

See discussions, stats, and author profiles for this publication at: <https://www.researchgate.net/publication/353391359>

Programação Científica com Python

Technical Report · July 2021

DOI: 10.13140/RG.2.2.10270.66888

CITATIONS

0

READS

9,718

1 author:



Alexandre L M Levada

Federal University of São Carlos

159 PUBLICATIONS 494 CITATIONS

SEE PROFILE



Departamento de Computação
Centro de Ciências Exatas e Tecnologia
Universidade Federal de São Carlos

Programação Científica com Python

Apostila sobre programação científica e aplicações

Prof. Alexandre Luis Magalhães Levada
Email: alexandre.levada@ufscar.br

Sumário

A linguagem Python.....	3
Métodos numéricos.....	5
O Método de Newton.....	5
Método da secante.....	8
Integração numérica: A regra de Simpson.....	14
Mínimos quadrados e regressão linear.....	17
Coeficiente de determinação.....	20
Listas, vetores e matrizes.....	26
Plotagem e visualização de dados.....	36
Recorrência logística e sistemas caóticos.....	42
Sistemas Lineares.....	49
O método da eliminação de Gauss.....	50
Algoritmos iterativos.....	54
O método de Jacobi.....	54
O método de Gauss-Seidel.....	56
Métodos iterativos: convergência.....	57
Decomposição em autovalores e autovetores (Eigendecomposition).....	58
Convolução e filtragem linear.....	59
Autômatos celulares.....	79
Filtragem de mínimos quadrados.....	87
Autovalores e autovetores.....	91
Análise de Componentes Principais (PCA).....	95
PCA pela Maximização da Variância.....	95
PCA pela Minimização do erro quadrático médio.....	101
O Perceptron.....	104
Bibliografia.....	116
Sobre o autor.....	116

“Antes de pensar em desistir, lembre-se: a última parte de uma árvore a crescer são os frutos.”
(Anônimo)

A linguagem Python

“Python é uma linguagem de programação de alto nível, interpretada, de script, imperativa, orientada a objetos, funcional, de tipagem dinâmica e forte. Foi lançada por Guido van Rossum em 1991.[1] Atualmente possui um modelo de desenvolvimento comunitário, aberto e gerenciado pela organização sem fins lucrativos Python Software Foundation. Apesar de várias partes da linguagem possuírem padrões e especificações formais, a linguagem como um todo não é formalmente especificada. A linguagem foi projetada com a filosofia de enfatizar a importância do esforço do programador sobre o esforço computacional. Prioriza a legibilidade do código sobre a velocidade ou expressividade. Combina uma sintaxe concisa e clara com os recursos poderosos de sua biblioteca padrão e por módulos e frameworks desenvolvidos por terceiros. Python é uma linguagem de propósito geral de alto nível, multiparadigma, suporta o paradigma orientado a objetos, imperativo, funcional e procedural. Possui tipagem dinâmica e uma de suas principais características é permitir a fácil leitura do código e exigir poucas linhas de código se comparado ao mesmo programa em outras linguagens”. (Wikipedia)

Nesse curso, optamos pela linguagem Python, principalmente pela questão didática, uma vez que a sua curva de aprendizado é bem mais suave do que linguagens de programação como C, C++ e Java. Existem basicamente duas versões de Python coexistindo atualmente. O Python 2 e o Python 3. Apesar de existirem poucas diferenças entre elas, é o suficiente para que um programa escrito em Python 2 possa não ser compreendido por interpretador do Python 3. Optamos aqui pelo Python 3, pois além de ser considerado uma evolução natural do Python 2, representa o futuro da linguagem.

Porque Python 3?

- Evolução do Python 2 (mais moderno)
- Sintaxe simples e de fácil aprendizagem
- Linguagem de propósito geral que mais cresce na atualidade
- Bibliotecas para programação científica e aprendizado de máquina

Scikit_learn Scikit_image NetworkX,...
Scipy Matplotlib Statsmodels Pandas, Ipython,...
Numpy
Python Standard Library

Dentre as principais vantagens de se aprender Python, podemos citar a enorme gama de bibliotecas existentes para a linguagem. Isso faz com que Python seja extremamente versátil. É possível desenvolver aplicações científicas como métodos matemáticos numéricos, processamento de sinais e imagens, aprendizado de máquina até aplicações mais comerciais, como sistemas web com acesso a bancos de dados.

Plataforma Python

Para desenvolver aplicações científicas em Python, é conveniente instalar um ambiente de programação em que as principais bibliotecas para computação científica estejam presentes. Isso poupa-nos muito tempo e esforço, pois além de não precisarmos procurar cada biblioteca individualmente, não precisamos saber a relação de dependência entre elas (quais devem ser instaladas primeiro e quais tem que ser instaladas posteriormente). Dentre as plataformas Python para computação científica, recomendamos a seguinte:

Anaconda - <https://www.anaconda.com/products/individual>

Uma ferramenta multiplataforma com versões para Windows, Linux e MacOS. Inclui mais de uma centena de pacotes para programação científica, o que o torna um ambiente completo para o desenvolvimento de aplicações em Python. Inclui diversos IDE's, como o idle, ipython e spyder.

Repl.it

Uma opção muito interessante é o interpretador Python na nuvem repl.it

Você pode desenvolver e armazenar seus códigos de maneira totalmente online sem a necessidade de instalar em sua máquina um ambiente de desenvolvimento local.

Nossa recomendação é a plataforma Anaconda, por ser disponível em todos os sistemas operacionais. Ao realizar o download, opte pelo Python 3, que atualmente deve estar na versão 3.7. Para a execução das atividades presentes nessa apostila, o editor IDLE é recomendado. Além de ser um ambiente extremamente simples e compacto, ele é muito leve, o que torna sua execução possível mesmo em máquinas com limitações de hardware, como pouca memória RAM e processador lento.

Após concluir a instalação, basta digitar anaconda na barra de busca do Windows. A opção Anaconda Prompt deve ser selecionada. Ela nos leva a um terminal onde ao digitar o comando idle e pressionarmos enter, seremos diretamente redirecionados ao ambiente IDLE. Um vídeo tutorial mostrando esse processo pode ser assistido no link a seguir:

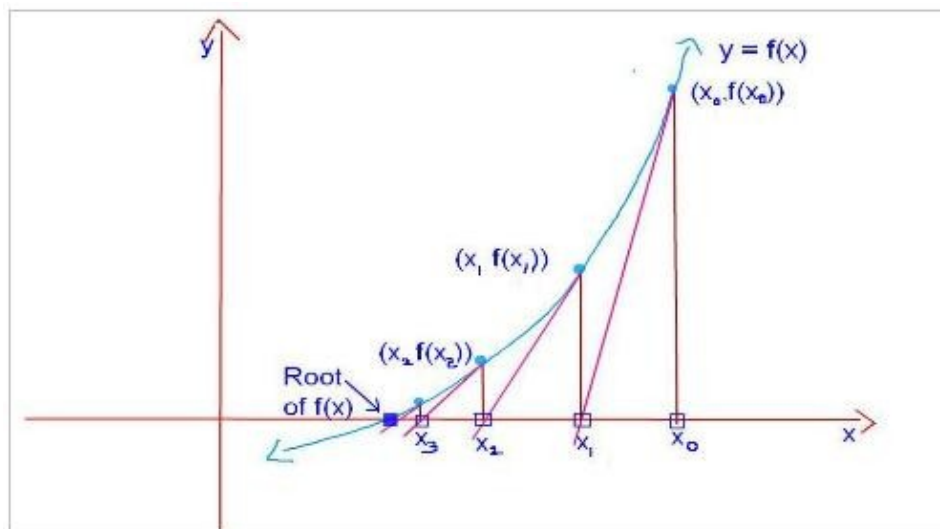
<https://www.youtube.com/watch?v=PWdrdWDmJIY&t=8s>

“Não trilhe apenas os caminhos já abertos. Por serem conhecidos eles nos levam somente até onde alguém já foi um dia.” (Alexander Graham Bell)

Métodos numéricos

O Método de Newton

Um problema de grande importância na matemática consiste em encontrar as raízes de uma função $f(x)$, quando elas existem. A ideia básica do método consiste em, a partir de uma escolha inicial x_0 relativamente próxima a verdadeira raiz, aproximar a função pela reta tangente a $(x_0, f(x_0))$ e então computar o ponto em que essa reta intercepta o eixo x , que tipicamente será uma melhor aproximação a raiz da função.



A equação da reta que passa pelo ponto $(x_0, f(x_0))$ e é tangente a curva nesse ponto tem inclinação igual a $m = f'(x_0)$ (coeficiente angular é a derivada). Dessa forma, temos a seguinte equação:

$$y - y_0 = m(x - x_0)$$

Substituindo os valores, temos:

$$y = f(x_0) + f'(x_0)(x - x_0)$$

Como queremos o ponto x em que a reta intercepta o eixo x , então $y = 0$:

$$f(x_0) + f'(x_0)(x - x_0) = 0$$

Dividindo ambos os lados por $f'(x_0)$:

$$\frac{f(x_0)}{f'(x_0)} + x - x_0 = 0$$

o que implica em

$$x = x_0 - \frac{f(x_0)}{f'(x_0)}$$

Como o processo é iterativo (deve ser repetido várias vezes), chega-se na seguinte relação de recorrência:

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

onde x_0 pode ser escolhido arbitrariamente (idealmente próximo da raiz).

Aplicações: cálculo da raiz quadrada de um número.

Suponha a equação $f(x) = x^2 - a = 0$. Assim, a derivada vale $f'(x) = 2x$ e temos:

$$x_{k+1} = x_k - \frac{x_k^2 - a}{2x_k} \rightarrow x_{k+1} = x_k - \frac{x_k}{2} + \frac{a}{2x_k} \rightarrow x_{k+1} = \frac{1}{2} \left(x_k + \frac{a}{x_k} \right)$$

Método de Newton: convergência

Iremos iniciar definindo a série de Taylor como uma ferramenta matemática para aproximar uma função $f(x)$ por um polinômio em que a i -ésima derivada do polinômio é igual a i -ésima derivada de $f(x)$. Seja $f(x)$ uma função real, infinitamente diferenciável definida em um ponto a . Então a expansão em série de Taylor de $f(x)$ é dada por:

$$f(x) = f(a) + f'(a)(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \frac{f'''(a)}{3!}(x-a)^3 + \dots$$

O objetivo aqui consiste em obter uma aproximação de segunda ordem para $f(x_n)$ assumindo a existência de uma raiz α nas proximidades de x_n .

$$f(\alpha) = f(x_n) + f'(x_n)(\alpha - x_n) + \frac{1}{2}f''(x_n)(\alpha - x_n)^2 = 0$$

o que nos leva a

$$f(x_n) + f'(x_n)(\alpha - x_n) = -\frac{1}{2}f''(x_n)(\alpha - x_n)^2$$

Dividindo ambos os lados por $f'(x_n)$ temos:

$$\frac{f(x_n)}{f'(x_n)} + \alpha - x_n = -\frac{1}{2} \frac{f''(x_n)}{f'(x_n)} (\alpha - x_n)^2$$

Do método de Newton sabemos que:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

o que nos leva a:

$$\alpha - x_{n+1} = -\frac{f''(x_n)}{2f'(x_n)}(\alpha - x_n)^2$$

Como os erros na iteração n e $n+1$ são dados por $(\alpha - x_n) = \varepsilon_n$ e $(\alpha - x_{n+1}) = \varepsilon_{n+1}$ e tomando o módulo (pois não importa se erro é para mais ou para menos):

$$|\varepsilon_{n+1}| = \frac{|f''(x_n)|}{2|f'(x_n)|} \varepsilon_n^2$$

o que mostra que o erro no passo $n+1$ depende do quadrado do erro em n . Portanto, temos que a taxa de convergência é quadrática (significa que método é rápido).

Obs: Para que na prática observemos uma taxa de convergência quadrática, são necessários 3 condições: a) $f'(x) \neq 0$, b) $f''(x)$ é contínua e c) x_0 suficientemente próximo de α

Aplicações: cálculo da raiz quadrada de um número.

Suponha a equação $f(x) = x^2 - a = 0$. Assim, a derivada vale $f'(x) = 2x$ e temos:

$$x_{k+1} = x_k - \frac{x_k^2 - a}{2x_k} \rightarrow x_{k+1} = x_k - \frac{x_k}{2} + \frac{a}{2x_k} \rightarrow x_{k+1} = \frac{1}{2} \left(x_k + \frac{a}{x_k} \right)$$

A solução desse problema foi apresentada em capítulos anteriores.

Ex50: Faça um programa que compute as raízes das funções a seguir utilizando o método de Newton. Utilize como critério de parada o erro absoluto entre duas estimativas: $|x_k - x_{k+1}|$. Se essa diferença for muito pequena, ou seja, menor que 10^{-7} , o método para de executar.

