .

fx

	A	B	C	D	E
1	V	dV	A	dA	
2	1.916	0.004	19.12	0.07	
3	3.826	0.008	38.2	0.1	
4	5.69	0.01	56.8	0.2	
5	7.60	0.02	76.0	0.5	
6	9.41	0.03	94.0	0.6	
7	10.39	0.03	103.7	0.7	
8	10.88	0.03	108.5	0.7	
9	11.29	0.03	112.5	0.7	
10	11.82	0.03	117.8	0.7	
11	11.98	0.03	119.3	0.7	
12					
13					
14					

Figura 1: Organização dos dados para importar em `Python`

0.3.1 Arquivos CSV

Arquivos `CSV` (*Comma Separated Values*) são os mais típicos para armazenamento de dados para análises. Eles são basicamente arquivos de textos, então podem ser visualizados e editados em qualquer editor de texto, mas os valores são separados por algum carácter específico, normalmente a vírgula (`,`), e as primeiras linhas podem ser utilizadas como *header*

dos dados.

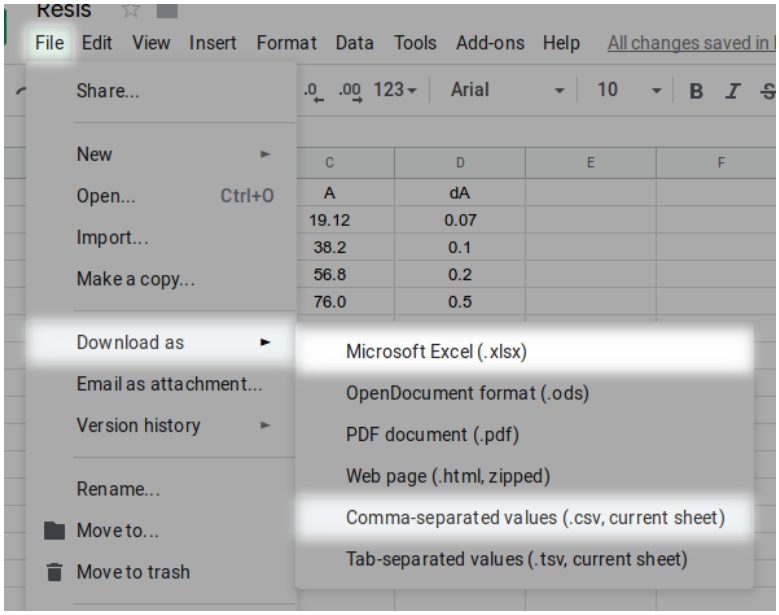

Figura 2: Recuperando os valores de uma folha de planilha no Google Planilhas

Apesar de o exemplo da figura **??** ser apenas no `Google Planilhas`, qualquer outro gerenciador de tabelas atual terá uma opção similar como parte do programa também.

Uma vez que o `CSV` esteja pronto, ele pode ser transformado em um `DataFrame` com a função `read_csv` do `Pandas`. O argumento `decimal` da função representa qual o carácter a ser entendido como separador decimal. No exemplo do código **??**, é usado a vírgula como separador, mas se por acaso estiver usando o ponto final para isso, lembre-se de mudar para `'.'`. Se o separador estiver diferente, o `Pandas` vai entender o valor como texto, o que não servirá para a montagem do gráfico depois.

Código 3: Leitura de um `DataFrame` a partir de um `CSV`

```
# -- bibliotecas e configurações -- #

dados = pd.read_csv('dados.csv', decimal=',')
```

0.3.2 Arquivos do Excel

Também é possível ler diretamente arquivos do `Excel` ou arquivos do mesmo tipo (`.xlsx`) extraído de outro programa, como na figura **??**. Para isso, além de expecificar o nome do arquivo, é necessário dizer qual a folha da planilha a ser usada com o argumento `sheet_name`, como no código **??**, tudo com a função `read_excel`. Para este tipo de arquivo, não precisa preocupar com o separador de decimal, pois a representação interna do dado já é númerica, em vez de textual, como era no `CSV`.

Código 4: Leitura de um `DataFrame` a partir de um arquivo de `Excel`

```
# -- bibliotecas e configurações -- #

dados = pd.read_excel('dados.xlsx', sheet_name='Resistor')
```

0.4 Salvado os Gráficos

A interface `pyplot` também ajuda na hora de salvar os gráficos em imagens, com a função `savefig`. Essa função reconhece o tipo do arquivo pela extensão e já vem com várias opções de fábrica. Os tipos mais importantes normalmente são `PNG` e `PDF`, que podem ser criados seguindo os códigos **??** e **??**. Para o `PNG`, também existe a opção de controlar a resolução pelo `DPI` da imagem, com o argumento `dpi`.

Código 5: Salvando o gráfico em um arquivo PNG

```
# -- depois do gráfico pronto -- #

plt.savefig('grafico.png', dpi=200)
```

Código 6: Salvando o gráfico em um arquivo PDF

```
# -- depois do gráfico pronto -- #

plt.savefig('grafico.pdf')
```

Para gráficos vetorizados, a opção mais usada é o SVG, que também é resolvido por padrão com o Matplotlib , fazendo como no código **??**. No entanto, quando se quer usar o gráfico em um documento de LaTeX, uma opção muito útil é o PGF, que não passa de um arquivo de texto com comandos do pacote pgf para ser inserido em uma figura com um comando do tipo \input{grafico.pgf}, mas tomando cuidado com as configurações.

Código 7: Salvando o gráfico em um arquivo SVG ou PGF

```
# -- depois do gráfico pronto -- #

plt.savefig('grafico.svg')
# -- ou -- #
plt.savefig('grafico.pgf')
```

1 Apresentação dos Dados

Tensão	Corrente
-3.07 V	-34.38 mA
-2.70 V	-27.96 mA
-1.69 V	-15.83 mA
-1.47 V	-10.79 mA
-0.62 V	-7.94 mA
-0.04 V	-0.05 mA
0.72 V	7.43 mA
1.25 V	13.37 mA
2.35 V	21.56 mA
2.48 V	31.34 mA
3.38 V	33.32 mA

Tabela 1: Dados de corrente para cada tensão, gerados por computador

Nesta seção, será tomado como exemplo a relação de corrente e tensão em um resistor, dado de forma teórica pela relação (**??**). Por mais que os dados usados aqui sejam os da tabela **??**, essa parte de apresentação de dados é importante para todos os tipos de análise, em especial, para dados coletados manualmente, como é o caso da maioria dos experimentos da disciplina de F 329.

$$I = \frac{1}{R} V \tag{1}$$

1.1 Dados Pontuais

Código 1.1: Gerando um gráfico de dispersão

```
# -- recupera os dados antes -- #

plt.scatter(dados['Tensão'], dados['Corrente'])

# -- depois salva o gráfico -- #
```

Para apresentar os dados coletados, a melhor opção é a função `scatter` do `pyplot` , que recebe uma lista de valores de x e outra lista de y como argumentos e desenha cada ponto (x, y).

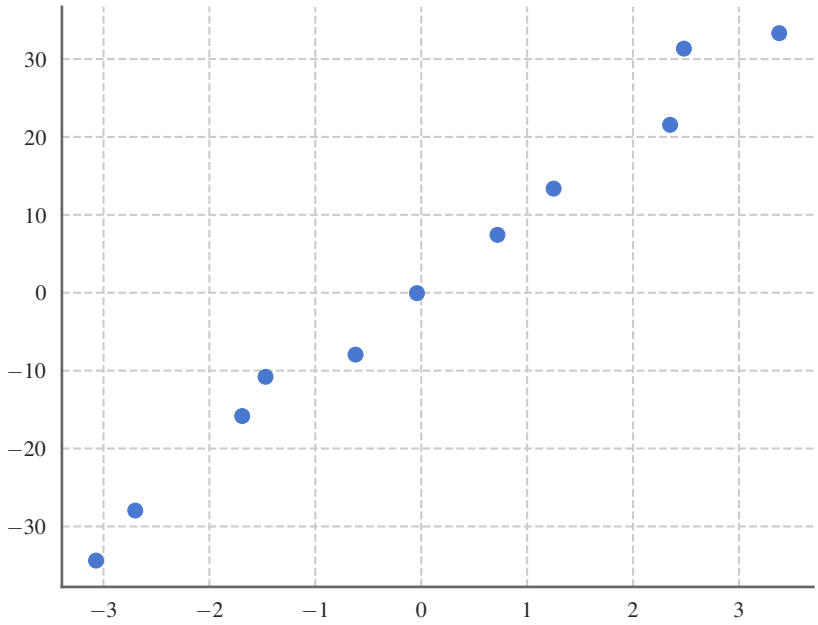

Figura 3: Gráfico de dispersão dos pontos

1.2 Texto dos Eixos e do Título

Em um gráfico como este, é necessário colocar texto no título e nos rótulos (*label*) dos eixos.