- a) $f(x) = x^3 + x - 1$
- b) $f(x) = x - \cos(x)$
- c) $f(x) = e^x - 2\cos(x)$
- d) $f(x) = x - 2\sin(x)$

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

# define função f(x)
def f(x):
    return x**3 + x - 1
    #return x - np.cos(x)
    #return np.exp(x) - 2*np.cos(x)
    #return x - 2*np.sin(x)

# define a derivada de f(x)
def df(x):
    return 3*x**2 + 1
    #return 1 + np.sin(x)
    #return np.exp(x) + 2*np.sin(x)
    #return 1 - 2*np.cos(x)
```

```

# método de Newton
# Parâmetros:
#   x - chute inicial
#   epsilon - tolerância para erro (quanto menor, melhor)
def newton(x, epsilon):
    novo_x = np.random.random()
    erro = abs(x - novo_x)

    while erro >= epsilon:
        novo_x = x - f(x)/df(x)
        erro = abs(x - novo_x)
        print('x : %.10f ***** Erro: %.10f' %(novo_x, erro))
        x = novo_x
    return x

# Início do script
x0 = float(input('Entre com o chute inicial (x0): '))
print('Raiz da função: %f' %newton(x0, 0.0000001))

# Plota gráfico da função
eixo_x = np.linspace(-1, 1, 1000) # intervalo de plotagem
eixo_y = f(eixo_x)
plt.figure(1)
plt.plot(eixo_x, eixo_y)
plt.show()

```

Limitações método de Newton

a) Bases de atração: pequenas variações em x_0 podem levar a grandes variações na solução

$$f(x) = x^3 - 2x^2 - 11x + 12$$

Verifique o que acontece se

$$x_0 = 2.35287527$$

$$x_0 = 2.35284172$$

$$x_0 = 2.352836323$$

b) Chute inicial estacionário (derivada é nula)

$$f(x) = 1 - x^2 \quad \text{com} \quad x_0 = 0$$

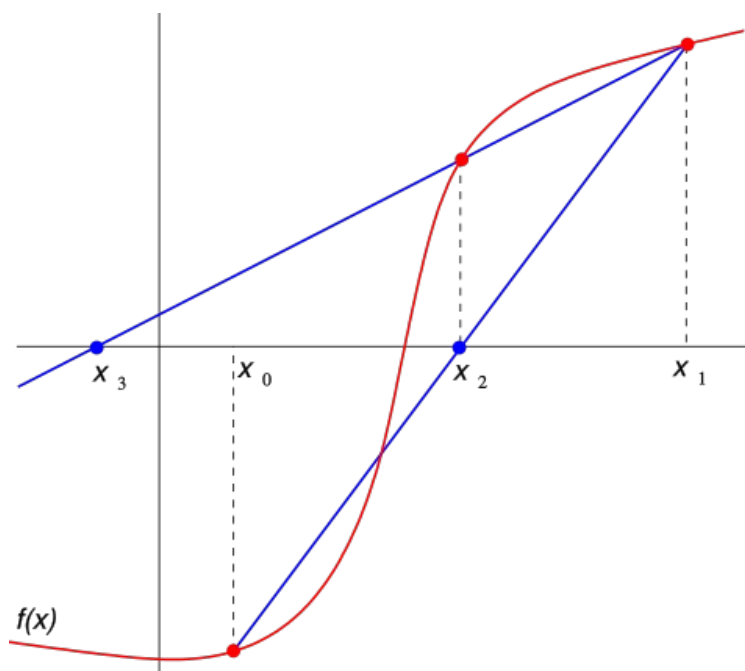
c) Iterações circulares (chute inicial gera loop infinito)

$$f(x) = x^3 - 2x + 2 \quad \text{com} \quad x_0 = 0$$

Método da secante

Um problema com o método de Newton é que ele depende explicitamente da derivada da função $f(x)$. Em alguns casos, ela pode ser difícil ou até mesmo impossível de calcular. Um outro método para encontrar raízes de funções que não requer derivadas é o método das secantes. Esse método

pode ser pensado como uma aproximação do método de Newton utilizando a técnica de diferenças finitas para o computo numérico das derivadas.



Iniciando pelos pontos x_0 e x_1 é possível construir uma linha entre $(x_0, f(x_0))$ e $(x_1, f(x_1))$, como indicado na figura acima. A equação dessa reta é dada por:

$$y - f(x_1) = m(x - x_1)$$

onde $m = \frac{f(x_1) - f(x_0)}{x_1 - x_0}$ (inclinação da reta)

Assim, temos

$$y = \frac{f(x_1) - f(x_0)}{x_1 - x_0}(x - x_1) + f(x_1)$$

Para encontrar o ponto em que essa reta intercepta o eixo x , basta atribuir valor zero a y :

$$\frac{f(x_1) - f(x_0)}{x_1 - x_0}(x - x_1) + f(x_1) = 0$$

Multiplicando ambos os lados por $\frac{x_1 - x_0}{f(x_1) - f(x_0)}$ temos:

$$x - x_1 + f(x_1) \frac{x_1 - x_0}{f(x_1) - f(x_0)} = 0$$

Isolar x nos leva a:

$$x = x_1 - f(x_1) \frac{x_1 - x_0}{f(x_1) - f(x_0)}$$

Como o processo é iterativo e deve ser repetido por várias vezes, chega-se a seguinte relação de recorrência:

$$x_k = x_{k-1} - f(x_{k-1}) \frac{x_{k-1} - x_{k-2}}{f(x_{k-1}) - f(x_{k-2})}$$

Método da secante: convergência

Seja α uma raiz de $f(x)$. Então, $f(\alpha) = 0$. Sem perda de generalidade, consideraremos a iteração dada por:

$$x_{n+1} = x_n - \left(\frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})} \right) f(x_n) \quad (*)$$

Definimos o erro na iteração n como $\varepsilon_n = x_n - \alpha$. Então, no passo $n+1$, temos $\varepsilon_{n+1} = x_{n+1} - \alpha$. Aplicando a recorrência (*) a x_{n+1} :

$$\varepsilon_{n+1} = x_n - \left(\frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})} \right) f(x_n) - \alpha$$

Agrupando termos:

$$\varepsilon_{n+1} = \varepsilon_n - f(x_n) \left(\frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})} \right)$$

Somando e subtraindo α do numerador temos:

$$\varepsilon_{n+1} = \varepsilon_n - f(x_n) \left(\frac{x_n - \alpha - (x_{n-1} - \alpha)}{f(x_n) - f(x_{n-1})} \right)$$

o que nos leva a:

$$\varepsilon_{n+1} = \varepsilon_n - f(x_n) \left(\frac{\varepsilon_n - \varepsilon_{n-1}}{f(x_n) - f(x_{n-1})} \right)$$

Novamente, agrupando alguns termos temos:

$$\varepsilon_{n+1} = \frac{\varepsilon_n (f(x_n) - f(x_{n-1})) - f(x_n) (\varepsilon_n - \varepsilon_{n-1})}{f(x_n) - f(x_{n-1})}$$

o que resulta em:

$$\varepsilon_{n+1} = \frac{\varepsilon_n f(x_n) - \varepsilon_n f(x_{n-1}) - \varepsilon_n f(x_n) + \varepsilon_{n-1} f(x_n)}{f(x_n) - f(x_{n-1})}$$

e simplifica-se para:

$$\varepsilon_{n+1} = \frac{\varepsilon_{n-1}f(x_n) - \varepsilon_n f(x_{n-1})}{f(x_n) - f(x_{n-1})}$$

A seguir iremos multiplicar e dividir o lado direito por $x_n - x_{n-1}$:

$$\varepsilon_{n+1} = \left[\frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})} \right] \left[\frac{\varepsilon_{n-1}f(x_n) - \varepsilon_n f(x_{n-1})}{x_n - x_{n-1}} \right]$$

E iremos colocar em evidência o produto $\varepsilon_n \varepsilon_{n-1}$ no segundo fator para chegar em:

$$\varepsilon_{n+1} = \left[\frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})} \right] \left[\frac{\frac{f(x_n)}{\varepsilon_n} - \frac{f(x_{n-1})}{\varepsilon_{n-1}}}{x_n - x_{n-1}} \right] \varepsilon_n \varepsilon_{n-1} \quad (**)$$

Utilizando uma série de Taylor de segunda ordem para aproximar $f(x_n)$ ao redor de α temos:

$$f(x_n) = f(\alpha) + f'(\alpha)(x_n - \alpha) + \frac{f''(\alpha)}{2}(x_n - \alpha)^2 = f'(\alpha)\varepsilon_n + \frac{f''(\alpha)}{2}\varepsilon_n^2$$

Assim,

$$\frac{f(x_n)}{\varepsilon_n} = f'(\alpha) + \frac{1}{2}f''(\alpha)\varepsilon_n \quad \text{e} \quad \frac{f(x_{n-1})}{\varepsilon_{n-1}} = f'(\alpha) + \frac{1}{2}f''(\alpha)\varepsilon_{n-1}$$

E então a diferença fica:

$$\frac{f(x_n)}{\varepsilon_n} - \frac{f(x_{n-1})}{\varepsilon_{n-1}} = \left[f'(\alpha) + \frac{1}{2}f''(\alpha)\varepsilon_n \right] - \left[f'(\alpha) + \frac{1}{2}f''(\alpha)\varepsilon_{n-1} \right] = \frac{1}{2}f''(\alpha)(\varepsilon_n - \varepsilon_{n-1})$$

Voltando a (**) temos:

$$\varepsilon_{n+1} \approx \left[\frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})} \right] \left[\frac{\frac{1}{2}f''(\alpha)(\varepsilon_n - \varepsilon_{n-1})}{x_n - x_{n-1}} \right] \varepsilon_n \varepsilon_{n-1}$$

Mas sabemos que

$$\varepsilon_n - \varepsilon_{n-1} = (x_n - \alpha) - (x_{n-1} - \alpha) = x_n - x_{n-1}$$

e assim

$$\varepsilon_{n+1} \approx \left[\frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})} \right] \left[\frac{1}{2}f''(\alpha) \right] \varepsilon_n \varepsilon_{n-1}$$

Para x_n e x_{n-1} suficientemente próximos de α sabemos que por diferenças finitas temos:

$$f'(\alpha) \approx \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}$$

o que nos leva finalmente a:

$$\varepsilon_{n+1} \approx \frac{1}{f'(\alpha)} \frac{1}{2} f''(\alpha) \varepsilon_n \varepsilon_{n-1} = \frac{f''(\alpha)}{2f'(\alpha)} \varepsilon_n \varepsilon_{n-1} = C \varepsilon_n \varepsilon_{n-1} \quad (***)$$

com C denotando uma constante.

Para determinar a ordem de convergência do método temos que descobrir o valor do expoente p a partir da expressão a seguir:

$$|\varepsilon_{n+1}| \approx A |\varepsilon_n|^p \quad (****)$$

É intuitivo perceber que de (****) também vale:

$$|\varepsilon_n| \approx A |\varepsilon_{n-1}|^p$$

que implica em

$$|\varepsilon_{n-1}|^p = A^{-1} |\varepsilon_n|$$

e consequentemente

$$|\varepsilon_{n-1}| = (A^{-1} |\varepsilon_n|)^{\frac{1}{p}}$$

Para ficar consistente com a equação (**) é preciso ter:

$$|\varepsilon_{n+1}| = A |\varepsilon_n|^p = C |\varepsilon_n| |\varepsilon_{n-1}| = C |\varepsilon_n| (A^{-1} |\varepsilon_n|)^{\frac{1}{p}}$$

ou seja,

$$A |\varepsilon_n|^p = C A^{-\frac{1}{p}} |\varepsilon_n|^{1+\frac{1}{p}}$$

Isolando as constantes no lado esquerdo, temos:

$$\frac{A}{A^{-\frac{1}{p}} C} |\varepsilon_n|^p = |\varepsilon_n|^{1+\frac{1}{p}}$$

que é igual a

$$\frac{A^{1-\frac{1}{p}}}{C} |\varepsilon_n|^p = |\varepsilon_n|^{1+\frac{1}{p}}$$

e finalmente chegamos a

$$\frac{A^{1-\frac{1}{p}}}{C} = \frac{|\varepsilon_n|^{1+\frac{1}{p}}}{|\varepsilon_n|^p} = |\varepsilon_n|^{1-p+\frac{1}{p}}$$

Como o lado esquerdo é uma constante, o lado direito da igualdade também tem que ser constante quando $n \rightarrow \infty$. Para isso, o expoente deve ser nulo, ou seja, devemos ter:

$$1 - p + \frac{1}{p} = 0$$

que implica na equação do segundo grau a seguir:

$$p^2 - p - 1 = 0$$

cujas soluções positivas são dadas por

$$p = \frac{1 + \sqrt{5}}{2} \approx 1.618$$

Ou seja, o método não é tão lento quanto um linear mas também não é tão rápido quanto um quadrático. Dizemos que o método da secante tem convergência superlinear.

Ex51: Faça um programa que compute as raízes das mesmas funções do exercício anterior utilizando o método da secante. Utilize como critério de parada o erro absoluto entre duas estimativas: $|x_k - x_{k-1}|$. Se essa diferença for muito pequena, ou seja, menor que 10^{-7} , o método para de executar.

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

# define função f(x)
def f(x):
    return x**3 + x - 1
    #return x - np.cos(x)
    #return np.exp(x) - 2*np.cos(x)
    #return x - 2*np.sin(x)
    #return x**100

# método da secante
# Parâmetros:
# x0, x1 - chutes iniciais
# epsilon - tolerância para erro (quanto menor, melhor)
def secante(x0, x1, epsilon):
    erro = abs(x0 - x1)
    while erro >= epsilon:
        novo_x = x1 - f(x1)*(x1 - x0)/(f(x1) - f(x0))
        erro = abs(x1 - novo_x)
        print('x : %.10f ***** Erro: %.10f' %(novo_x, erro))
        x0, x1 = x1, novo_x

    return novo_x
```

```
# Início do script
x0 = float(input('Entre com o primeiro chute inicial (x0): '))
x1 = float(input('Entre com o segundo chute inicial (x1): '))

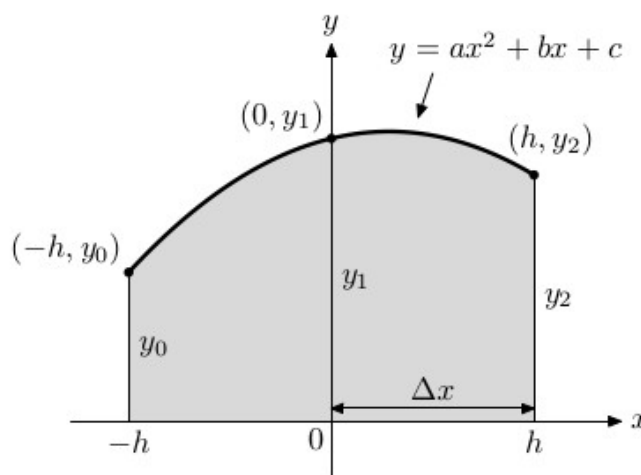
print('Raiz da função: %f' %secante(x0, x1, 0.0000001))

# Plota gráfico da função
eixo_x = np.linspace(-1, 1, 1000) # intervalo de plotagem
eixo_y = f(eixo_x)
plt.figure(1)
plt.plot(eixo_x, eixo_y)
plt.show()
```

Para interessados em métodos numéricos, existem outros algoritmos para encontrar raízes de equações, como o método da bisseção e o método da posição falsa. Não iremos discuti-los neste curso, mas nem por isso são menos importantes.

Integração numérica: A regra de Simpson

A regra de Simpson é um método numérico que aproxima o valor de uma integral definida através de polinômios quadráticos (parábolas). É muito utilizado como uma forma computacional de se calcular a área sob uma curva. Primeiro, iremos derivar uma fórmula para calcular a área sob uma parábola definida pela equação $y = ax^2 + bx + c$ passando por 3 pontos: $(-h, y_0)$, $(0, y_1)$ e (h, y_2) , conforme ilustra a figura a seguir.



A área sob a curva nada mais é que a integral definida da função $y = f(x)$ de $-h$ a h :

$$\begin{aligned}
 A &= \int_{-h}^h (ax^2 + bx + c) \, dx \\
 &= \left(\frac{ax^3}{3} + \frac{bx^2}{2} + cx \right) \Big|_{-h}^h \\
 &= \frac{2ah^3}{3} + 2ch \\
 &= \frac{h}{3} (2ah^2 + 6c)
 \end{aligned}$$

Como os 3 pontos $(-h, y_0)$, $(0, y_1)$ e (h, y_2) pertencem a parábola, eles satisfazem a equação $y = ax^2 + bx + c$ e portanto:

$$y_0 = ah^2 - bh + c$$

$$y_1 = c$$

$$y_2 = ah^2 + bh + c$$

Note porém que a seguinte igualdade é válida:

$$y_0 + 4y_1 + y_2 = (ah^2 - bh + c) + 4c + (ah^2 + bh + c) = 2ah^2 + 6c.$$

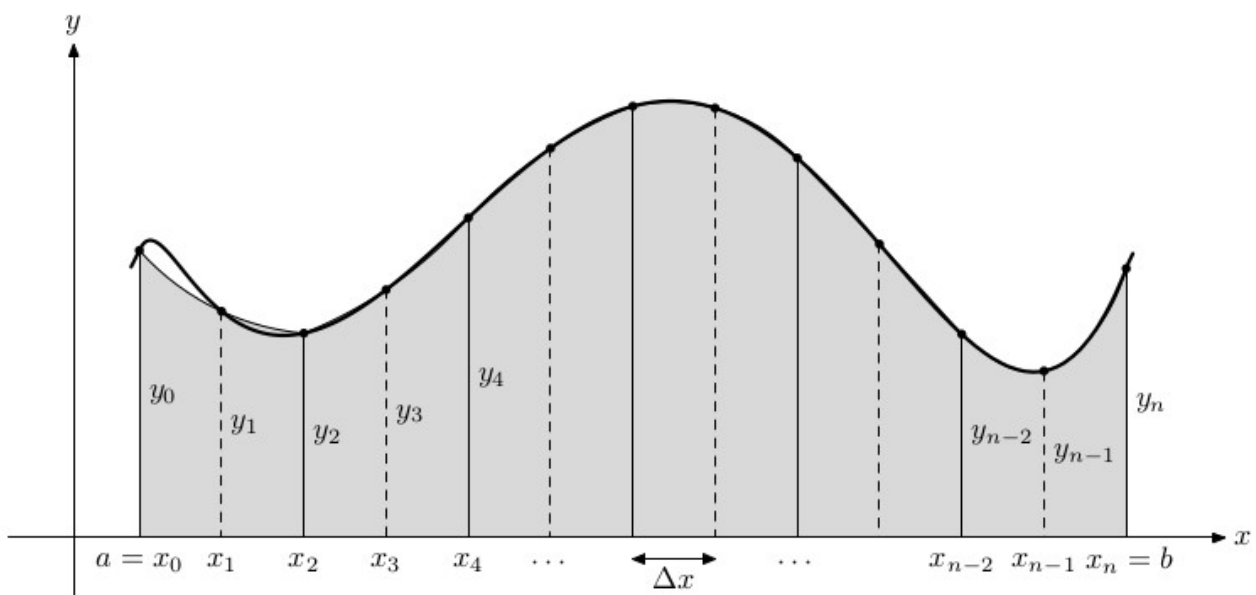
Assim, a área sob a parábola pode ser reescrita como:

$$A = \frac{h}{3} (y_0 + 4y_1 + y_2) = \frac{\Delta x}{3} (y_0 + 4y_1 + y_2)$$

Para aplicar a regra de Simpson para a integração numérica de uma função $f(x)$ qualquer, deseja-se resolver a seguinte integral: $\int_a^b f(x)$. Assumindo $f(x)$ contínua no intervalo $[a, b]$ e dividindo o intervalo em um número par n de subintervalos de tamanhos iguais a $\Delta x = \frac{b-a}{n}$, definimos $n+1$ pontos, para os quais podemos computar os valores da função $f(x)$:

$$x_0 = a, \quad x_1 = a + \Delta x, \quad x_2 = a + 2\Delta x, \quad \dots, \quad x_n = a + n\Delta x = b.$$

$$y_0 = f(x_0), \quad y_1 = f(x_1), \quad y_2 = f(x_2), \quad \dots, \quad y_n = f(x_n).$$



Podemos estimar a integral pela soma das áreas sob os arcos parabólicos formados por cada 3 pontos sucessivos:

$$\int_a^b f(x) dx \approx \frac{\Delta x}{3} (y_0 + 4y_1 + y_2) + \frac{\Delta x}{3} (y_2 + 4y_3 + y_4) + \cdots + \frac{\Delta x}{3} (y_{n-2} + 4y_{n-1} + y_n)$$

Simplificando a expressão anterior, chega-se a:

$$\int_a^b f(x) dx \approx \frac{\Delta x}{3} (y_0 + 4y_1 + 2y_2 + 4y_3 + 2y_4 + \cdots + 4y_{n-1} + y_n)$$

Exercício: Utilizando uma calculadora, aplique a regra de Simpson com $n = 6$ para aproximar a integral

$$\int_1^4 \sqrt{1+x^3} dx$$

Para $n = 6$, temos subintervalos de largura $\Delta x = \frac{b-a}{n} = \frac{4-1}{6} = 0.5$. Assim, os pontos ficam:

x	1	1.5	2	2.5	3	3.5	4
$y = \sqrt{1+x^3}$	$\sqrt{2}$	$\sqrt{4.375}$	3	$\sqrt{16.625}$	$\sqrt{28}$	$\sqrt{43.875}$	$\sqrt{65}$

Portanto, o valor final da integral pode ser computado por:

$$\int_1^4 \sqrt{1+x^3} dx \approx \frac{0.5}{3} \left(\sqrt{2} + 4\sqrt{4.375} + 2(3) + 4\sqrt{16.625} + 2\sqrt{28} + 4\sqrt{43.875} + \sqrt{65} \right) \approx \boxed{12.871}$$

Veremos a seguir um script em Python que implementa a regra de Simpson.

```
import numpy as np

def simpson(f, a, b, n):
    h = (b - a)/n
    soma_pares, soma_impares = 0, 0
    # Soma os pares
    for i in range(2, n, 2):
        k = a + i*h
        soma_pares = soma_pares + f(k)
    # Soma os ímpares
    for i in range(1, n, 2):
        k = a + i*h
        soma_impares = soma_impares + f(k)
    area = (h/3)*(f(a) + 4*soma_impares + 2*soma_pares + f(b))
    return area
```

```
# Define funções (forma alternativa: lambda functions)
f = lambda x: np.sqrt((1 + x**3))           # de 1 a 4
g = lambda x: 1/np.sqrt((1 + x**4))         # de 0 a 2
p = lambda x: (1/(2*np.pi)**0.5)*np.exp(-0.5*x**2) # de 0 a 5

a = float(input('Entre com o limite inferior (a): '))
b = float(input('Entre com o limite superior (b): '))
n = int(input('Entre com o número de subintervalos (n): '))

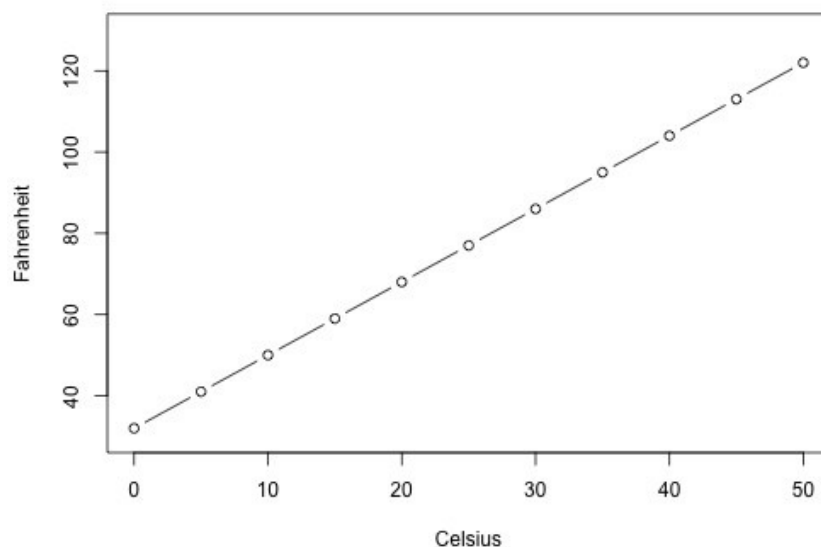
print('A área sob a curva vale %f' %simpson(f, a, b, n))
```

Mínimos quadrados e regressão linear

Mínimos quadrados ou regressão linear é um método estatístico que nos permite estudar e analisar relacionamentos entre duas variáveis aleatórias:

- uma variável, denotada por x, denominada de variável independente ou exploratória;
- outra variável, denotada por y, denominada de variável resposta ou dependente;

Antes de proceder, devemos esclarecer que tipos de relacionamentos não nos interessa estudar na regressão linear: relacionamentos determinísticos ou funcionais. A figura a seguir ilustra um exemplo de relacionamento determinístico: a relação entre temperaturas em Celsius e Farenheit.



Note que os pontos (x, y) observados caem diretamente sobre uma linha reta. Isso ocorre porque o relacionamento entre graus Celsius e Farenheit é dada por:

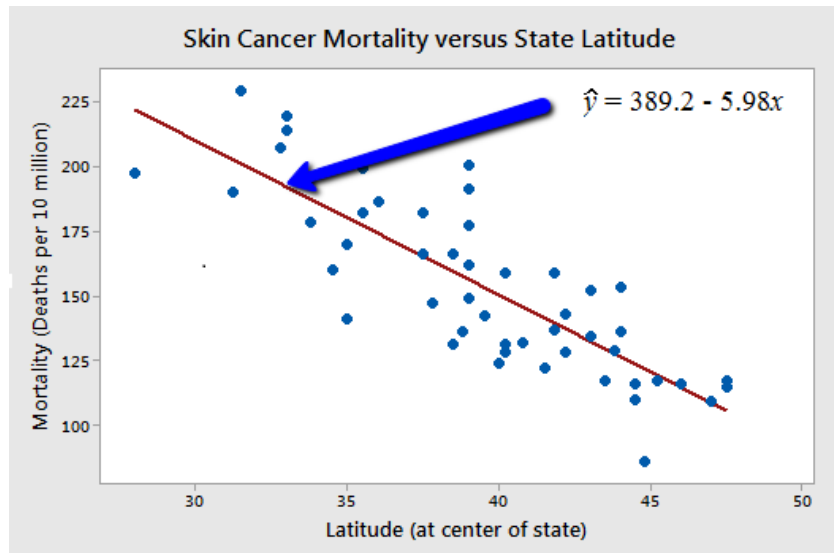
$$F = \frac{9}{5}C + 32$$

ou seja, se você conhece a temperatura em Celsius você pode usar essa equação para determinar a temperatura em Farenheit exatamente. Alguns outros exemplos de relações determinísticas incluem:

- circunferência = $2\pi r$
- Lei de Ohm: $U = Ri$
- velocidade = s / t

Para cada uma dessas relações determinísticas existe uma função que descreve exatamente o relacionamento entre duas variáveis. Regressão linear não está interessada em estudar esses tipos de relações, mas sim relações estatísticas (não-determinísticas).

O exemplo a seguir ilustra um relacionamento estatístico entre duas variáveis: a variável resposta y é a mortalidade devido ao câncer de pele (número de mortes para cada 10 milhões de pessoas) e a variável exploratória x é a latitude do centro de cada um dos 49 estados norte-americanos.



O gráfico sugere um relacionamento negativo entre a latitude e a mortalidade devido ao câncer de pele, mas o relacionamento não é perfeito, ou seja, o gráfico exhibe um comportamento aproximado, uma tendência, devido ao espalhamento e a incerteza presente nos dados. Alguns outros exemplos de relacionamentos estatísticos incluem:

- Altura e peso
- Álcool consumido e taxa de álcool no sangue
- Capacidade pulmonar e anos de fumo

Assim, o problema em questão consiste em, dado um conjunto de pontos observados (x_i, y_i) para $i=1, \dots, n$, estimar o melhor relacionamento linear possível entre as duas variáveis. Em outras palavras, desejamos encontrar a reta que melhor se ajusta aos dados. O grau de ajuste é definido em termos dos erros entre os verdadeiros valores de y_i e suas previsões lineares \hat{y}_i . O objetivo consiste em encontrar a reta

$$y = \alpha + \beta x$$

que minimiza a soma dos resíduos ao quadrado, dado por

$$Q(\alpha, \beta) = \sum_{i=1}^n e_i^2 = \sum_{i=1}^n (y_i - \hat{y}_i)^2 = \sum_{i=1}^n (y_i - \alpha - \beta x_i)^2$$

Derivando $Q(\alpha, \beta)$ em relação a α e igualando o resultado a zero, tem-se:

$$\frac{dQ(\alpha, \beta)}{d\alpha} = \sum_{i=1}^n (y_i - \alpha - \beta x_i) = 0$$

Aplicando a distributiva:

$$\sum_{i=1}^n y_i - n\alpha - \beta \sum_{i=1}^n x_i = 0$$

Dividindo tudo por n, temos:

$$\bar{y} - \alpha - \beta \bar{x} = 0$$

Isolando α chega-se a:

$$\alpha = \bar{y} - \beta \bar{x} \quad (I)$$

Da mesma forma, derivando $Q(\alpha, \beta)$ em relação a β nos leva a:

$$\frac{dQ(\alpha, \beta)}{d\beta} = \sum_{i=1}^n (y_i - \alpha - \beta x_i) x_i = 0$$

Aplicando a distributiva:

$$\sum_{i=1}^n (x_i y_i - \alpha x_i - \beta x_i^2) = 0$$

Separando o somatório nos leva a:

$$\sum_{i=1}^n x_i y_i - \alpha \sum_{i=1}^n x_i - \beta \sum_{i=1}^n x_i^2 = 0$$