Código 1.2: Montagem dos textos do gráfico

```
# -- desenha os dados no gráfico antes -- #

plt.xlabel('Tensão [V]')
plt.ylabel('Corrente [mA]')
plt.title('Relação da Corrente pela Tensão em um Resistor')

# -- depois salva o gráfico -- #
```

1.3 Resultado

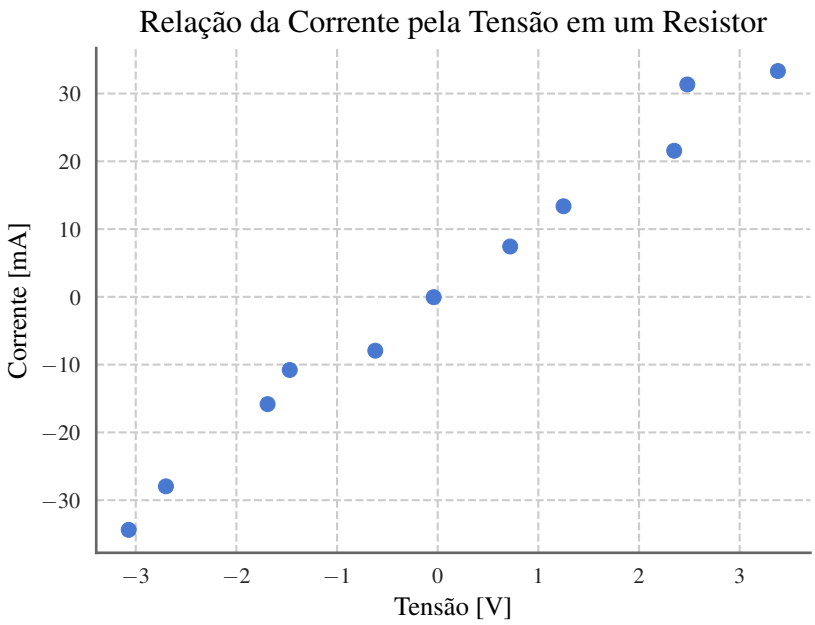

Figura 4: Exemplo de um gráfico com pontos para os dados

2 Regressão Linear

É muito comum aparecer algum tipo de relação linear entre os dados. Nesse tipo de relação costuma-se aplicar técnicas de regressão para encontrar a melhor reta que representa esses dados.

Pelo alinhamento dos pontos da seção **??** e pela equação teórica **??**, fica clara a possibilidade de se aplicar uma regressão linear e, portanto, os dados continuarão os mesmos nessa seção.

2.1 Resultados Coletados

A primeira coisa é pegar os dados e mostrar cada ponto coletado, como na seção **??**. Só que como vamos precisar dos dados depois, o melhor é separar as colunas dos dados em suas próprias variáveis para facilitar a análise dos dados depois, apesar disso não ser necessário.

Para este gráfico, como teremos dois tipos de figuras, os dados realmente coletados e a reta resultante da regressão, vamos precisar colocar a legenda. É possível já colocar o texto da legenda na construção do gráfico com o argumento `label`. Isso pode ser visto no código **??**, mas lá também tem um argumento extra, `zorder`, que controla a ordem dos desenhos e que foi usado aqui para colocar os pontos acima da reta que será feita depois.

Código 2.1: Separando e desenhando os dados pontuais

```
# -- recupera os dados antes -- #

# guarda as colunas em variáveis novas
x, y = dados['Tensão'], dados['Corrente']

# e coloca os dados pontuais no gráfico
plt.scatter(x, y, zorder=10, label='Dados Coletados')

# -- para fazer a regressão depois -- #
```

2.2 Aplicação da Regressão

Existem muitas maneiras diferentes em `Python` de se realizar um regressão linear. Uma das formas mais abrangentes é com a biblioteca `odr` do `SciPy` , feita para regressão por distância ortogonal dos dados, mas pode ser utilizada com mínimos quadrados, mudando apenas seu tipo, como no código **??**.

Código 2.2: Importando o pacote `odr` da biblioteca `SciPy`

```
# -- outras bibliotecas -- #
from scipy import odr

# -- resto do código -- #
```

Para tanto, é preciso organizar os dados em uma instância de `RealData` e usar isso para criar uma instância da `ODR` com o modelo da regressão, que em todos os exemplos desse material será o `odr.models.unilinear`. Se preferir usar o método dos mínimos quadrados é só chamar o método `set_job` com argumento `fit_type=2`. Depois é só rodar a regressão com o `run`, que retorna um objeto `Output` com várias informações, entre elas os coeficientes e a matriz de covariância deles, nos atributos `beta` e `cov_beta`, respectivamente.

Código 2.3: Regressão Linear com Mínimos Quadrados

```
# regressão linear
data = odr.RealData(x, y)
odreg = odr.ODR(data, odr.models.unilinear)
odreg.set_job(fit_type=2)  # muda para mínimos quadrados
ans = odreg.run()

a, b = ans.beta                          # y = ax + b
da, db = np.sqrt(np.diag(ans.cov_beta)) # incertezas de a e b

# mostrando os coeficientes e suas incertezas
print(f'coef. angular = ({a}+-{da}) [mA/V -> kOhm^-1 -> kS]')
print(f'coef. linear  = ({b}+-{db}) [mA]')

# -- desenha a reta resultante da regressão e completa o gráfico -- #
```

A incerteza foi adotada como o desvio padrão de cada coeficiente, que é calculado pela raiz quadrada da diagonal da matriz de covariância. Usando o `NumPy` , isso é feito com `sqrt` e `diag`. Já as últimas linhas do código **??** servem apenas para mostrar os valores dos coeficientes no terminal ou no *notebook* do `Jupyter`, quando executado.

2.3 Desenho da Regressão

Desenhar a reta da regressão é com a função `plot`, mas antes precisamos montar o rótulo que irá na legenda do gráfico. No código **??** têm dois exemplos para o rótulo, um apenas textual e outro com os coeficientes da regressão.

Código 2.4: Desenho da reta encontrada

```
# -- depois da regressão -- #

rotulo = 'Regressão Linear'
# ou
rotulo = f'Regressão: $y = ({a:.1f} \pm {da:.1f})~x + ({b:.1f} \pm {db:.1f})$'

# monta os limites para desenho da reta
X = np.linspace(min(x), max(x), num=200)
Y =  a * X + b
# e faz o gráfico dela atrá dos pontos
plt.plot(X, Y, color='red', alpha=0.4, label=rotulo)

# para exibir as legendas do gráfico
plt.legend()

# -- depois completa a formatação do gráfico e salva em uma imagem -- #
```

Qualquer tipo de curva no `Matplotlib` é feita com pontos que são ligados entre si, ou seja, são apenas segmentos de

reta conectados. Para simular a continuidade em outras curvas, costuma-se fazer vários pontos na região em que se deseja desenhar. Podemos fazer isso com a função `linspace` do `NumPy` , em que o primeiro argumento é o começo dos pontos, o segundo é o final do intervalo e o argumento `num` é a quantidade de pontos igualmente espaçados nessa região.

Por mais que isso não seja muito importante nesse caso, no código tem um exemplo de como fazer esse intervalo com 200 pontos igualmente espaçados, armazenado em `X`. Com esses pontos, é possível aplicar a função da curva nesse intervalo para encontrar a imagem desse intervalo, mas devido às técnicas de vetorização do `NumPy` , isso é feito como se fosse aplicar a operação em apenas um valor, mas a biblioteca realiza o laço implicitamente e retorna outro vetor em `Y`.

Além disso, para diferenciar entre a regressão e os valores, ela foi desenhada em vermelho, com `color='red'`, e com 40% de transparência, em `alpha=0.4`. Depois do gráfico já desenhado, foi usada ainda a função `legend` para desenhar a legenda. Por fim, basta colocar os nomes dos eixos e o título, como na seção **??**.

2.4 Resultado

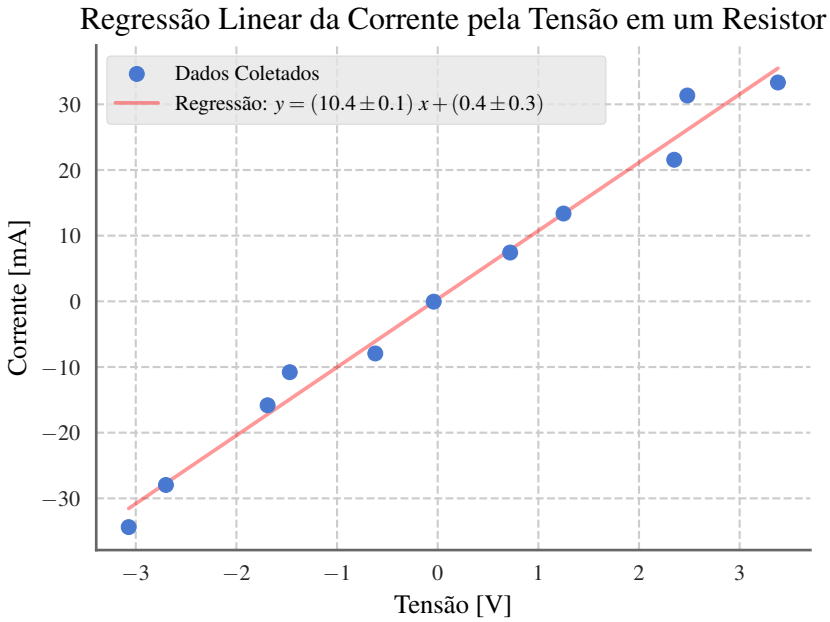

Figura 5: Exemplo de regressão linear

3 Barras de Incerteza

Como exemplo para a aplicação de barras de incerteza, continuaremos com os mesmo dados da seção **??**, porém agora com as incertezas associadas a cada medida, que foram criadas, novamente, com o auxílio de um computador, e podem ser vistas na tabela **??**.