Dividindo tudo por n temos:

$$\overline{xy} - \alpha \bar{x} - \beta \overline{x^2} = 0 \quad (II)$$

Substituindo a equação (I) na equação (II) temos:

$$\overline{xy} - (\bar{y} - \beta \bar{x}) \bar{x} - \beta \overline{x^2} = 0$$

Aplicando a distributiva:

$$\overline{xy} - \bar{x} \bar{y} + \beta \bar{x}^2 - \beta \overline{x^2} = 0$$

Finalmente, isolando β nos leva a:

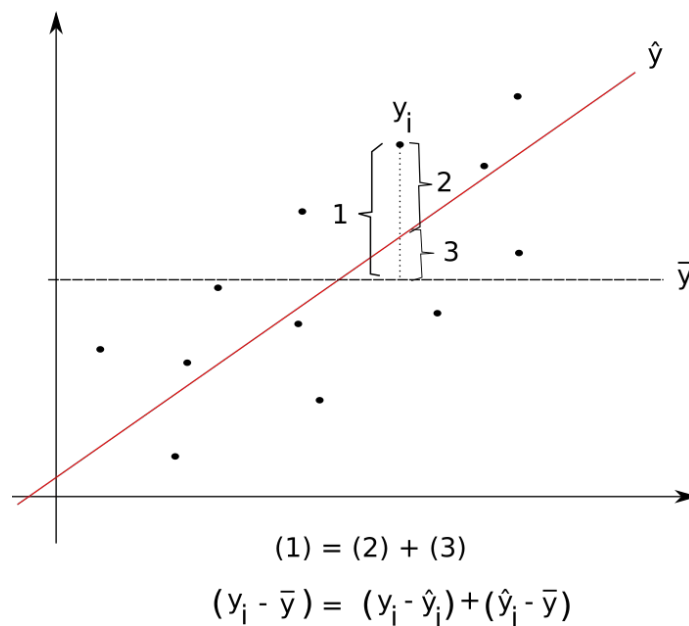
$$\beta = \frac{\overline{xy} - \bar{x} \bar{y}}{\bar{x}^2 - \overline{x^2}} \quad (\text{inclinação da reta})$$

Após o cálculo de β , basta substituir seu valor em α para determinar seu valor e consequentemente a equação da reta desejada. Em geral, após estimar os parâmetros α e β e construir a reta de regressão, utilizamos a equação da reta para estimar valores de y para novos valores de x.

Coeficiente de determinação

Em problemas de regressão linear, o coeficiente de determinação, denotado por r^2 , é a proporção da variância da variável dependente que é predita a partir da variável independente. É uma medida de ajustamento de um modelo estatístico linear generalizado. Seu valor varia de 0 a 1, indicando, em porcentagem, o quanto o modelo consegue explicar os valores observados. Quanto maior r^2 mais explicativo é o modelo, melhor ele se ajusta a amostra.

A seguir iremos deduzir a expressão de r^2 a partir de um modelo de regressão linear simples.



Da figura acima, fica claro que para um ponto y_i vale a relação $y_i - \bar{y} = (y_i - \hat{y}_i) + (\hat{y}_i - \bar{y})$. Elevando ambos os lados ao quadrado temos:

$$(y_i - \bar{y})^2 = [(y_i - \hat{y}_i) + (\hat{y}_i - \bar{y})]^2$$

Somando para todos os pontos (aplicando somatório):

$$\sum_i (y_i - \bar{y})^2 = \sum_i [(y_i - \hat{y}_i) + (\hat{y}_i - \bar{y})]^2 = \sum_i (y_i - \hat{y}_i)^2 + 2 \sum_i (y_i - \hat{y}_i)(\hat{y}_i - \bar{y}) + \sum_i (\hat{y}_i - \bar{y})^2$$

Iremos chamar as quantidades acima de:

$$TSS = \sum_i (y_i - \bar{y})^2 \quad (\text{Total Sum of the Squares})$$

$$RSS = \sum_i (y_i - \hat{y}_i)^2 \quad (\text{Residual Sum of the Squares})$$

$$ESS = \sum_i (\hat{y}_i - \bar{y})^2 \quad (\text{Explained Sum of the Squares})$$

Desejamos mostrar que $TSS = ESS + RSS$. Para isso, devemos mostrar que o termo

$$\sum_i (y_i - \hat{y}_i)(\hat{y}_i - \bar{y}) \quad (*)$$

é igual a zero. Do modelo de regressão linear sabemos que:

$$\begin{aligned}\hat{y}_i &= \alpha + \beta x_i \\ \bar{y} &= \alpha + \beta \bar{x}\end{aligned}$$

Então,

$$\begin{aligned}\hat{y}_i - \bar{y} &= \alpha + \beta x_i - \alpha - \beta \bar{x} = \beta (x_i - \bar{x}) \\ y_i - \hat{y}_i &= (y_i - \bar{y}) - (\hat{y}_i - \bar{y}) = (y_i - \bar{y}) - \beta (x_i - \bar{x})\end{aligned}$$

Também sabemos que:

$$\beta = \frac{\overline{xy} - \bar{x}\bar{y}}{\overline{x^2} - \bar{x}^2} \quad (**)$$

O numerador de (**) pode ser reescrito como:

$$\overline{xy} - \bar{x}\bar{y} = \overline{xy} - \bar{x}\bar{y} - \bar{x}\bar{y} + \bar{x}\bar{y}$$

Multiplicando tudo por n (podemos fazer isso pois iremos multiplicar o denominador também):

$$\sum_{i=1}^n x_i y_i - \bar{y} \sum_{i=1}^n x_i - \bar{x} \sum_{i=1}^n y_i + n \bar{x} \bar{y} = \sum_{i=1}^n x_i y_i - \bar{y} \sum_{i=1}^n x_i - \bar{x} \sum_{i=1}^n y_i + \sum_{i=1}^n \bar{x} \bar{y}$$

Agrupando em um somatório único:

$$\sum_{i=1}^n (x_i y_i - x_i \bar{y} - y_i \bar{x} + \bar{x} \bar{y}) = \sum_{i=1}^n [y_i (x_i - \bar{x}) - \bar{y} (x_i - \bar{x})] = \sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})$$

Analogamente, o denominador de (**) pode ser reescrito como:

$$\overline{x^2} - \bar{x}^2 = \overline{x^2} - \bar{x}^2 - \bar{x}^2 + \bar{x}^2$$

Multiplicando tudo por n (podemos fazer isso pois multiplicamos o numerador também):

$$\sum_{i=1}^n x_i^2 - \bar{x} \sum_{i=1}^n x_i - \bar{x} \sum_{i=1}^n x_i + n \bar{x}^2 = \sum_{i=1}^n x_i^2 - \bar{x} \sum_{i=1}^n x_i - \bar{x} \sum_{i=1}^n x_i + \sum_i \bar{x}^2$$

Agrupando em um somatório único:

$$\sum_{i=1}^n (x_i^2 - \bar{x} x_i - x_i \bar{x} + \bar{x}^2) = \sum_{i=1}^n [x_i (x_i - \bar{x}) - \bar{x} (x_i - \bar{x})] = \sum_{i=1}^n (x_i - \bar{x})(x_i - \bar{x}) = \sum_{i=1}^n (x_i - \bar{x})^2$$

Portanto, o valor de β é dado por:

$$\beta = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^n (x_i - \bar{x})^2}$$

Voltando a equação (*), temos:

$$\sum_i (y_i - \hat{y}_i)(\hat{y}_i - \bar{y}) = \beta \sum_i (x_i - \bar{x})(y_i - \hat{y}_i)$$

Substituindo $(y_i - \hat{y}_i)$ chegamos em:

$$\beta \sum_i (x_i - \bar{x})[y_i - \bar{y} - \beta(x_i - \bar{x})] = \beta \left[\sum_i (x_i - \bar{x})(y_i - \bar{y}) - \beta \sum_i (x_i - \bar{x})^2 \right]$$

Finalmente, substituindo o expressão para o β mais interno nos leva a:

$$\beta \left[\sum_i (x_i - \bar{x})(y_i - \bar{y}) - \beta \sum_i (x_i - \bar{x})^2 \right] = \beta \left[\sum_i (x_i - \bar{x})(y_i - \bar{y}) - \frac{\sum_i (x_i - \bar{x})(y_i - \bar{y})}{\sum_i (x_i - \bar{x})^2} \sum_i (x_i - \bar{x})^2 \right] = 0$$

E portanto, temos a relação desejada $TSS = ESS + RSS$, dada por:

$$\sum_i (y_i - \bar{y})^2 = \sum_i (y_i - \hat{y}_i)^2 + \sum_i (\hat{y}_i - \bar{y})^2$$

Define-se o coeficiente de determinação como a fração entre ESS e TSS:

$$r^2 = \frac{ESS}{TSS} = 1 - \frac{RSS}{TSS}$$

Pode-se mostrar que r nada mais é que o coeficiente de correlação de Pearson uma vez que

$$r^2 = \frac{ESS}{TSS} = \frac{\sum_i (\hat{y}_i - \bar{y})^2}{\sum_i (y_i - \bar{y})^2} = \frac{\beta^2 \sum_i (x_i - \bar{x})^2}{\sum_i (y_i - \bar{y})^2} = \frac{\left[\frac{\sum_i (x_i - \bar{x})(y_i - \bar{y})}{\sum_i (x_i - \bar{x})^2} \right]^2 \sum_i (x_i - \bar{x})^2}{\sum_i (y_i - \bar{y})^2} = \frac{\left[\sum_i (x_i - \bar{x})(y_i - \bar{y}) \right]^2}{\sum_i (x_i - \bar{x})^2 \sum_i (y_i - \bar{y})^2}$$

Obs:

$$\sum_i (x_i - \bar{x})(y_i - \bar{y}) = \sum_i x_i y_i - \bar{y} \sum_i x_i - \bar{x} \sum_i y_i + n \bar{x} \bar{y} \quad (\text{dividindo numerador e denominador por } n)$$

$$\overline{xy} - \bar{x} \bar{y} - \bar{x} \bar{y} + \bar{x} \bar{y} = \overline{xy} - \bar{x} \bar{y}$$

$$\sum_i (x_i - \bar{x})(x_i - \bar{x}) = \sum_i x_i^2 - \bar{x} \sum_i x_i - \bar{x} \sum_i x_i + n \bar{x}^2 \quad (\text{dividindo numerador e denominador por } n)$$

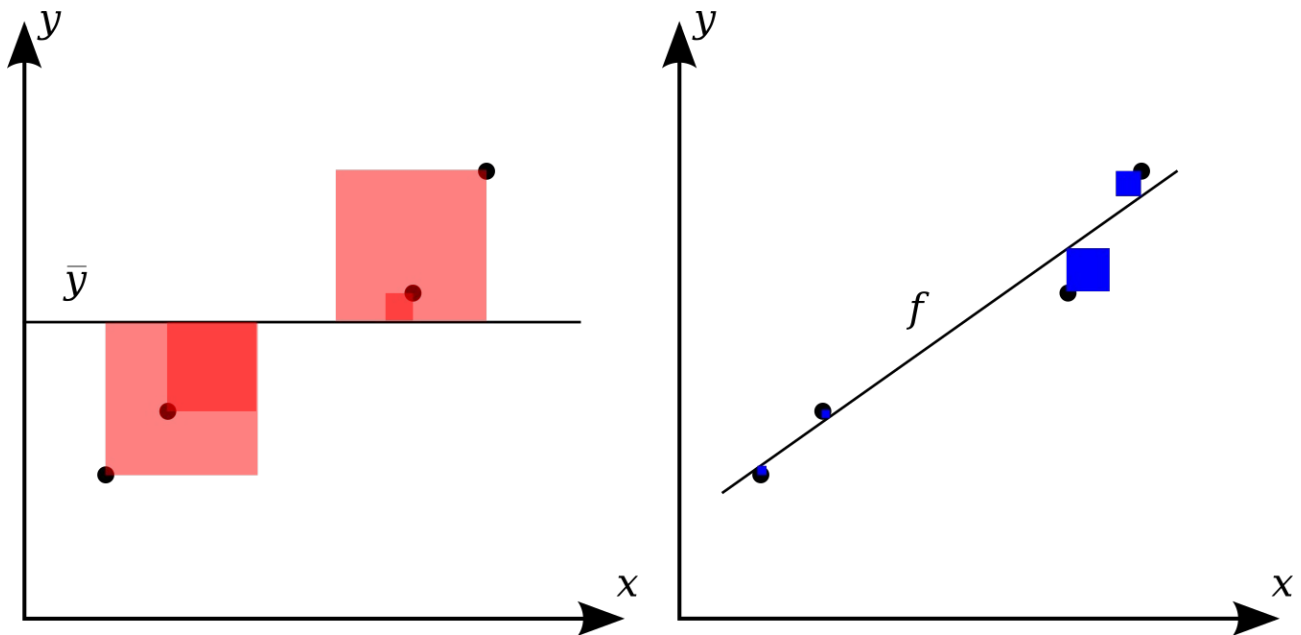
$$\overline{x^2} - \bar{x}^2 - \bar{x}^2 + \bar{x}^2 = \overline{x^2} - \bar{x}^2$$

Analogamente, para a variável y temos $\sum_i (y_i - \bar{y})(y_i - \bar{y}) = \overline{y^2} - \bar{y}^2$

Portanto, chegamos em:

$$r^2 = \frac{[\overline{xy} - \bar{x}\bar{y}]^2}{(\overline{x^2} - \bar{x}^2)(\overline{y^2} - \bar{y}^2)} \quad (\text{quadrado do coeficiente de correlação de Pearson})$$

Quanto melhor a regressão linear ajusta os dados em comparação com a simples média, mais próximo de 1 o valor de r^2 . As áreas na figura a seguir indicam o quadrado dos resíduos.



$$r^2 = 1 - \frac{RSS}{TSS} \quad (\text{RSS é a soma das áreas dos quadrados em relação a reta de regressão})$$

Quanto menor RSS, melhor pois a curva se ajusta melhor aos pontos.

Exercício: Taxas de juros fornecem um bom indicador para a previsão de aquecimento ou desaquecimento do mercado imobiliário. Em geral, com a queda na taxa de juros, nota-se um aumento no número de financiamentos imobiliários. Suponha os dados da tabela abaixo, em que x denota a taxa de juros e y denota o número de financiamentos imobiliários em uma determinada cidade.