Tensão [V]	Corrente [mA]
-3.07 ± 0.29	-34.38 ± 3.16
-2.70 ± 0.27	-27.96 ± 3.07
-1.69 ± 0.18	-15.83 ± 1.65
-1.47 ± 0.16	-10.79 ± 0.94
-0.62 ± 0.06	-7.94 ± 0.72
-0.04 ± 0.02	-0.05 ± 0.11
0.72 ± 0.08	7.43 ± 0.72
1.25 ± 0.13	13.37 ± 1.36
2.35 ± 0.20	21.56 ± 2.16
2.48 ± 0.27	31.34 ± 2.90
3.38 ± 0.36	33.32 ± 3.33

Tabela 2: Dados de corrente por tensão com suas incertezas

3.1 Dados Pontuais com Barras de Incerteza

Código 3.1: Separação das colunas e gráfico com barras de incerteza

```
# -- recupera os dados antes -- #

# guarda as colunas em variáveis novas
x, dx = dados['V'], dados['dV']
y, dy = dados['I'], dados['dI']

# e coloca os dados pontuais com barras de incerteza
plt.errorbar(
    x, y, xerr=dx, yerr=dy,
    fmt='o', elinewidth=2/3, capsize=2, capthick=2/3, color='black',
    zorder=10, label='Dados Coletados'
)

# -- para fazer a regressão depois -- #
```

Para facilitar, as colunas foram separadas em `x` e `y`, com suas incertezas `dx` e `dy` em cada ponto. Agora, para desenhar as barras de incerteza, basta utilizar a função `errorbar` com os argumentos `xerr=dx` e `yerr=dy`. O argumento `fmt` foi usado para fazer com que os dados fossem desenhados como pontos, que é o formato `'o'`. O formato `'.'` pode ser usado se preferir pontos menores ou quando a incerteza é muito pequena.

Além disso, os argumentos `elinewidth`, `capsize` e `capthick` controlam a grossura da barra de incerteza, o comprimento do topo da barra e a grossura desse topo, respectivamente. A cor dos pontos e das barras foi alterado para preto com `color='black'`. Novamente, esses desenhos foram colocados com ordem alta de desenho e com um rótulo para a legenda.

3.2 Regressão Linear com Incertezas

A regressão com incertezas fica bem parecida com a da seção **??**, utilizando o pacote `odr`. A primeira diferença é que o `RealData` agora tem as incertezas de x e y em `sx` e `sy`. O problema com a regressão da seção anterior é que o método dos mínimos quadrados não é capaz de analisar as incertezas em x, então vamos deixar a `ODR` aplicar a regressão por distância ortogonal, como no código **??**.

Código 3.2: Regressão Linear com Inceretzas

```
# regressão linear com incertezas
data = odr.RealData(x, y, sx=dx, sy=dy)
odreg = odr.ODR(data, odr.models.unilinear)
#odreg.set_job(fit_type=2)  # mínimos quadrados não suporta incertezas em x
ans = odreg.run()

a, b = ans.beta                          # y = ax + b
da, db = np.sqrt(np.diag(ans.cov_beta)) # incertezas de a e b

# mostrando os coeficientes e suas incertezas
print(f'coef. angular = ({a}+-{da}) [mA/V -> kOhm^-1 -> kS]')
print(f'coef. linear  = ({b}+-{db}) [mA]')

# -- desenha a reta resultante da regressão e completa o gráfico -- #
```

3.3 Resultados

Note que os coeficientes a e b da regressão em **??**, tanto em seus valores quanto nas suas incertezas, são levemente diferentes dos da figura **??**, mesmo com os dados numéricos idênticos. A diferença aqui se deve ao método de regressão diferente, mas mesmo utilizando a regressão ortogonal na seção **??**, o valor difiriria já que agora as incertezas dos dados estão sendo levadas em conta. ■

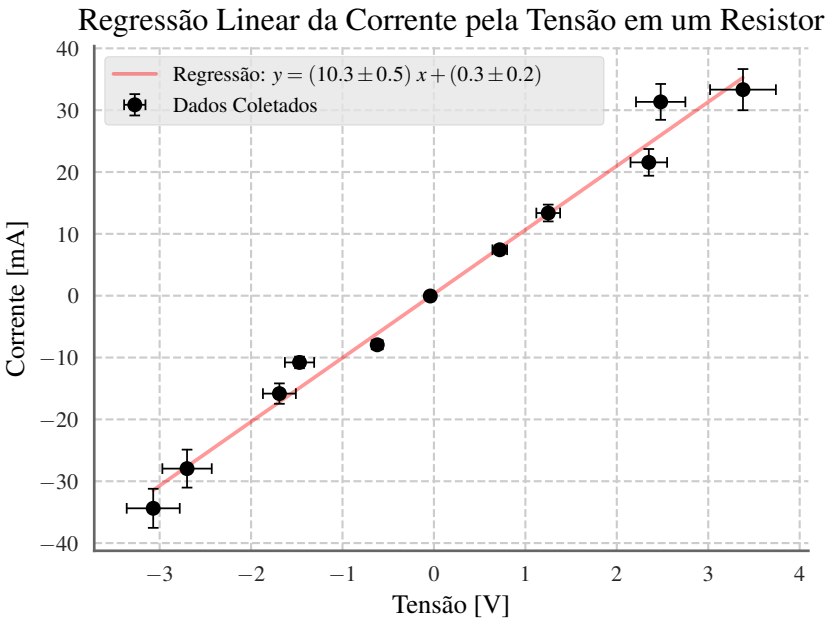

Figura 6: Exemplo de baras de incerteza

4 Escala Logarítmica

Em várias outras vezes, no entanto, os dados não apresentam relação linear. Em alguns desses caso é possível encontrar alguma técnica de linearização que transforma os dados para novos valores dependentes, mas que se relacionam de maneira linear. Algo como a relação **??**.

$$f(x,y) = a + b\,g(x,y) \tag{2}$$

Dentre as técnicas mais comuns, muitas envolvem a aplicação de logaritmos para linearizar algum monômio, isto é, nos casos de $y \propto x^k$, ou alguma relação exponencial, $y \propto k^x$. Para esses casos, é comum a utilização de escala logarítmica na intenção de se observar melhor os dados, em que $f(x,y) = \log(y)$ e $g(x,y) = \log(x)$.

4.1 Gráfico Log-Log

$$R_d = \frac{R_1 R_2}{R_x} \tag{3}$$

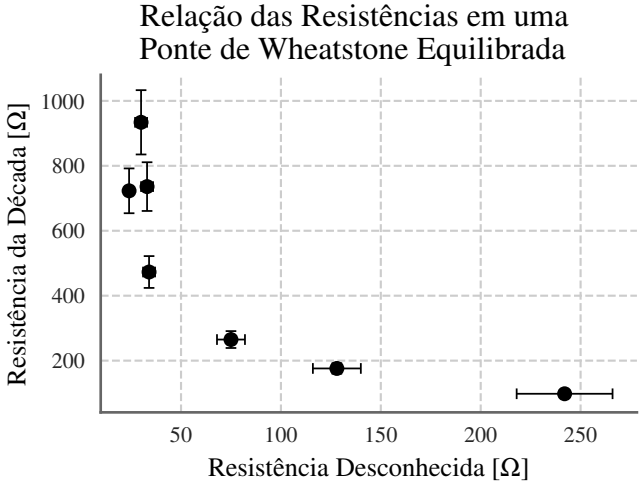

Figura 7: Gráfico da ponte de Wheatstone (eq. **??**)

Se imaginarmos os dados do gráfico **??** como parte de um caso da ponte de Wheatstone dado pela equação **??**, sendo

R_x a resistência desconhecida e R_d a resistência da década, podemos aplicar a seguinte técnica de linearização:

$$\begin{aligned}\log(R_d) &= \log\left(R_1 R_2\ (R_x)^{-1}\right) \\ &= \log(R_1 R_2) + \log\left((R_x)^{-1}\right) \\ &= \log(R_1 R_2) - \log(R_x)\end{aligned}$$

Código 4.1: Contrução de um gráfico `log-log` com barras de incerteza

```
    # -- recupera os dados antes -- #

    x, dx = dados['Rx'], dados['dRx']
    y, dy = dados['Rd'], dados['dRd']

    plt.errorbar(
        x, y, xerr=dx, yerr=dy,
        fmt='o', elinewidth=2/3, capsize=2, capthick=2/3, color='black',
        zorder=10, label='Dados Coletados'
    )

    # escala log-log
    plt.xscale('log')
    plt.yscale('log')

    # linhas de grid internas
    plt.grid(True, which='minor', axis='both', color='0.9')

    # -- depois completa os textos e salva a imagem -- #
```

Portanto, podemos montar um gráfico `log-log` de R_d por R_x, que pode ser feito como em **??**. O `pyplot` tem as funções `xscale` e `yscale` para mudar as escalas dos eixos, que podem receber como argumento `'linear'` e `'log'` por padrão, além de algumas outras opções, mas é possível adicionar escalas customizadas no `Matplotlib` .

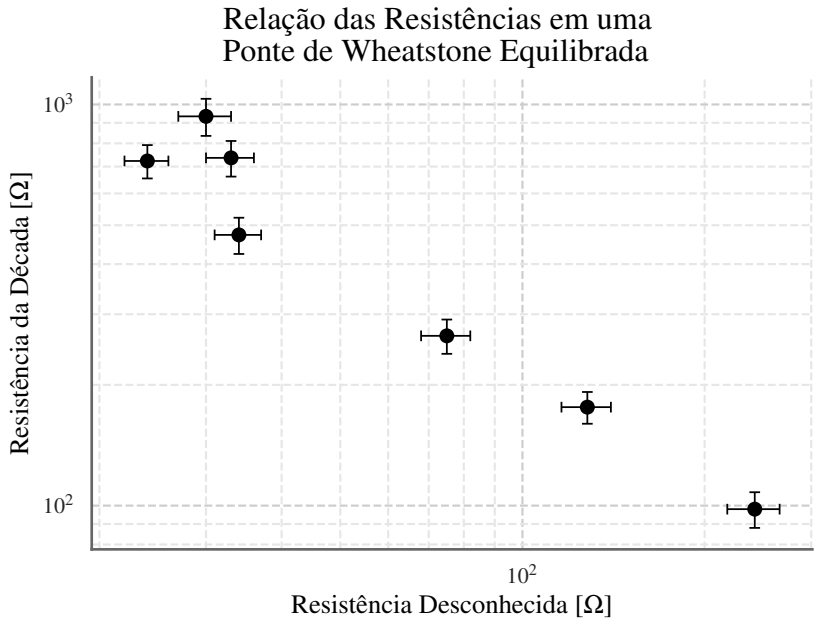

Figura 8: Gráfico `log-log` da Ponte de Whetstone (**??**)

Como as linhas de *grid* ficaram bem espaçadas, foram colocadas linhas internas com a função `grid` do `pyplot` e os argumentos `True`, para mostrar as linhas, e `which='minor'`, para desenhar entre as linhas principais. Elas também foram colocadas em ambos os eixos e com cor 90% branca, com as opções `axis='both'` e `color='0.9'`.