Ano	1985	1986	1987	1988	1989	1990	1991	1992	1993	1994	1995	1996
x (%)	6.5	6.0	6.5	7.5	8.5	9.5	10.0	9.0	7.5	9.0	11.0	15.0

y	2165	2984	2780	1940	1750	1535	962	1310	2050	1695	856	510
-----	------	------	------	------	------	------	-----	------	------	------	-----	-----

a) Encontre a reta de regressão para os dados, computando manualmente os valores α e β

Para encontrar α e β devemos computar as quantidades: \bar{x} , \bar{y} , \overline{xy} , $\overline{x^2}$ e \bar{x}^2

$$\bar{x} = \frac{106}{12} = 8.833, \quad \bar{y} = \frac{20537}{12} = 1711.416, \quad \overline{xy} = \frac{163160}{12} = 13596.667, \quad \overline{x^2} = \frac{1003.5}{12} = 83.625$$

$$\bar{x}^2 = 8.833^2 = 78.021$$

Assim, temos:

$$\beta = \frac{\overline{xy} - \bar{x}\bar{y}}{\overline{x^2} - \bar{x}^2} = \frac{13596.667 - 8.833 * 1711.416}{83.625 - 78.021} = \frac{13596.667 - 15116.937}{83.625 - 78.021} = \frac{-1520.27}{5.604} = -271.28$$

$$\alpha = \bar{y} - \beta \bar{x} = 1711.416 - (-271.28) * 8.833 = 1711.416 + 2396.216 = 4107.632$$

E a equação da reta está definida como:

$$y = 4107.63 - 271.28x$$

b) Estime o número de financiamentos em 1997 se a taxa de juros for reduzida para 12.5%

Aplicando a equação da reta obtida no item a), temos:

$$\hat{y} = 4107.63 - 271.28 * 12.5 = 716.63 \approx 717$$

c) Compute o coeficiente de correlação de Pearson r

O coeficiente de correlação é dado por:

$$r = \frac{\overline{xy} - \bar{x}\bar{y}}{\sqrt{(\overline{x^2} - \bar{x}^2)(\overline{y^2} - \bar{y}^2)}}$$

e por isso precisamos computar a quantidade $\overline{y^2}$

$$\overline{y^2} = \frac{41212111}{12} = 3434342.58$$

$$r_{xy} = \frac{13596.667 - 15116.937}{\sqrt{(83.625 - 78.021) * (3434342.58 - 2928924.18)}} = \frac{-1520.27}{\sqrt{5.6 * 505418.4}} = \frac{-1520.7}{1682.36} = -0.9039$$

d) De acordo com o modelo, qual deveria ser a taxa de juros para que em um ano fossem realizados 1000 financiamentos?

Para isso devemos resolver a seguinte equação em x:

$$y = 4107.63 - 271.28x$$

$$x = \frac{1000 - 4107.63}{-271.28} = 11.45$$

Veremos a seguir um script em Python que implementa a regressão linear simples.

```

import numpy as np
import matplotlib.pyplot as plt

# Leitura dos dados
x = np.loadtxt('./slr06/slr06l1.txt')
y = np.loadtxt('./slr06/slr06l2.txt')

# Estima beta (inclinação da reta)
x_bar = x.mean()
y_bar = y.mean()
x2_bar = (1/len(x))*sum(x*x)
y2_bar = (1/len(y))*sum(y*y)
xy_bar = (1/len(x))*sum(x*y)
beta = (xy_bar - x_bar*y_bar)/(x2_bar - x_bar**2)

# Estima intercepto
alpha = y_bar - beta*x_bar

# Estima coeficiente de correlação de Pearson
r = (xy_bar - x_bar*y_bar)/np.sqrt((x2_bar - x_bar**2)*(y2_bar - y_bar**2))

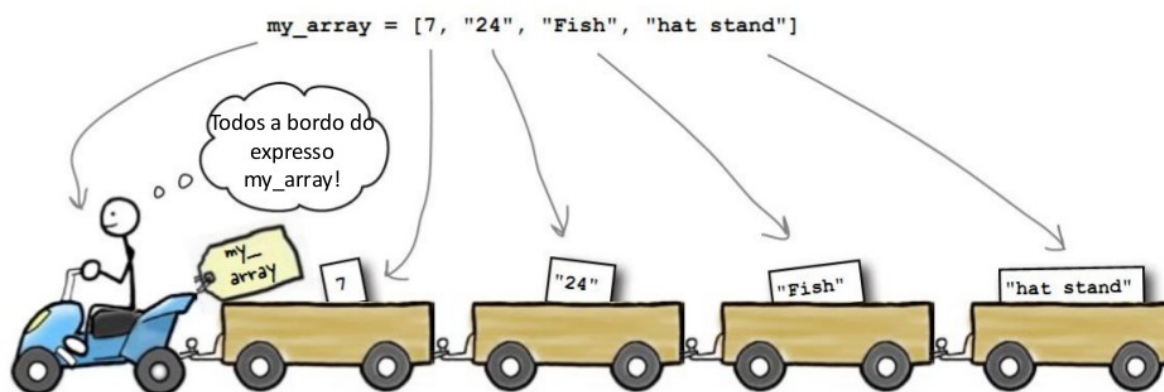
print('Equação da reta: y = %.3f*x + %.3f' %(beta, alpha))
print('Coeficiente de correlação de Pearson: %.3f' %r)

# Plota os gráficos
plt.figure(1)
eixox = np.linspace(min(x), max(x), 100)
eixoy = beta*eixox + alpha

plt.plot(eixox, eixoy)
plt.scatter(x, y, s=20, c='r')
plt.show()

```

Listas, vetores e matrizes



O trem de dados my_array é uma única variável

Tuplas e listas são variáveis compostas heterogêneas, isto é, estruturas de dados em Python muito úteis para armazenar uma coletânea de objetos de tipos de dados diversos de maneira indexada. A figura a seguir ilustra essa ideia como um trem de dados.

A única diferença entre uma tupla e uma lista, é que tuplas são objetos imutáveis, ou seja, uma vez criados não podem ser modificados. Em outras palavras, é impossível usar o operador de atribuição com uma tupla. As listas por outro lado podem ser modificadas sempre que necessário e portanto são muito mais utilizadas na prática do que as tuplas.

```
# cria uma lista vazia
lista = []
# cria uma lista com 3 notas
notas = [7.5, 9, 8.3]
# imprime a primeira nota
print(notas[0])
# mudando a primeira nota
notas[0] = 8.7
```

Ex: Faça um programa que leia 4 notas, armazene-as em uma lista e mostre-as na tela juntamente com a média.

```
notas = []
soma = 0
i = 1
while i <= 4:
    n = float(input("Nota: "))
    notas.append(n)
    soma += n
    i += 1
print("Notas:", notas)
print("Média: %4.2f" % (soma/4))
```

Slicing (Fatiamento)

Ao se trabalhar com listas em Python, uma ferramenta bastante poderosa para obter subconjuntos de uma lista é a técnica conhecida como *slicing*. Perguntas como quais são os elementos das posições pares, ou qual é a sublista obtida entre o terceiro e o sétimo elemento podem ser facilmente respondidas através do fatiamento. Suponha a lista $L = [9, 8, 7, 6, 5, 4, 3, 2, 1]$.

$L[-1]$ retorna o último elemento: 1
 $L[-2]$ retorna o penúltimo elemento: 2
 $L[-3]$ retorna o antepenúltimo elemento: 3

$L[:3]$ retorna: [9, 8, 7] (note que não inclui o limite superior)
 $L[3:]$ retorna: [6, 5, 4, 3, 2, 1]
 $L[2:5]$ retorna: [7, 6, 5]

$L[::2]$ retorna: [9, 7, 5, 3, 1] (apenas as posições pares)
 $L[::3]$ retorna: [9, 6, 3]
 $L[::-1]$ retorna: [1, 2, 3, 4, 5, 6, 7, 8, 9] (lista ao contrário)
 $L[::-2]$ retorna: [1, 3, 5, 7, 9]

List comprehensions

Suponha agora que desejamos criar uma nova lista $L2$ que contenha somente os elementos de L que sejam divisíveis por 3. Uma forma rápida de construir $L2$ é através da técnica list comprehensions.

```
L = [9, 8, 7, 6, 5, 4, 3, 2, 1]
L2 = [x for x in L if x % 3 == 0]
```

Por exemplo, se desejamos construir L3 com os números pares e uma lista L4 com o quadrados dos números ímpares, teremos:

```
L3 = [x for x in L if x % 2 == 0]
L4 = [x**2 for x in L if x% 2 == 1]
```

Veremos a seguir problemas e exercícios que utilizam estruturas do tipo lista.

Ex: Supondo que uma lista de n elementos reais represente um vetor no R^n , faça uma função para computar o produto escalar entre 2 vetores.

```
# Calcula o produto escalar entre 2 vetores
def produto_escalar(u, v):
    soma = 0
    for i in range(len(u)):
        soma = soma + u[i]*v[i]
    return soma
```

Faça uma função que compute a norma de um vetor: $\|\vec{v}\| = \sqrt{v_1^2 + v_2^2 + \dots + v_n^2}$

Ex: Faça uma função que encontre o maior elemento de uma lista.

```
# Retorna o maior elemento de uma lista de inteiros
def maximo(lista):
    maior = lista[0]
    for i in range(len(lista)):
        if lista[i] > maior:
            maior = lista[i]
    return maior
```

Ex: Uma tarefa fundamental na computação consiste em dado uma lista e um valor qualquer, verificar se aquele valor pertence a lista ou não. Essa funcionalidade é usada por exemplo em qualquer sistema que exige o login de um usuário (para verificar se o CPF da pessoa está cadastrada). Faça uma função que, dada uma lista de inteiros L e um número inteiro x, verifique se x está ou não em L. A função deve retornar o índice do elemento (posição) caso ele pertença a ele ou o valor lógico False se ele não pertence a L. (isso equivale ao operador in de Python)

```
def busca_sequencial(lista, x):
    achou = False
    i = 0
    while i < len(lista) and not achou:
        if (lista[i] == x):
            achou = True
            pos = i
        else:
            i = i + 1
    if achou:
        return pos
    else:
        return achou
```

Ex: Deseja-se contar quantos números entre 1000 e 9999 (inclusive) são ímpares, divisíveis por 7 e cujo primeiro algarismo é um número par. Como resolver o problema em Python com listas?

```
def impar(n):
    if n % 2 == 1:
        return True

def div_por_7(n):
    if n % 7 == 0:
        return True

def inicia_par(s):
    num = int(s[0])
    if num % 2 == 0:
        return True

def armazena_numeros(low, high):
    L = []
    for i in range(low, high+1):
        s = str(i) # converte para string
        if impar(i) and div_por_7(i) and inicia_par(s):
            L.append(i)
    return L

# Início do script
a = int(input('Entre com o limite inferior do intervalo: '))
b = int(input('Entre com o limite superior do intervalo: '))

lista = armazena_numeros(a, b)

print(lista)
print('Há %d números satisfazendo as condições.' %(len(lista)))
```

Ex: Faça uma função que receba como entrada um número inteiro na base 10 e retorne uma lista contendo sua representação em binário (cada elemento da lista é um bit 0 ou 1).

```
def decimal_binario(n):
    binario = []
    while n != 0:
        quociente = n // 2
        resto = n % 2
        binario.append(resto)
        n = quociente
    return binario[::-1]

# Início do script
n = int(input('Entre com um número inteiro: '))
b = decimal_binario(n)
print('O número %d em binário é: ' %n)
print(b)
```

Ex: Dada a seguinte lista de 10 elementos L = [-8, -29, 100, 2, -2, 40, 23, -8, -7, 77] faça funções que recebam L como entrada e retorne:

- a) o menor número da lista
- b) o maior número da lista
- c) a média de todos os elementos da lista
- d) a lista inversa
- e) o desvio padrão dos elementos da lista

$$Desvio = \sqrt{\frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^2}, \text{ onde } \bar{x} \text{ é a média e } n \text{ é o tamanho da lista}$$

O script a seguir mostra a solução em Python.

```
import math

def minimo(L):
    menor = L[0]
    for x in L:
        if x < menor:
            menor = x
    return menor

def maximo(L):
    maior = L[0]
    for x in L:
        if x > maior:
            maior = x
    return maior

def media(L):
    soma=0
    for x in L:
        soma = soma + x
    media = soma/len(L)
    return media

def desvio_padrao(L):
    m = media(L)
    soma = 0
    for x in L:
        soma = soma + (x - m)**2
    desvio = math.sqrt(soma/len(L))
    return desvio

def inverte(L):
    return L[::-1]

# Início do script
L = [-8, -29, 100, 2, -2, 40, 23, -8, -7, 77]
print('Mínimo: %d' %minimo(L))
print('Máximo: %d' %maximo(L))
print('Média: %f' %media(L))
print('Desvio padrão: %f' %desvio_padrao(L))
print('Inverso:', inverte(L))
```

O site Project Euler (<https://projecteuler.net/archives>) contém uma série de problemas matemáticos desafiadores que podem ser resolvidos através da programação de computadores. O exercício anterior é um deles. O exercício a seguir é o problema 14 da lista do Project Euler.