4.2 Gráfico Semi-Log

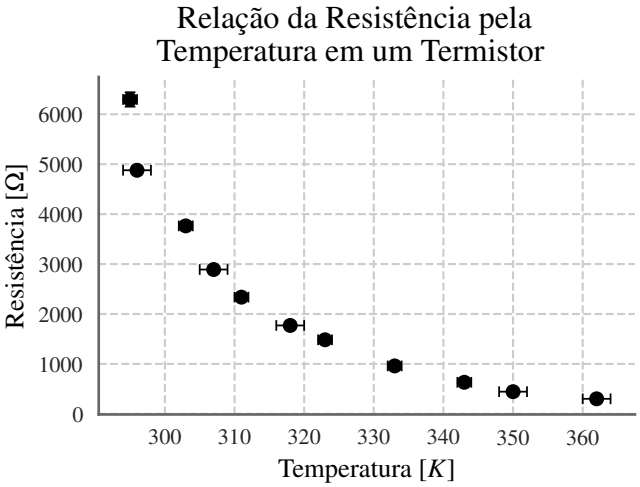

Figura 9: Gráfico do Termistor (eq. **??**)

$$R = A\ \exp\left(B\ T^{-1}\right) \tag{4}$$

Agora com os dados da figura **??** e a relação (**??**), com R como a resistência e T^{-1} o inverso da temperatura, a linearização se torna:

$$\begin{aligned}\ln(R) &= \ln\left(A\ \exp\left(B\ T^{-1}\right)\right)\\ &= \ln(A) + \ln\left(\exp\left(B\ T^{-1}\right)\right)\\ &= \ln(A) + B\ T^{-1}\end{aligned}$$

Que pode ser usada em um gráfico `semi-log` de $R \times T^{-1}$, feito similar à seção **??**, que pode ser visto no código **??** com a figura **??** como resultado. Uma das diferenças específicas da equação (**??**) é a transformação do eixo x de temperatura para T^{-1}.

Código 4.2: Contrução de um gráfico `semi-log` com barras de incerteza

```
# -- recupera os dados antes -- #

T, dT = dados['T'], dados['dT']
y, dy = dados['R'], dados['dR']

# transformação do eixo x
x = 1/T
dx = dT/T**2

plt.errorbar(
    x, y, xerr=dx, yerr=dy,
    fmt='o', elinewidth=2/3, capsize=2, capthick=2/3,
    color='black', zorder=10,
)

# escala logarítmica
plt.yscale('log')

# linhas de grid internas no eixo y
plt.grid(True, which='minor', axis='y', color='0.9')

# -- depois completa os textos e salva a imagem -- #
```

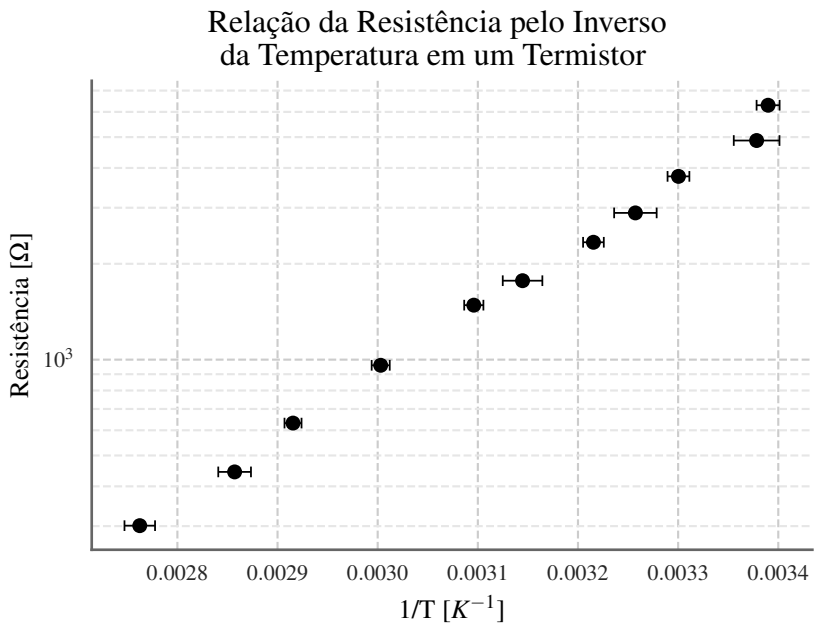

Figura 10: Gráfico `semi-log` do Termistor (eq. **??**)

A escolha de transformar o eixo, como no eixo x da figura **??**, ou mudar a escala, que foi feito no eixo y, é uma parte importante da montagem dos gráficos. Normalmente, é preferível mexer na escala quando é uma escala conhecida ou quando se busca explicitar os valores originais do eixo. Nos casos em que isso não é necessário ou não é possível, a transformação dos eixos acaba sendo uma opção menos confusa. ■

4.3 Regressão em Escala Logarítmica

Código 4.3: Aplicando a regressão ortogonal depois de uma transformação númerica dos eixos

```
# transforma os dados para a linearização
logx = np.log10(x)
logy = np.log10(y)

dlogx = dx / (x * np.log(10))
dlogy = dy / (y * np.log(10))

# regressão linear com incertezas
data = odr.RealData(logx, logy, sx=dlogx, sy=dlogy)
odreg = odr.ODR(data, odr.models.unilinear)
ans = odreg.run()

a, b = ans.beta                         # logy = a logx + b
da, db = np.sqrt(np.diag(ans.cov_beta)) # incertezas de a e b

# mostrando os coeficientes e suas incertezas
print(f'coef. angular = {a}+-{da}')
print(f'coef. linear  = {b}+-{db}')

# -- desenha a reta resultante da regressão e completa o gráfico -- #
```

A regressão de uma curva de um monômio ou de uma exponencial é possível com técnicas de regressão não-linear, só que essas técnicas não cabem no escopo dessa matéria. Uma outra opção muito utilizada é encontrar uma linearização, como na equação (**??**), e, com a nova relação linear de $f(x,y) \times g(x,y)$, aplicar a regressão como da seção **??**.

O único detalhe é que é preciso encontrar os valores de $f(x,y)$ e $g(x,y)$ e suas incertezas para cada par (x,y) dos dados e só com esses valores pode-se encontrar os coeficientes a e b, como foi feito no código **??**. No caso, como a transformação é $f(x,y) = \log_{10}(x)$, a incerteza fica $\sigma_{\log x} = \frac{1}{\ln(10)} \frac{\sigma_x}{x}$, sendo que para o eixo y a transformação é igual.

Foram utilizadas as funções matemáticas do `NumPy` , pelas técnicas de vetorização oferecidas, sendo elas `log` e `log10`, que equivalem a $\ln(x)$ e $\log_{10}(x)$.

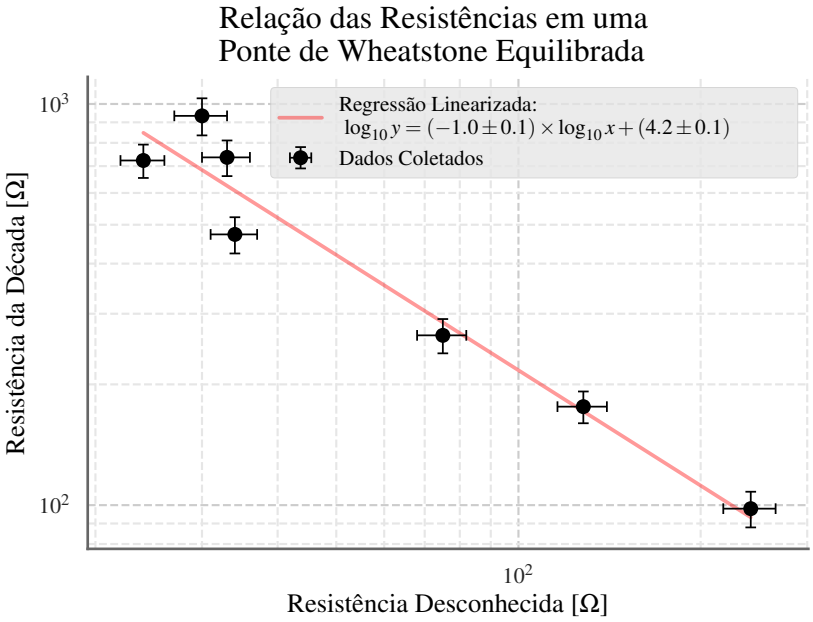

Figura 11: Exemplo de regressão em escala `log-log`

Perceba que a regressão obteve um coeficiente angular próximo de -1, como era o esperado pela equação inicial (**??**).

5 Equação Característica

Um bom exemplo de equação característica a ser encontrada é relação do termistor (eq. **??**). Então, os dados serão os mesmo da seção **??**.