Ex47: A maior sequencia de Collatz

Esse é um dos problemas matemáticos mais misteriosos do mundo, principalmente porque não existem provas formais da sua solução. Suponha que seja escolhido um número inteiro qualquer n . A sequencia iterativa a seguir é definida para qualquer n positivo:

$$\begin{aligned} n &\rightarrow n/2, && \text{se } n \text{ é par} \\ n &\rightarrow 3n + 1, && \text{se } n \text{ é ímpar} \end{aligned}$$

Por exemplo, usando o número 13 como entrada, iremos produzir a seguinte sequencia:

$13 \rightarrow 40 \rightarrow 20 \rightarrow 10 \rightarrow 5 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1$

O fato intrigante é que para todo n , cedo ou tarde a sequencia atinge o número 1 (essa é a conjectura de Collatz). A pergunta é: qual é o número n até 1 milhão, que produz a maior sequencia, isto é, que demora mais para atingir o número 1. Sugestão: armazene a sequencia de cada número numa lista e retorne como saída a maior delas.

```
def collatz(n):
    sequencia = [n]
    while n > 1:
        if n % 2 == 0:
            n = n//2
        else:
            n = 3*n+1
        sequencia.append(n)
    return sequencia

# Testa as sequencias com todos valores de 1 a 1000000
maior = 1
lista = []
for i in range(1, 1000000):
    L = collatz(i)
    if len(L) > maior:
        maior = len(L)
        lista = L

print(lista)
print()
print('Tamanho: %d' %maior)
```

Vetores

Em Python, listas são objetos extremamente genéricos, podendo ser utilizados em uma gama de aplicações. Entretanto, o pacote Numpy (Numerical Python) oferece a definição de um tipo de dados específico para representar vetores: são os chamados **arrays**. Em resumo, um array é praticamente uma lista homogênea de números inteiros ou reais. A principal diferença para uma lista é que aqui todos os elementos devem ser do mesmo tipo de dados: int ou float. Por serem estruturas de dados homogêneas possuem diversas funções que facilitam diversos cálculos

matemáticos. Um exemplo é o seguinte: suponha que você tenha uma lista em que cada elemento representa o gasto em reais de uma compra efetuada pelo seu cartão.

```
L = [13.5, 8.0, 5.99, 27.30, 199.99, 57.21]
```

Você deseja dividir cada um dos valores pelo gasto total, para descobrir a porcentagem referente a cada compra. Como você tem uma lista, você deve fazer o seguinte:

```
lista = L
for i in range(len(L)):
    lista[i] = L[i]/sum(L)
```

Ou seja, não é permitido fazer

```
lista = L/sum(L) → ERRO
```

Mas se os dados tiverem sido armazenados num vetor, é possível fazer diretamente o comando acima:

```
import numpy as np

# criando um vetor (as duas maneiras são equivalentes)
v = np.array(L)
v = np.array([13.5, 8.0, 5.99, 27.30, 199.99, 57.21])

v = v/sum(v) → OK
```

Algumas funções importantes que todo objeto vetor possui, supondo que nosso vetor chama-se v:

```
v.max() - retorna o maior elemento do vetor
v.min() - retorna o menor elemento do vetor
v.argmax() - retorna o índice do maior elemento do vetor
v.argmin() - retorna o índice do menor elemento do vetor
v.sum() - retorna a soma dos elementos do vetor
v.mean() - retorna a média dos elementos do vetor
v.prod() - retorna o produto dos elementos do vetor
v.T - retorna o vetor transposto
v.clip(a, b) - o que é menor que a vira a e o que é maior que b vira b
v.shape() - retorna as dimensões do vetor/matriz
```

Matrizes

Matrizes em Python também são objetos criados com o comando `np.array`. Elas são vetores que possuem mais de uma dimensão, sendo tipicamente duas. Suponha que desejamos criar uma matriz 3 x 3 contendo os elementos de 1 até 9.

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

Isso é feito com o seguinte comando:

```
M = np.array([[1,2,3], [4,5,6], [7,8,9]])
```

Note que uma matriz em Python nada mais é que uma lista de listas homogêneas de inteiros ou floats. A seguir veremos um exemplo onde algumas operações básicas com matrizes são realizadas.

```
import numpy as np

M = np.array([[1,2],[3,4]])
N = np.array([[5,6],[7,8]])

# soma de 2 matrizes
C = M + N
# subtração de 2 matrizes
D = M - N
# multiplicação elemento por elemento
E = M*N
# multiplicação de matrizes (ou produto escalar)
F = np.dot(M, N)
```

A classe ndarray

Todas as variáveis compostas definidas pelo pacote numpy são da classe *ndarray*, o que faz com que elas herdem vários atributos e métodos. Veremos a seguir alguns deles em exemplos práticos.

```
# gera uma matriz com números aleatórios entre 1 e 99
A = np.random.randint(1, 100, (5, 5))
print(A.ndim)    # retorna o número de dimensões

B = np.random.random((3, 3, 3))
print(B.ndim)

print(A.dtype)   # retorna o tipo de dados
print(B.dtype)

print(A.shape)   # retorna as dimensões
print(B.shape)

print(A.size)     # retorna o número de elementos
print(B.size)

print(A.itemsize) # retorna o tamanho em bytes de cada elemento
print(B.itemsize)
```

Para a criação de vetores e matrizes, há diversas funções úteis, como as descritas a seguir:

```
A = np.zeros((5, 5)) # Cria matriz 5 x 5 de 0's
B = np.ones((5, 5))  # Cria matriz 5 x 5 de 1's
# Cria matriz 2 x 2 de complexos
C = np.array([[1+2j, 3- 4j ], [5 + 6j, 7 - 8j]])
```

Para criar sequências igualmente espaçadas, o pacote numpy disponibiliza duas funções: *arange()* e *linspace()*. Vejamos exemplos a seguir.

```
x = np.arange(0, 2, 0.2) # divide [0, 2) em intervalos de 0.2
y = np.linspace(0, 2, 9) # divide [0, 2] em 9 pontos igualmente espaçados
```

Obs: *arange* não inclui o limite superior, mas *linspace* inclui!

Uma tarefa comum em programação numérica consiste em converter uma matriz $N \times N$ em um vetor com N^2 elementos, empilhando as linhas da matriz. Em Python isso pode ser feito pelo comando `reshape`.

```
B = np.random.randint(1, 10, (5, 5))
A = B.reshape((1, B.size))
```

Também é possível utilizar diretamente o método `ravel()`

```
A = B.ravel()
```

Em matrizes, podemos computar uma determinada operação para todos os elementos de uma linha ou coluna. Vejamos alguns exemplos a seguir.

```
A = np.random.randint(1, 100, (5, 5))
A.sum()      # soma todos elementos
A.min()      # determina menor elemento
A.max()      # determina o maior elemento
A.mean()     # calcula a média

A.sum(axis=0) # retorna a soma de cada coluna
A.sum(axis=1) # retorna a soma de cada linha
A.min(axis=0) # menor de cada coluna
A.min(axis=1) # menor de cada linha
A.max(axis=0) # maior de cada coluna
A.max(axis=1) # maior de cada linha
A.mean(axis=0) # média de cada coluna
A.mean(axis=1) # média de cada linha
```

Assim como nas listas, é possível selecionar submatrizes utilizando o fatiamento (*slicing*). Seja A:

```
[[18, 34, 26, 94, 90],
 [96, 68, 65, 19, 90],
 [59, 15, 7, 71, 47],
 [6, 99, 50, 26, 62],
 [63, 44, 36, 39, 80]]
```

Ao acessarmos

```
A[1:4, 1:4]      # pega as linhas 1, 2 e 3 e colunas 1, 2 e 3
```

estamos descartando as bordas e tomando apenas o miolo, ou seja:

```
[[68, 65, 19],
 [15, 7, 71],
 [99, 50, 26]]
```

Outra operação importante é a concatenação de vetores, realizada pela função *concatenate*:

```
x = np.random.randint(1, 10, 5)
y = np.random.randint(1, 10, 5)
z = np.concatenate((x, y))
```

Um aspecto muito importante em Python é que, com variáveis compostas, como vetores e matrizes, ao se utilizar o operador de atribuição =, na verdade estamos passando uma referência e não uma cópia da variável em questão. Por exemplo:

```
A = np.random.randint(1, 10, (5, 5))
print(A)
```

```
[[8, 1, 6, 2, 5],
 [7, 4, 5, 1, 5],
 [7, 1, 2, 8, 2],
 [5, 8, 8, 8, 1],
 [5, 1, 6, 8, 3]]
```

```
B = A
A[0, 0] = -1
print(B)
```

```
[[-1, 1, 6, 2, 5],
 [ 7, 4, 5, 1, 5],
 [ 7, 1, 2, 8, 2],
 [ 5, 8, 8, 8, 1],
 [ 5, 1, 6, 8, 3]]
```

Note que ao modificar B, essa mudança foi refletida em A. Isso ocorre pois na verdade pois A e B são apenas duas referências distintas para a mesma variável. Se quisermos realizar uma cópia dos dados de A para B, é preciso utilizar o método *copy()*.

```
B = A.copy()
```

Ex: Uma imagem em tons de cinza é representada computacionalmente por uma matriz de dimensões n x n em que cada pixel é um número inteiro entre 0 (preto) e 255 (branco). O programa a seguir, lê uma imagem da internet, salva-a no computador e faz a sua leitura, criando uma matriz de inteiros. O histograma de uma imagem é um gráfico que mostra a distribuição dos níveis de cinza da imagem, indicando por exemplo se a imagem é muito escura ou clara.

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

# Calcula a moda - a cor (nível de cinza) que mais ocorre na imagem
def moda(h):
    maior = h[0]
    pos = 0
    for i in range(len(h)):
        if h[i] > maior:
            maior = h[i]
            pos = i
    return pos

urllib.request.urlretrieve('https://www.ece.rice.edu/~wakin/images/
lena512.bmp', 'lena.bmp')

matriz = plt.imread('lena.bmp')
plt.figure(1)
```

```
plt.imshow(matriz, cmap='gray') # mapa de cores em tons de cinza
plt.show()

# Cria o histograma
h = np.zeros(256)
for i in range(matriz.shape[0]):
    for j in range(matriz.shape[1]):
        h[matriz[i,j]] += 1

# Plota o gráfico do histograma
eixo_x = list(range(256))
plt.figure(2)
plt.bar(eixo_x, h)
plt.show()

m = moda(h)
print('Moda = %d' %m)
```

Para funções relacionadas a álgebra linear, como inversão de matrizes, solução de sistemas lineares, decomposição espectral (cálculos de autovalores e autovetores), há o pacote linalg, dentro do numpy. Um resumo das principais funções é listado abaixo.

Produtos entre vetores e matrizes

<code>dot(a, b[, out])</code>	Produto de duas matrizes.
<code>inner(a, b)</code>	Produto interno entre dois vetores
<code>outer(a, b[, out])</code>	Produto externo entre dois vetores
<code>matrix_power(a, n)</code>	Eleva matriz quadrada a potência n
<code>kron(a, b)</code>	Produto de Kronecker

Decomposições de matrizes

<code>linalg.cholesky(a)</code>	Decomposição de Cholesky
<code>linalg.qr(a[, mode])</code>	Fatorização de QR
<code>linalg.svd(a)</code>	Decomposição em valores singulares

Autobvalores e autovetores

<code>linalg.eig(a)</code>	Computa autovalores e autovetores
<code>linalg.eigh(a)</code>	Computa os autovalores e autovetores de matriz Hermitiana complexa
<code>linalg.eigvals(a)</code>	Computa os autovalores

Normas

<code>linalg.norm(x)</code>	Calcula a norma
<code>linalg.cond(x[, p])</code>	Calcula o número de condição de uma matriz
<code>linalg.det(a)</code>	Calcula determinante de uma matriz
<code>trace(a)</code>	Soma dos elementos da diagonal

Sistemas lineares

<code>linalg.solve(a, b)</code>	Resolve sistema linear de equações
<code>linalg.lstsq(a, b)</code>	Retorna a solução de mínimos quadrados para sistema linear
<code>linalg.inv(a)</code>	Inversa da matriz
<code>linalg.pinv(a)</code>	Pseudo-inversa da matriz

Para a referência completa do pacote numpy, recomenda-se acessar o link abaixo:

<https://docs.scipy.org/doc/numpy-dev/user/quickstart.html>

Plotagem e visualização de dados

O principal pacote da linguagem Python para a plotagem e visualização de dados é o matplotlib. Ele serve de base para diversos outros pacotes mais avançados e acaba sendo a alternativa mais conhecida para esse tipo de aplicação. Originalmente criado pelo biólogo e neurocientista americano John D. Hunter, a biblioteca hoje possui uma comunidade ativa atuando em seu desenvolvimento. Pyplot é um módulo do Matplotlib que oferece uma interface semelhante ao MATLAB. Matplotlib é projetado e desenvolvido para ter a mesma usabilidade do MATLAB, mas com a flexibilidade da linguagem Python, e a vantagem de ser código aberto (*open source*) e totalmente gratuito. Sendo assim, iremos sempre importar esse módulo, com o comando:

```
import matplotlib.pyplot as plt
```

A seguir veremos como criar alguns tipos de gráficos comumente utilizados na análise de dados, como gráficos de linha, histogramas, gráficos de dispersão, etc.

Exemplo 1 – Gráfico de linha

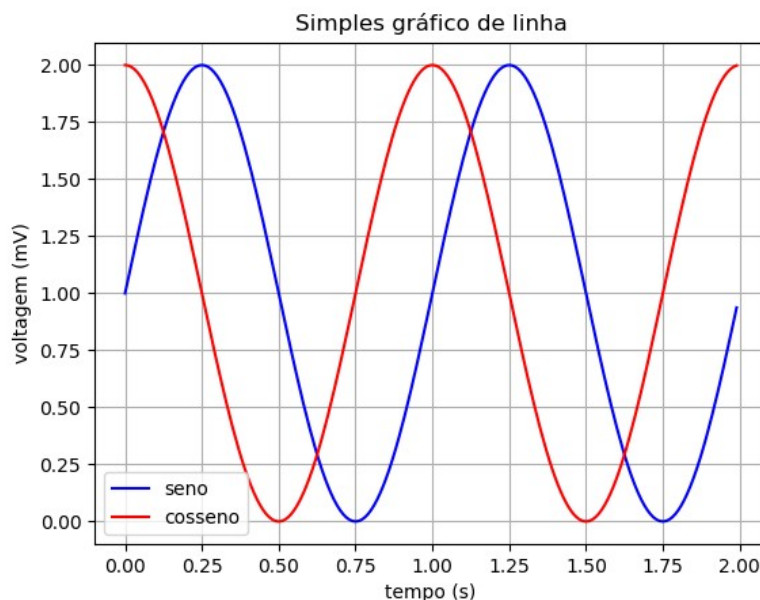
```
import matplotlib.pyplot as plt
import numpy as np

# Dados para plotagem
t = np.arange(0.0, 2.0, 0.01)    # de 0 a 2 em intervalos de 0.01
s = 1 + np.sin(2*np.pi*t)       # senoide ao redor da reta y = 1
y = 1 + np.cos(2*np.pi*t)       # cosseno ao redor da reta y = 1

fig, ax = plt.subplots()         # cria figura com um único eixo
ax.plot(t, s, color='blue', label='seno')    # plota gráfico no eixo
ax.plot(t, y, color='red', label='cosseno')   # plota gráfico no eixo