5.1 Encontrando os Coeficientes

Para encontrar os coeficientes da equação característica, o primeiro passo normalmente é conseguir uma relação de linearização para poder aplicar alguma técnica de regressão linear e encontrar os coeficientes dessa relação. Isso pode ser feito como na seção **??**.

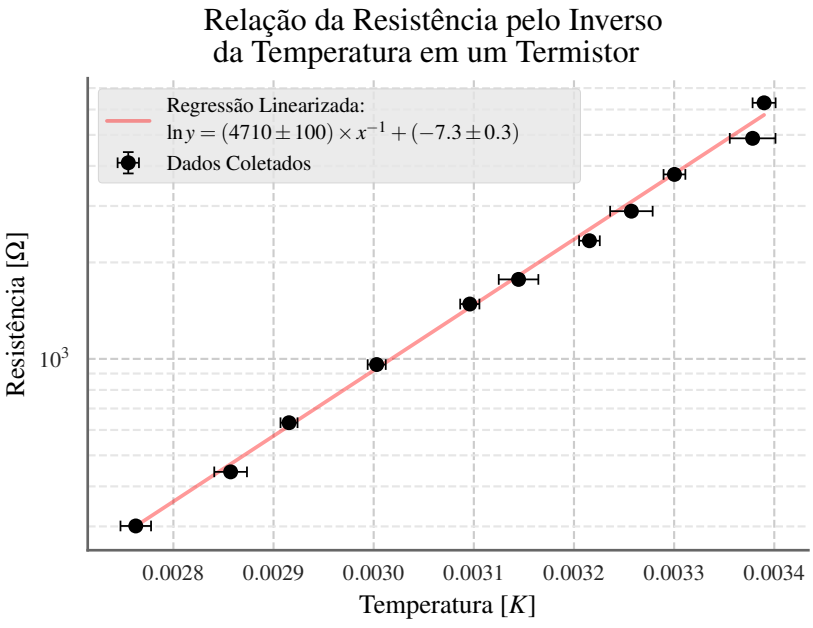

Figura 12: Regressão da equação linearizada do termistor

Código 5.1: Código completo para encontrar a equação característica no exemplo do termistor

```
# -- importa as bibliotecas -- #

# coletando os dados
dados = pd.read_csv('termistor.tsv', decimal=',', sep='\t')

T, dT = dados['T'], dados['dT']
R, dR = dados['R'], dados['dR']

# valores medidos
plt.errorbar(
    R, T, xerr=dR, yerr=dT,
    fmt='o', elinewidth=2/3, capsize=2, capthick=2/3, color='black',
    zorder=10, label='Dados Coletados'
)

# transformação dos eixos
logR = np.log(R)
dlogR = dR / R

Tinv = 1/T      # inverso da temperatura
dTinv = dT/T**2

# regressão linear com incertezas
data = odr.RealData(Tinv, logR, sx=dTinv, sy=dlogR)
odreg = odr.ODR(data, odr.models.unilinear)
ans = odreg.run()

# coeficientes: lnR = a T^-1 + b
a, b = ans.beta
da, db = np.sqrt(np.diag(ans.cov_beta))

# transforma para R = A e^(B/T)
A = np.exp(b)
dA = db * np.exp(b)

B = a
dB = da

# mostra os coeficientes da eq. característica
print(f'valor inicial   = {A}+-{dA}')
print(f'fator de cresc. = {B}+-{dB}')

# desenha a eq. característica
rotulo = f'''Equação Característica:
$T = \\frac{{({B:.0f} \pm {dB:.0f})}}{{\ln(R) - \ln({A:.4f} \pm {dA:.4f})}}$'''

Rs = np.linspace(min(R) - 2*min(dR), max(R) + 2*max(dR), num=200)
# então T = B/ln(R/A) = B/(lnR-lnA)
Ts = B/(np.log(Rs) - np.log(A))
plt.plot(Rs, Ts, color='red', alpha=0.4, label=rotulo)

# formatações do gráfico
plt.xlabel('Resistência [$\Omega$]')
plt.ylabel('Temperatura [$K$]')
plt.title(f'''Relação da Temperatura pela
Resistência em um Termistor''')
plt.legend()

# -- salva o gráfico -- #
```

Depois que os coeficientes da reta foram encontrados, é preciso transformar os coeficientes para a forma inicial da equação. No caso do termistor, isso seria $A = e^b$ e $B = a$, com incertezas $\sigma_A = e^b \sigma_b$ e $\sigma_B = \sigma_a$. Em **??**, todas as transformações, juntamente com a regressão, são feitas diretamente no código, como exemplo. No exemplo também é usada a função `exp`, que é apenas $\exp(x) := e^x$.

Note que para este exemplo, no entanto, faz mais sentido tratar a equação característica da forma inversa (eq. **??**). Por causa disso, os eixos do gráfico **??** estão invertidos em relação à figura **??**.

$$T = \frac{B}{\ln(R) - \ln(A)} \tag{5}$$

5.2 Resultado

O código **??** produz como resultado a figura **??**.

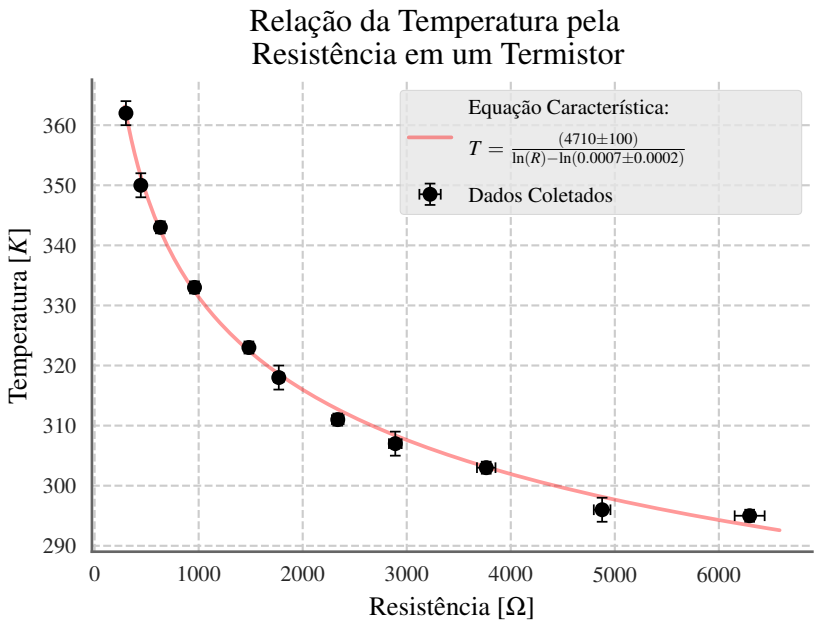

Figura 13: Exemplo de Equação Característica do Termistor

5.3 Banda de Incerteza

Para a equação **??**, como A e B são constantes, bem como suas incertezas, σ_A e σ_B, a equação da incerteza σ_T de T fica como função apenas de R e σ_R:

$$\sigma_T = \sigma_T(R, \sigma_R) = \frac{\sqrt{(\ln(R) - \ln(A))^2 \sigma_B^2 + \left(\frac{\sigma_A}{A}\right)^2 + \left(\frac{\sigma_R}{R}\right)^2}}{(\ln(R) - \ln(A))^2} \tag{6}$$

Se a equação característica for usada com um equipamento já pré-estabelecido, é possível assumir que a incerteza da medida, no caso a resistência, vai seguir uma função $\sigma_R = \sigma_R(R)$ conhecida para cada R. Dependendo da situação, é possível assumir que σ_R é constante ou que a incerteza relativa $\frac{\sigma_R}{R}$ é constante. Para este exemplo, vamos assumir as equações (**??**), (**??**) e (**??**).

$$\text{resolução}(R) = \begin{cases} 0.1\,\Omega, & \text{se } 0\,\Omega \leq R \leq 600\,\Omega \\ 1\,\Omega, & \text{se } 600\,\Omega < R \leq 6\,\text{k}\Omega \\ 10\,\Omega, & \text{se } 6\,\text{k}\Omega < R \leq 60\,\text{k}\Omega \\ \text{indefinido}, & \text{caso contrário} \end{cases} \tag{7}$$

$$\mu_{\text{calibração}}(R) = \begin{cases} 1\%R + 3 \times \text{resolução}(R), & \text{se } 0\,\Omega \leq R \leq 600\,\Omega \\ 0.5\%R + 2 \times \text{resolução}(R), & \text{se } 600\,\Omega < R \leq 60\,\text{k}\Omega \\ \text{indefinido}, & \text{caso contrário} \end{cases} \tag{8}$$

$$\sigma_R(R) = \sqrt{\left(\frac{\text{resolução}(R)}{2\sqrt{3}}\right)^2 + \left(\frac{\mu_{\text{calibração}}(R)}{\sqrt{3}}\right)^2} \tag{9}$$

Código 5.2: Implementação das funções para o cálculo da incerteza

```
# -- funções de cálculo de incerteza -- #

def resolucao(R):
    if      0 <= R <= 600:
        return 0.1
    elif   600 < R <= 6_000:
        return 1
    elif 6_000 < R <= 60_000:
        return 10
    else:
        raise ValueError


def dR_calibracao(R):
    d = resolucao(R)
    if    0 <= R <= 600:
        return 0.010 * R + 3 * d
    elif 600 < R <= 60_000:
        return 0.005 * R + 2 * d
    else:
        raise ValueError


def dist_retangular(a):
    return a / (2 * np.sqrt(3))


@np.vectorize
def dR_total(R):
    dR_calib = dist_retangular(2 * dR_calibracao(R))
    dR_leitura = dist_retangular(resolucao(R))

    return np.sqrt(dR_calib**2 + dR_leitura**2)

# -- #
```

Código 5.3: Cálculo da incerteza e desenho da banda de incerteza

```
# -- depois de mostrar a eq. característica -- #

# incerteza de cada R mostrado na reta
dRs = dR_total(Rs)

# e a inceteza de cada T
lnRmlnA = np.log(Rs) - np.log(A)
dTs = np.sqrt((lnRmlnA * dB)**2 + (dA/A)**2 + (dRs/Rs)**2)/lnRmlnA**2

# desenho da banda de incerteza
plt.fill_between(
    Rs, Ts-dTs, Ts+dTs,
    color='red', alpha=0.15,
    label='Faixa de Incerteza'
)

# -- coloca os textos e a legenda, depois salva o gráfico -- #
```

No código **??** foi necessário utilizar a função `sqrt`, que é apenas a raiz quadrada: $\text{sqrt}(x) \coloneqq \sqrt{x};$. Para facili-

tar o código, foi utilizada também a função `vectorize` como decorador, que transforma uma função qualquer de `Python` em uma função vetorizada como as funções de `NumPy` . Isso poderia ser feito com um simples `for` também: `dRs = [dR_total(R) for R in Rs]`; mas isso retorna uma lista, que às vezes pode causar problemas com os vetores de `NumPy` .

O código **??** é similar aos outros códigos de montagem de gráfico, só que com uma nova função, `fill_between`, que desenha uma região delimitada por duas curvas `y1` e `y2`, sendo $y_1 = T - \sigma_T$ e $y_2 = T + \sigma_T$, formando a faixa de incerteza. Essa região foi montada com a mesma cor da curva característica, com uma opacidade menor (`alpha=0.15`), o que pode ser visto na figura **??**.

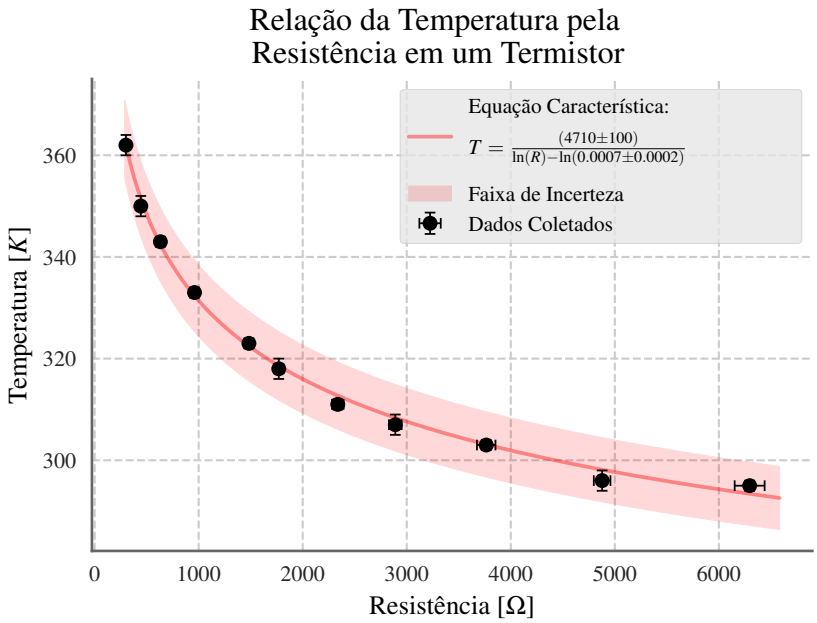

Figura 14: Exemplo de Equação Característica com Banda de Incerteza

Normalmente não é possível assumir uma incerteza determinística $\sigma_y(x, \sigma_x) = \sigma_y(x)$, como foi feito com a incerteza da temperatura nessa seção. Para esses casos, é possível desconsiderar a incerteza em x, $\sigma_x = 0$, que serveria para mostrar a menor incerteza que a equação característica encontrada consegue alcançar. No entanto, se esse for o caso, pode ser que a banda de incerteza acabe não sendo muito útil e apenas deixe a leitura do gráfico mais complicada. ■

6 Gráficos de Múltiplas Variáveis

Para os casos em que é necessário apresentar dados com mais de uma váriavel dependente de um mesmo dado x, existe a opção de gráficos múltiplos. Eles servem para comparar as relações do tipo $y_1 = f(x)$ e $y_2 = g(x)$, quando x, y_1 e y_2 são medidos em conjunto.

Em experimentos com circuitos, esse tipo de dado aparece, por exemplo, na medição de tensão em nós diferentes para a comparação de seus comportamentos no tempo. É o caso do circuito da figura **??**, cujos dados foram os da tabela **??**.

Tempo	V_1	V_2	Corrente
0.0 ms	0.0 V	2.0 V	-20 mA
0.4 ms	1.9 V	2.3 V	-3 mA
0.8 ms	3.6 V	2.2 V	14 mA
1.2 ms	4.7 V	1.8 V	29 mA
1.6 ms	5.0 V	1.1 V	39 mA
2.0 ms	4.5 V	0.2 V	43 mA
	...		

Tabela 3: Dados gerados com simulador. Primeiros 6 valores.

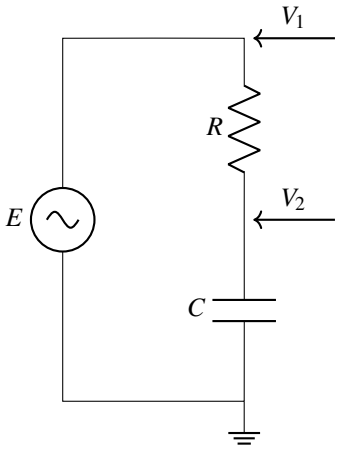

Figura 15: Circuito de defasagem da tensão por um capacitor

6.1 Gráficos de Eixos Compartilhados

Código 6.1: Montagem completa do gráfico de duas varíaveis com eixos compartilhados

```
# -- importa os dados antes -- #

# separando os dados
t = dados['t']
V1 = dados['V1']
V2 = dados['V2']
I = dados['I']

# desenha os gráficos
plt.plot(t, V1, color='red', alpha=0.6, label='Tensão de Entrada')
plt.plot(t, V2, color='blue', alpha=0.6, label='Tensão de Saída')

# coloca legenda e os textos
plt.legend()
plt.xlabel('Tempo [$ms$]')
plt.ylabel('Tensão [$V$]')
plt.title('Comportamento da Tensão em um Circuito RC')

# -- depois salva a figura -- #
```

Esse método é melhor para visualizar a diferença de escala entre os dados, mas, se algum dos dados tem uma escala muito diferente, o gráfico pode acabar pecando na percepção dos dados. Um dos maiores limites para esse método, no entanto, é que as variáveis dependentes precisam ter a mesma motivação física e, por causa disso, a mesma gradeza, caso contrário, o eixo compartilhado entre elas perde completamente o sentido.

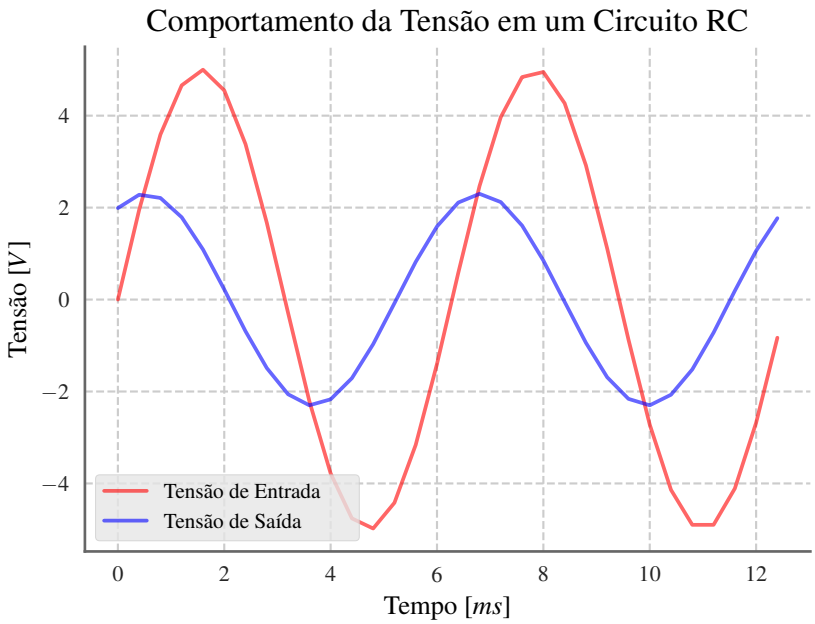

Figura 16: Exemplo de gráfico com as curvas das duas tensões V_1 e V_2

6.2 Gráficos com Apenas a Abscissa Comum

Se as escalas entre y_1 e y_2 forem muito diferentes ou se as grandezas forem diferentes, uma opção viável é montar um gráfico de três eixos. Pra fazer isso no `Matplotlib` é preciso tratar diretamente dos *eixos* do gráfico, que são instâncias da classe `Axes`. Normalmente, a interface `pyplot` gera esse objetos automaticamente, quando necessário, mas aqui vamos precisar tratar da construção deles também, que pode ser feito com a função `subplot`, como no código **??**.

O outro objeto, com o terceiro eixo, é um tipo de eixo chamado de *gêmeo*, já que o eixo x dele é o mesmo que o do primeiro. Para criar um eixo gêmeo em x é com o método `twinx`. Para desenhar em cada eixo, as funções são parecidas com as do `pyplot` , como função `plot`, mas aqui se deve tomar cuidado com qual eixo se deseja desenhar.