# Define rótulos para eixos x, y e cria título (opcional)
ax.set(xlabel='tempo (s)', ylabel='voltagem (mV)', title='Gráfico de linha')
ax.grid()                        # adiciona grade no gráfico
ax.legend()                      # adiciona legenda

fig.savefig('line.png')          # salva gráfico como arquivo de imagem
plt.show()                      # mostra gráfico na tela
```



Exemplo 2 – gráfico com dois eixos

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

# Cria os eixos x (com 200 e 100 pontos)
x1 = np.linspace(0.0, 5.0, 200)
x2 = np.linspace(0.0, 2.0, 100)

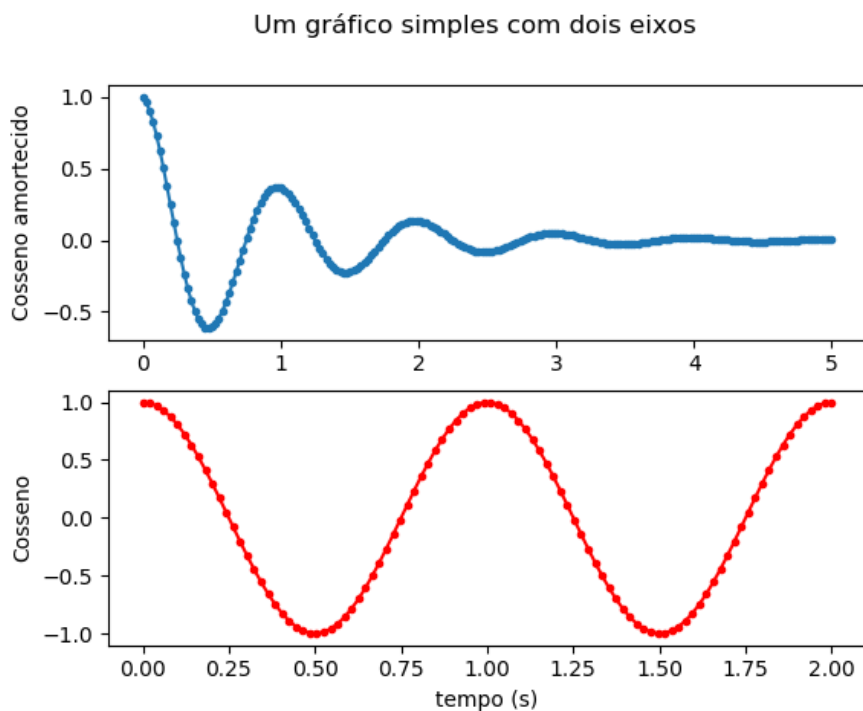
# Calcula o cosseno e sua versão amortecida
y1 = np.cos(2 * np.pi * x1) * np.exp(-x1)
y2 = np.cos(2 * np.pi * x2)

# Gera uma figura com dois eixos (2 linhas e 1 coluna)
fig, (ax1, ax2) = plt.subplots(2, 1)
fig.suptitle('Um gráfico simples com dois eixos')

# Eixo 1
ax1.plot(x1, y1, '-.')
ax1.set_ylabel('Cosseno amortecido')

# Eixo 2
ax2.plot(x2, y2, '-.', color='red')
ax2.set_xlabel('tempo (s)')
ax2.set_ylabel('Cosseno')

plt.show()
```



Exemplo 3 – Histograma

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

# Semente para geração de números aleatórios (opcional)
np.random.seed(19680801)

# Gera dados de distribuição normal
mu = 100          # média
sigma = 15        # desvio padrão
x = mu + sigma * np.random.randn(500)  # 500 números

# Define o número de bins do histograma
num_bins = 50

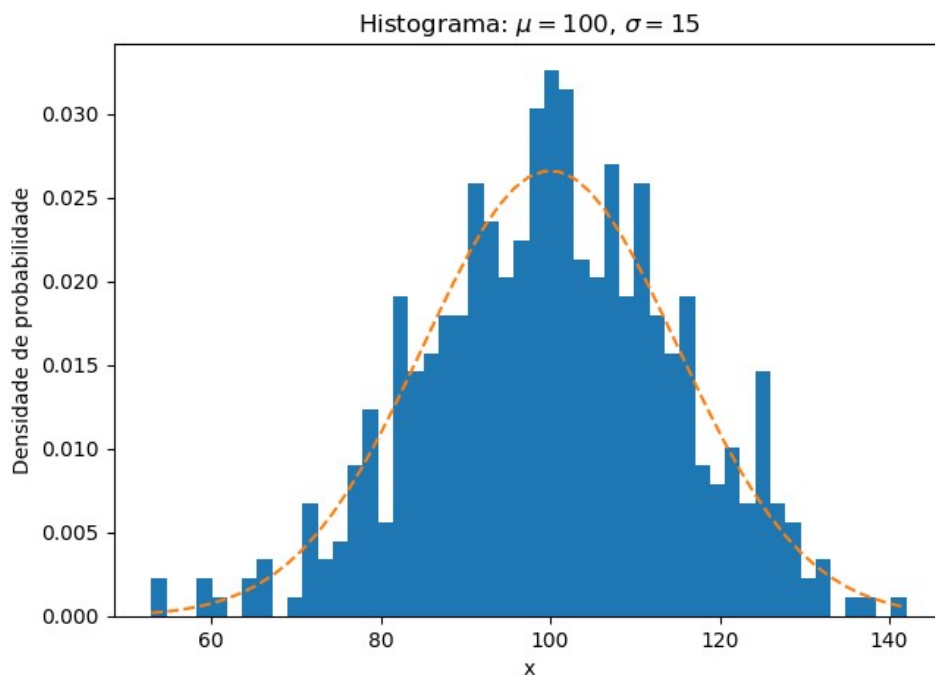
# Cria figura com um único eixo
fig, ax = plt.subplots()

# Plota histograma dos dados
n, bins, patches = ax.hist(x, num_bins, density=True)

# Faz o gráfico da distribuição que melhor se ajusta
y = ((1/(np.sqrt(2*np.pi)*sigma))*np.exp(-0.5*(1/sigma*(bins - mu)**2))
ax.plot(bins, y, '--')
ax.set_xlabel('x')
ax.set_ylabel('Densidade de probabilidade')
ax.set_title(r'Histograma:  $\mu=100$ ,  $\sigma=15$ ')

# Ajusta espaçamento
fig.tight_layout()

plt.show()
```



Exemplo 4 – Dispersão

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

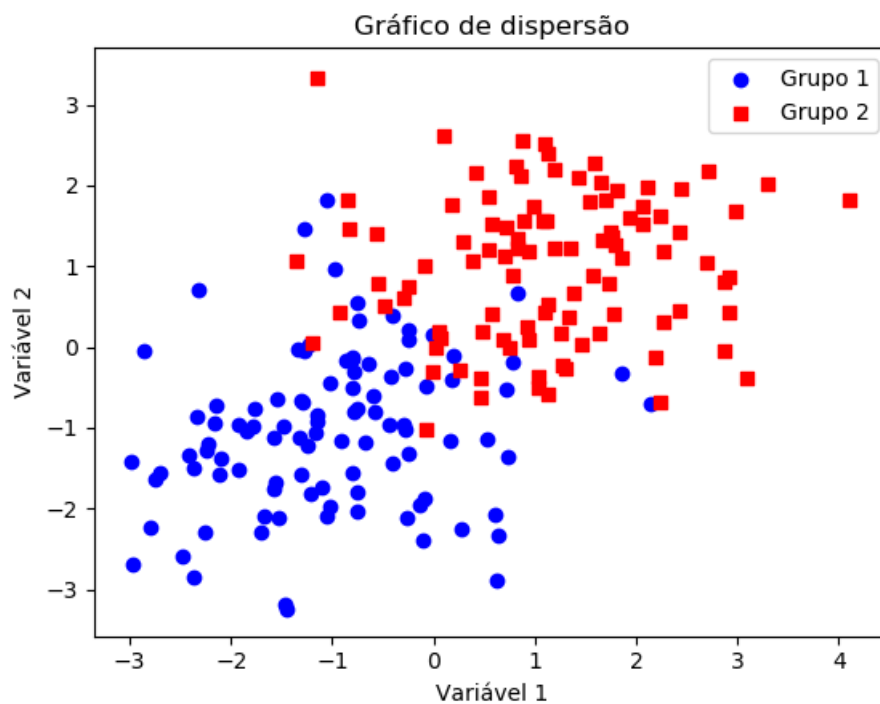
# Gera dados do agrupamento 1
x = np.random.normal(-1, 1, (2, 100))

# Gera dados do agrupamento 2
y = np.random.normal(1, 1, (2, 100))

# Cria figura com um único eixo
fig, ax = plt.subplots()

# Plota gráfico de dispersão
ax.scatter(x[0,:], x[1,:], marker='o', color='blue', label='Grupo 1')
ax.scatter(y[0,:], y[1,:], marker='s', color='red', label='Grupo 2')
ax.set_xlabel('Variável 1')
ax.set_ylabel('Variável 2')
ax.set_title('Gráfico de dispersão')
ax.legend()

# Mostra na tela
plt.show()
```



Exemplo 5 – Múltiplos gráficos

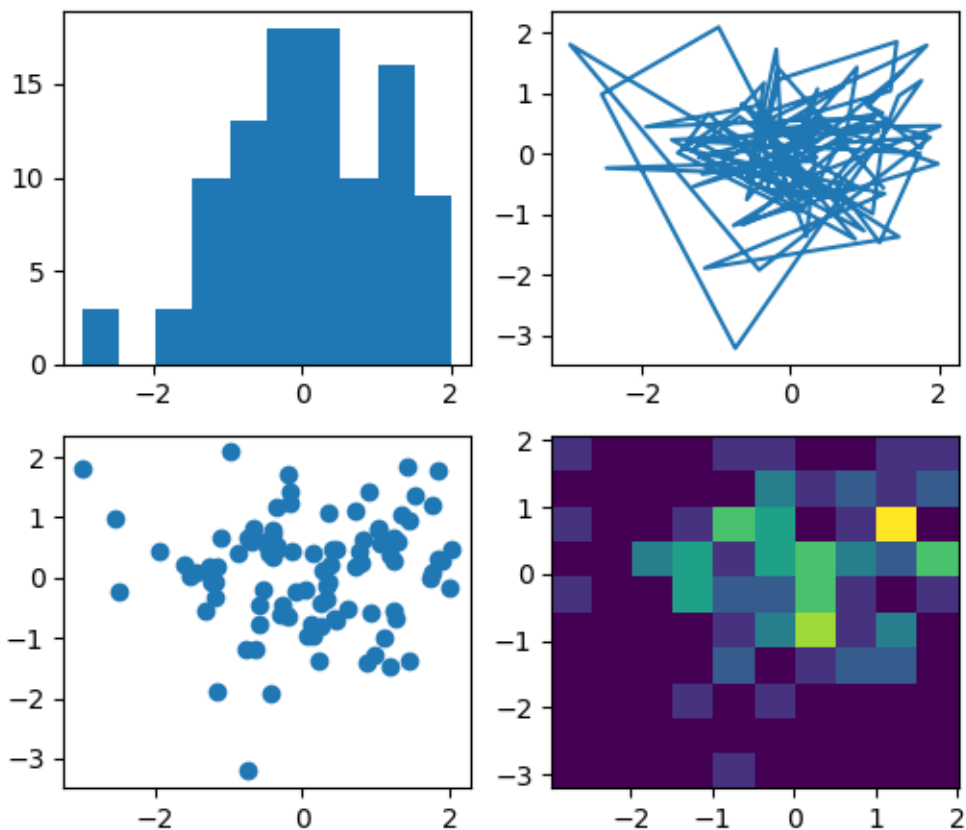
```
import matplotlib.pyplot as plt
import numpy as np

# Gera dados com distribuição normal padrão
data = np.random.randn(2, 100)

# Cria uma figura com 4 eixos
fig, axs = plt.subplots(2, 2, figsize=(5, 5))

# Plota diferentes tipos de gráficos
axs[0, 0].hist(data[0])
axs[1, 0].scatter(data[0], data[1])
axs[0, 1].plot(data[0], data[1])
axs[1, 1].hist2d(data[0], data[1])

# Mostra na tela
plt.show()
```



Exemplo 6 – Gráfico em 3 dimensões (surface plot)

```
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
from matplotlib import cm
import numpy as np

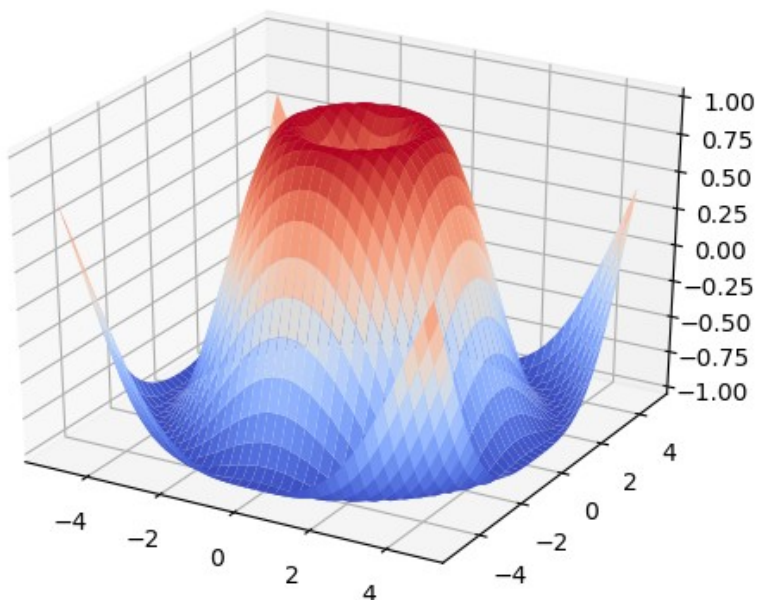
# Cria uma figura
fig = plt.figure()
# Cria eixos 3D (precisa incluir pacote Axes3D)
ax = fig.gca(projection='3d')

# Gera função para plotagem
X = np.arange(-5, 5, 0.25)
Y = np.arange(-5, 5, 0.25)
X, Y = np.meshgrid(X, Y)
R = np.sqrt(X**2 + Y**2)
Z = np.sin(R)

# Plota a superfície
surf = ax.plot_surface(X, Y, Z, cmap=cm.coolwarm)

# Define o intervalo no eixo z
ax.set_zlim(-1.01, 1.01)

# Mostra na tela
plt.show()
```



Recorrência logística e sistemas caóticos

Um modelo matemático simples, porém capaz de gerar comportamentos caóticos e imprevisíveis é a recorrência logística (*logistic map*). Imagine que desejamos criar uma equação para modelar o número de indivíduos de uma população de coelhos a partir de uma população inicial. A equação mais simples seria algo do tipo:

$$x_{n+1} = r x_n$$

onde $r > 0$ denota a taxa de crescimento. Porém, na natureza sabemos que devido a limitação de espaço e a disputa pelos recursos, populações não tendem a crescer indefinidamente. Há um ponto de equilíbrio em que o número de indivíduos tende a se estabilizar ao redor. Sendo assim, podemos definir a seguinte equação:

$$x_{n+1} = r x_n (1 - x_n)$$

em que o $x_n \in [0,1]$ a porcentagem de indivíduos vivos e o termo $(1 - x_n)$ tende a zero quando essa porcentagem se aproxima do valor máximo de 100%. Esse modelo é conhecido como recorrência logística. Veremos a seguir que fenômenos caóticos emergem desse simples modelo, que aparentemente possui um comportamento bastante previsível.