Código 6.2: Montagem completa do gráfico de duas varíaveis com abscissa compartilhada

```
# -- importa e separa os dados antes -- #

# criação dos eixos
eixo_esq = plt.subplot()
eixo_dir = eixo_esq.twinx()

# desenho e formatações do eixo esquerdo
eixo_esq.plot(t, V2, color='red', alpha=0.6, label='Tensão')
eixo_esq.set_ylabel('Tensão [$V$]')
eixo_esq.grid(False)

# mesmo para o eixo direito
eixo_dir.plot(t, I, color='blue', alpha=0.6, label='Corrente')
eixo_dir.set_ylabel('Corrente [$mA$]')
eixo_dir.grid(False)

# título e rótulo do eixo compartilhado
eixo_esq.set_title('Relação de Corrente e Tensão em um Capacitor')
eixo_esq.set_xlabel('Tempo [$ms$]')
# coluna do eixo da direita
eixo_dir.spines['right'].set_visible(True)
# remove a coluna sobreposta
eixo_dir.spines['left'].set_visible(False)

# cores das colunas dos eixos
eixo_esq.spines['left'].set_color('red')
eixo_esq.spines['left'].set_alpha(0.6)
eixo_dir.spines['right'].set_color('blue')
eixo_dir.spines['right'].set_alpha(0.6)

# -- salva a figura -- #
```

Os métodos de formatação dos `Axes` também são semelhantes às funções de formatação do `pyplot` , como `grid` e `set_ylabel`. Os métodos `set_xlabel` e `set_title` também são idênticos aos do `pyplot` , mas nesse gráfico tem um detalhe a mais: como o segundo eixo é um gêmeo em x do primeiro eixo, o título e o rótulo do eixo x não podem ser modificados pelo eixo gêmeo.

Apesar de ter gerado o eixo gêmeo, o `Matplotlib` está configurado para não mostrar a coluna (*spine*) na parte direita do gráfico. Isso pode ser alterado mexendo diretamente com as instâncias da classe `Spine`, que podem ser acessadas pelo eixo, através de um dicionário que relaciona o nome da coluna com o objeto dela. No caso, vamos acessar a coluna de nome `'right'` e mudar sua visibilidade com `set_visible`.

Para deixar os eixos mais reconhecíveis, é possível colocar legenda neles, mas elas seriam redundantes aqui. Uma outra opção, é o reconhecimento por cor, que pode ser feito na coluna do eixo com os métodos `set_color` e `set_alpha`. Também foi removido a coluna esquerda do eixo gêmeo para evitar sobreposição visual das colunas.

> As vezes, os gráficos com múltiplas curvas podem ficar sobrecarregados de informação. Quando isso acontece, o melhor é separar os dados em gráficos distintos pra manter a legibilidade. Gráficos de três eixos podem ficar complicados com facilidade. ■

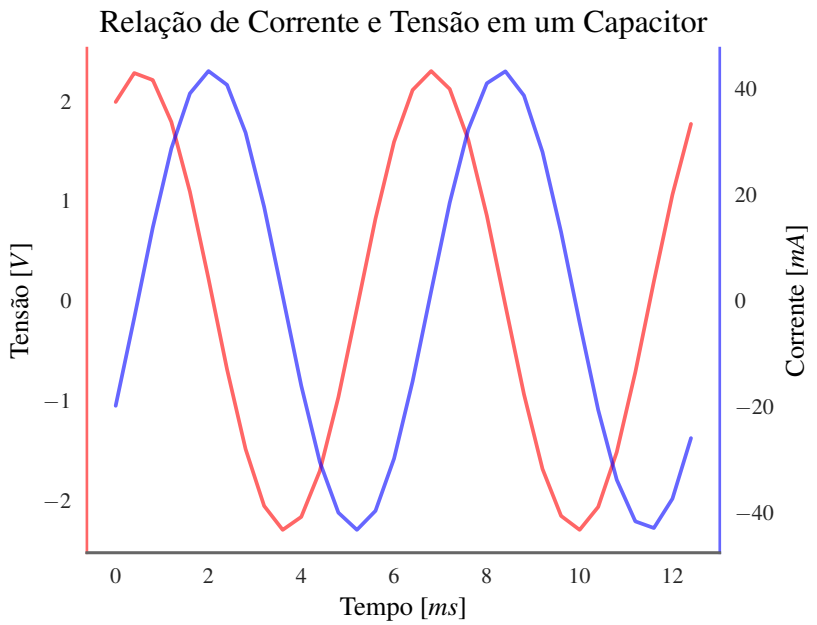

Figura 17: Exemplo de gráfico de três eixos para a corrente e a tensão em cada tempo

6.3 Gráficos de Eixos Separados

A opção mais genérica para mostrar os dois canais ao mesmo tempo é colocar cada um em seu próprio gráfico com seus próprios eixos. Normalmente, isso é feito com duas imagens diferentes, mas para facilitar a comparação entre os gráficos, eles podem ser feitos em uma mesma figura. No `Matplotlib` , isso pode ser feito com a ideia de *subplots*.

A função do `pyplot` para isso é `subplots`, que retorna a instância da figura, um objeto `Figure`, e as instâncias dos eixos, montados em uma matriz de eixos dada pelo número de linhas e de colunas, que no código **??** é `nrows=2` e `ncols=1`. No exemplo, os eixos x de cada gráfico são os mesmo, então faz sentido eles serem representados do mesmo jeito, que foi pra isso o argumento `sharex=True`.

Além disso, os eixos foram separados em superior e inferior, sendo que o título foi colocado apenas no superior e o rótulo do eixo x, só no inferior.

Código 6.3: Montagem completa do gráfico de duas varíaveis com abscissa compartilhada

```
# -- importa e separa os dados antes -- #

_, eixos = plt.subplots(nrows=2, ncols=1, sharex=True)
eixo_sup = eixos[0]
eixo_inf = eixos[1]

# eixo superior
eixo_sup.plot(t, V2, color='red', alpha=0.6)
eixo_sup.set_ylabel('Tensão [$V$]')

# eixo inferior
eixo_inf.plot(t, I, color='blue', alpha=0.6)
eixo_inf.set_ylabel('Corrente [$mA$]')

# formatações gerais
eixo_sup.set_title('Relação de Corrente e Tensão em um Capacitor')
eixo_inf.set_xlabel('Tempo [$ms$]')

# -- depois salva a figura -- #
```

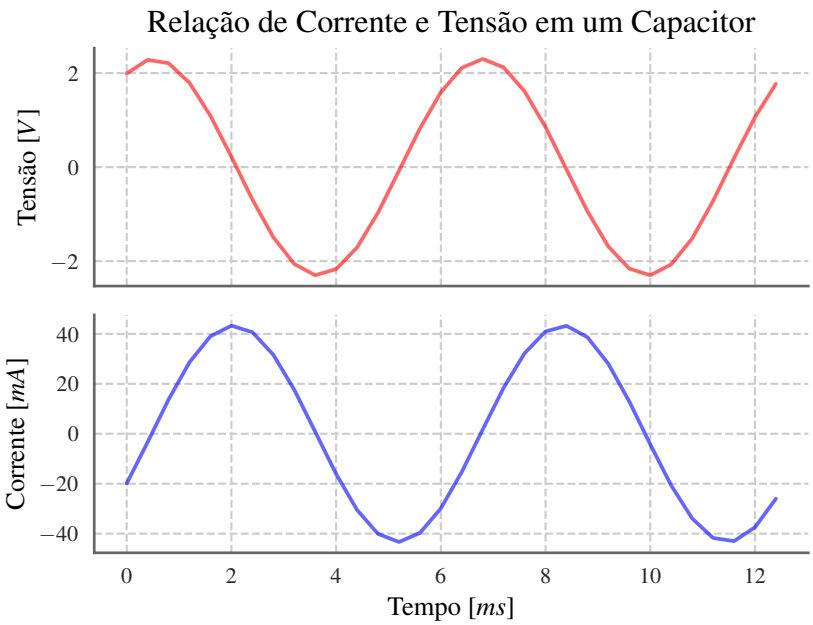

Figura 18: Exemplo de imagem com dois gráficos

7 Curvas de Nível

Curvas de nível servem para representar dados tridimensionais em um plano. As três variáveis para esse tipo de gráfico são x e y independentes e $z = f(x,y)$.

Serão usados, como exemplo, dados semelhantes aos do experimento sobre potencial elétrico entre barras de cobre em uma solução condutiva. Nesse caso, as variáveis independentes são as distâncias x e y no plano e a variável dependente é o potencial V de cada ponto.

Posição X [cm]	Posição Y [cm]	Tensão [V]
-12.0 ± 0.5	-7.0 ± 0.5	0.330 ± 0.024
-8.0 ± 0.5	-7.3 ± 0.5	0.329 ± 0.024
-4.0 ± 0.5	-7.3 ± 0.5	0.330 ± 0.024
0.0 ± 0.5	-7.4 ± 0.5	0.328 ± 0.024
4.0 ± 0.5	-7.4 ± 0.5	0.327 ± 0.023
8.0 ± 0.5	-7.3 ± 0.5	0.328 ± 0.024
12.0 ± 0.5	-7.4 ± 0.5	0.332 ± 0.024
-12.0 ± 0.5	-4.6 ± 0.5	0.661 ± 0.048
-8.0 ± 0.5	-4.5 ± 0.5	0.661 ± 0.048
-4.0 ± 0.5	-4.6 ± 0.5	0.663 ± 0.047
	...	

Tabela 4: Primeiros 10 pontos coletados.

7.1 Curvas por Malha Triangular

Código 7.1: Desenho das curvas de nível

```
# separando os dados
x, dx = dados['x'], dados['dx']
y, dy = dados['y'], dados['dy']
V, dV = dados['V'], dados['dV']

# faz o contorno
plt.tricontour(x, y, V)
```

A forma mais abrangente de se desenhar as curvas de nível é formando uma malha triangular dos pontos, onde cada

ponto (x, y) é tratado como um vétice de um triângulo e a váriavel dependente é usada para montar planos com os vértices de cada triângulo. Por mais que o processo pareça um pouco complicado, isso é feito com apenas uma função, `tricontour`.

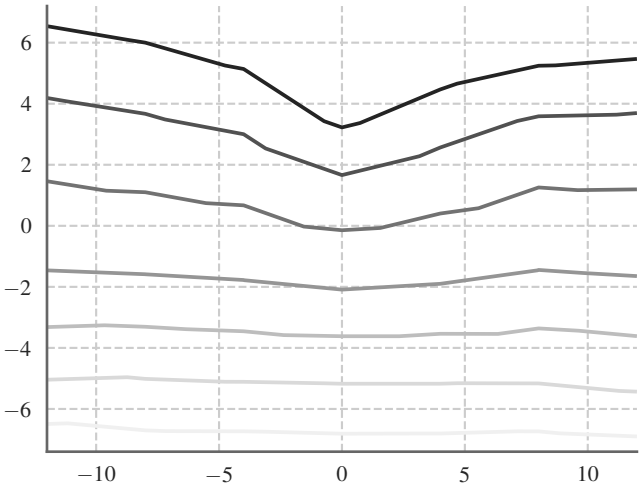

Figura 19: Exemplo de curvas de nível

7.2 Escala de Cores

Por padrão, o `tricontour` usa uma escala de tons de cinza para representar a variável dependente, no nosso caso o potencial, sendo branco o menor valor e preto, o maior. Isso pode ser facilmente alterado com o argumento `cmap`, que pode receber um objeto `Colormap` personalizado, mas em geral os mapas de cores do `Matplotlib` já são o bastante e só precisam ser passados pelos seu nomes ou nome de sua versão invertida, com `'_r'` ao final. Isso pode ser visto no código **??** com as cores usadas aqui, que são da escala `'winter'` invertida.

Código 7.2: Curvas de nível com escala de cores

```
# curvas de nível
plt.tricontour(x, y, V, cmap='winter_r')
# barra de cores
plt.colorbar()
```

Agora, para que as cores façam sentido, é preciso de uma legenda para elas, que normalmente são usadas com barras de cores na lateral. A função do `pyplot` para isso é a `colorbar`, como no código **??**.

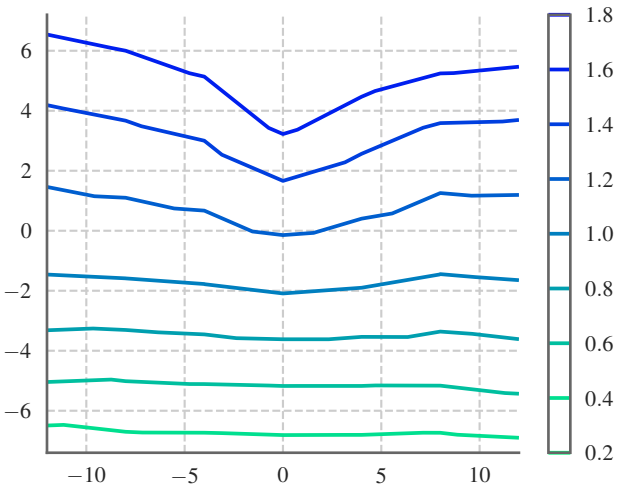

Figura 20: Exemplo de escala de cores

7.3 Escolha dos Níveis

É possível escolher quais os níveis onde serão feitas as curvas com um quarto argumento posicional. Esse argumento deve ser uma lista com as alturas ou, nesse caso, as equipotenciais que serão desenhadas. No exemplo **??**, essa lista está na variável `niveis`.

Código 7.3: Curvas desenhadas em níveis escolhidos

```
# niveis a serem desenhados
niveis = [0.34, 0.66, 1.00, 1.33, 1.65]

# curvas de nível
plt.tricontour(x, y, V, niveis, cmap='winter_r')
# barra de cores
plt.colorbar()
```

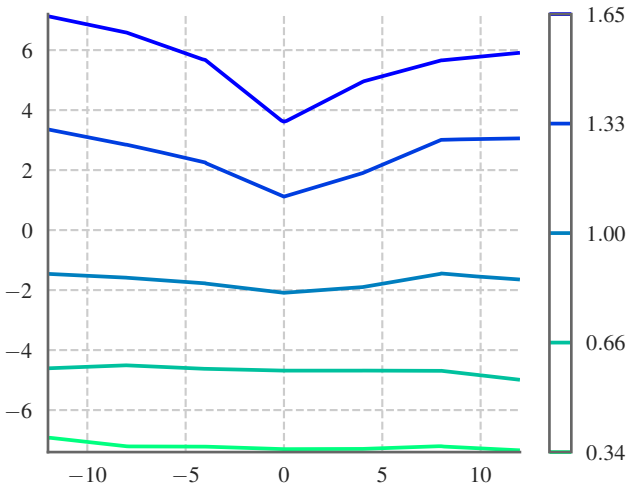

Figura 21: Exemplo das curvas em níveis específicos

Devido às imperfeições da técnica de triangulação, é possível que a escolha manual dos níveis gere alguns problemas no desenho, em especial nas extremidades da figura. Por exemplo, com `niveis = [0.33, 0.66, 1.00, 1.33, 1.66]`, o código **??** gera a curva de maior potencial com várias semi-retas sem sentido e a curva de menor potencial termina antes das outras.

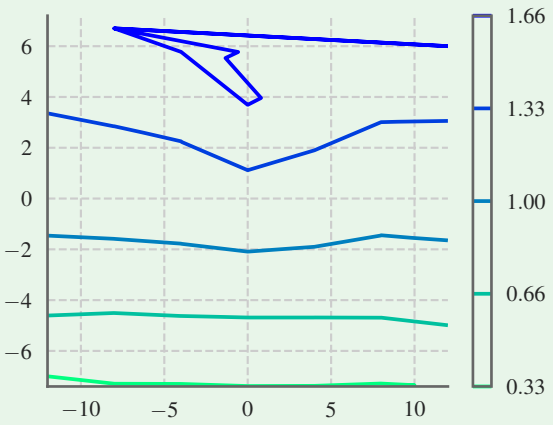

Figura 22: Exemplo de níveis com problemas

■

7.4 Formatação da Escala de Cores

Para colocar os textos do gráfico, como títulos e rótulos, o procedimento é como nos casos anteriores, com as funções `title`, `xlabel` e `ylabel`. O único problema aqui é o com a escala de cores, que, até mesmo pelo modo como funciona, não é possível de se fazer uma função diretamente no `pyplot` para descrevê-la. No entanto, as funções do `pyplot` , em geral, retornam o objeto que foi criado, então a função `colorbar` retorna o objeto `Colorbar` que foi criado. Esse objeto tem um método próprio para mudar sua descrição, o `set_label`, que é o que pode ser visto no código **??**.

Código 7.4: Exemplo de formatação da escala de cores e das curvas de nível

```
# -- depois de separar os dados -- #

# niveis a serem desenhados
niveis = [0.34, 0.66, 1.00, 1.33, 1.65]

# curvas de nível e escala de cores
plt.tricontour(x, y, V, niveis, cmap='winter_r', extend='both')
escala_de_cores = plt.colorbar()

# textos
escala_de_cores.set_label('Tensão [$V$]')
plt.xlabel('Posição X [$cm$]')
plt.ylabel('Posição Y [$cm$]')
plt.title('Potencial na Cuba com uma Ponta em uma das Placas')

# -- salva a figura -- #
```

Além disso, podemos ver nas escalas de cores anteriores, como da seção **??**, que os níveis mais extremos não aparecem muito bem. Para resolver isso, podemos extender um pouco a escala para mostrar valores além dos limites de potencial do gráfico. O argumento para isso é `extend='both'` em `tricontour`, que faz extensão em ambos os extremos. Outro argumento poderia ser `extendrect=True`, que faz a extensão ser retangular, em vez de triangular.

7.5 Resultado

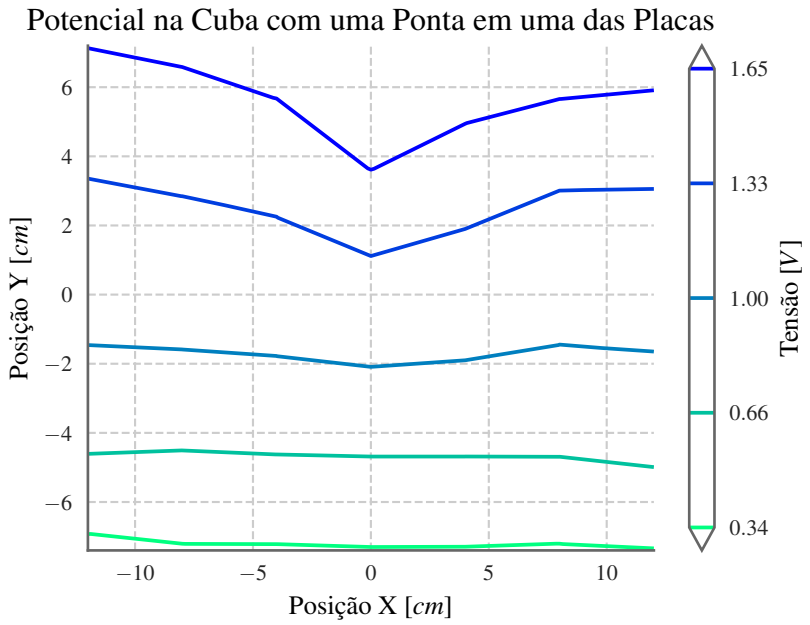

Figura 23: Exemplo completo de curvas de nível