Primeiramente, note que o número de indivíduos no tempo $n+1$ é uma função quadrática do número de indivíduos no tempo n , pois:

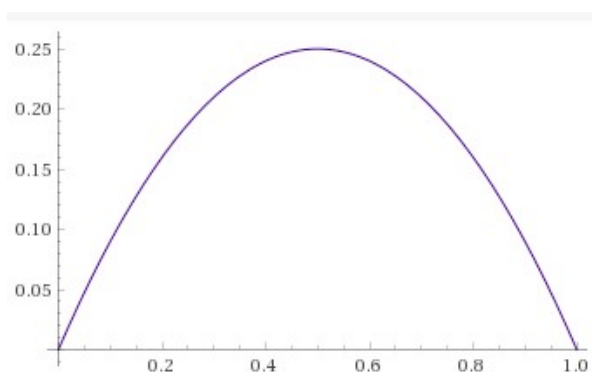
$$x_{n+1} = f(x_n) = -r x_n^2 + r x_n$$

ou seja, temos uma equação do segundo grau com $a = -r$, $b = r$ e $c = 0$. Como $a < 0$, a concavidade da parábola é para baixo, ou seja, ela admite um ponto de máximo. Derivando $f(x_n)$ em relação a x_n e igualando a zero, temos o ponto de máximo:

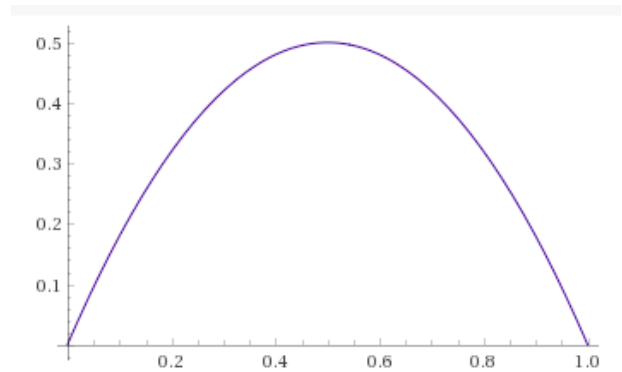
$$-2r x_n + r = 0$$

o que nos leva a $x_n^* = \frac{1}{2}$. Note que nesse ponto o valor da função vale: $f(x_n^*) = -\frac{r}{4} + \frac{r}{2} = \frac{r}{4}$

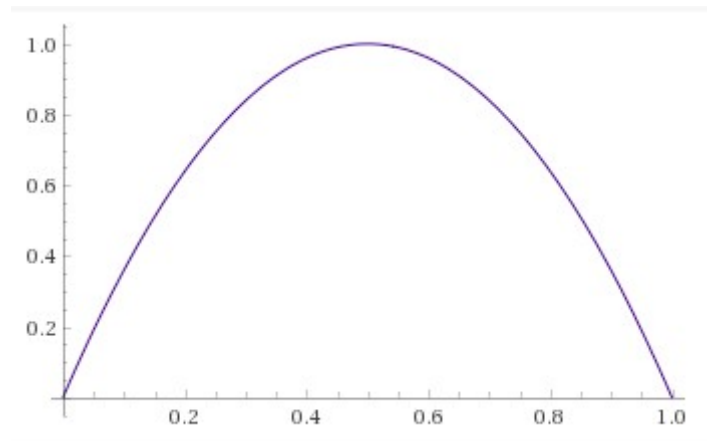
Note também que como $c = 0$, $f(0) = 0$, ou seja, a parábola passa pela origem. Note ainda que $f(1) = 0$, ou seja a parábola corta o eixo x no ponto $x = 1$. De forma gráfica temos para alguns valores de r as seguintes parábolas:



$r = 1$



$r = 2$



$r = 4$

Vamos simular várias iterações do método em Python para analisar o comportamento do tamanho da população em função do tempo t . O script em Python a seguir mostra uma implementação computacional do modelo utilizando 100 iterações.

```
import matplotlib.pyplot as plt

# Número de iterações para atingir equilíbrio
MAX = 100

r = float(input('Entre com a constante r: '))
x = float(input('Entre com x0: '))

population = [x]

for i in range(1, MAX):
    x = r*x*(1 - x)
    population.append(x)

print('População no longo prazo: ', population[-1])

# Plota gráfico da população pelo tempo
eixox = list(range(MAX))
plt.figure(1)
plt.plot(eixox, population)
plt.show()
```

Execute o script e veja o que acontece para as entradas a seguir:

- a) $r = 1$ e $x_0 = 0.4$ (extinção)
- b) $r = 2$ e $x_0 = 0.4$ (equilíbrio em 50%)
- c) $r = 2.4$ e $x_0 = 0.6$ (pequena oscilação, mas atinge equilíbrio em 58%)
- d) $r = 3$ e $x_0 = 0.4$ (não há equilíbrio, população oscila, mas em torno de uma média)
- e) $r = 4$ e $x_0 = 0.4$ (comportamento caótico, totalmente imprevisível)

Em seguida, iremos estudar o que acontece com a população de equilíbrio conforme variamos o valor do parâmetro r . A ideia é que no eixo x iremos plotar os possíveis valores de r e no eixo y iremos plotar a população de equilíbrio para aquele valor de r específico. Iremos considerar que a população do equilíbrio é obtida depois de 1000 iterações. O script em Python a seguir mostra a implementação computacional dessa análise.

```

import matplotlib.pyplot as plt
import numpy as np

# Cria um vetor com todos os possíveis valores de r
R = np.linspace(0.5, 4, 20000)

m = 0.5

# Inicializa os eixos x e y vazios
X = []
Y = []
# Loop principal (iterar para todo r em R)
for r in R:
    # Adiciona r no eixo x
    X.append(r)

    # Escolhe um valor aleatório entre 0 e 1
    x = np.random.random()

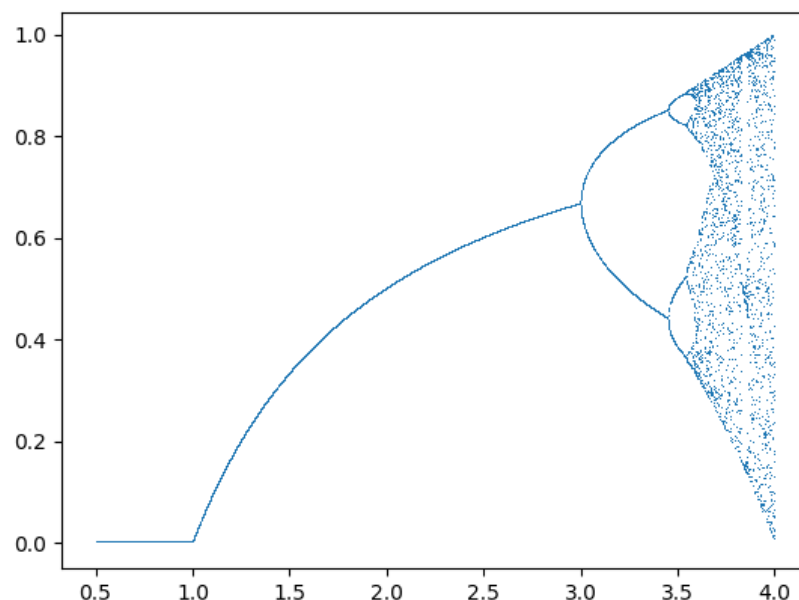
    # Gera população de equilíbrio
    for l in range(1000):
        x = r*x*(1-x)

    Y.append(x)

# Plota o gráfico sem utilizar retas ligando os pontos
plt.plot(X, Y, ls='', marker=',')
plt.show()

```

O gráfico plotado pelo script acima é conhecido como *bifurcation map*. Esse fenômeno da bifurcação ocorre como uma manifestação do comportamento caótico da população de equilíbrio para valores de r maiores que 3. Na prática, o que temos é que para um valor de $r = 3.49999$, a população de equilíbrio é muito diferente daquela obtida para $r = 3.50000$ por exemplo. Pequenas perturbações no parâmetro r causam um efeito devastador na população de equilíbrio. Esse é o lema da teoria do caos, que pode ser parafraseado pela célebre sentença: o simples bater de asas de uma borboleta pode levar ao surgimento de um furacão, conhecido também como o efeito borboleta.



Uma das propriedades do caos é que é possível encontrar ordem e padrões em comportamentos caóticos. Por exemplo, a seguir iremos desenvolver um script em Python para plotar uma sequência de populações, começando de uma população inicial arbitrária e utilizando o valor de $r = 3.99$.

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

r = 3.99

x = np.random.random()      # a população é x minúsculo!
X = [x]                     # a lista é X maiúsculo!

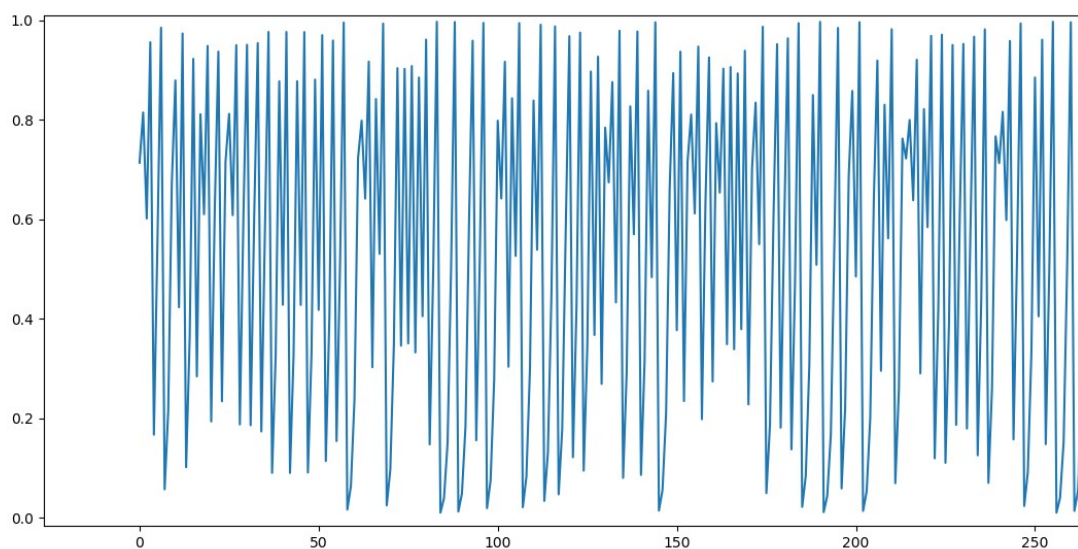
for i in range(1000):
    x = r*x*(1 - x)
    X.append(x)

# Plota o gráfico da sequência
plt.figure(1)
plt.plot(X)
plt.show()

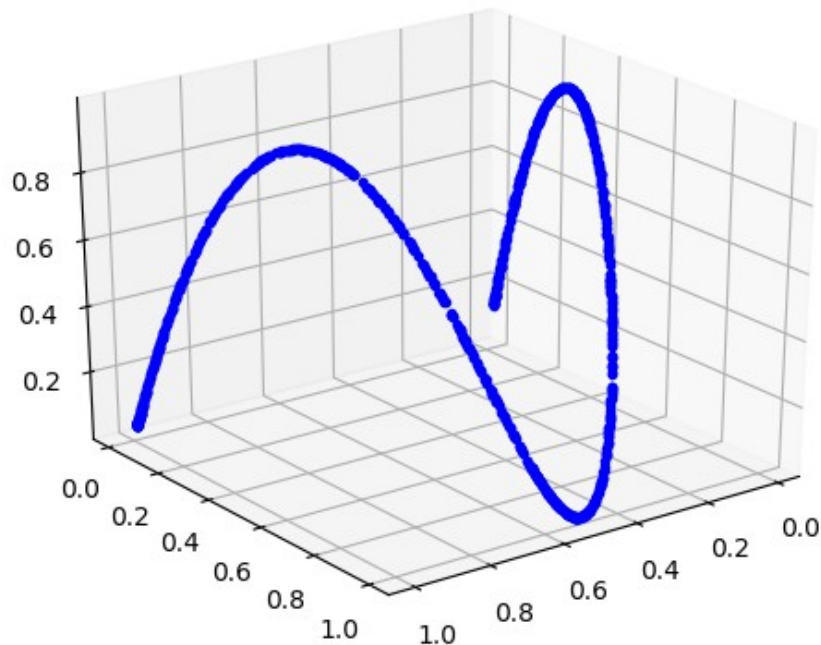
A = X[:len(X)-2]
B = X[1:len(X)-1]
C = X[2:]

#Plota atrator em 3D
fig = plt.figure(3)
ax = fig.add_subplot(111, projection='3d')
ax.plot(A, B, C, '.', c='red')
plt.show()
```

Note que o gráfico mostrado na figura 1 parece o de uma sequência totalmente aleatória.



A seguir, iremos plotar cada subsequência (x_n, x_{n+1}, x_{n+2}) como um ponto no \mathbb{R}^3 . Na prática, isso significa que no eixo X iremos plotar a sequência original, no eixo Y iremos plotar a sequência deslocada de uma unidade e no eixo Z iremos plotar a sequência deslocada de duas unidades. Qual será o gráfico formado? Se de fato a sequência for completamente aleatória, nenhum padrão deverá ser observado, apenas pontos dispersos aleatoriamente pelo espaço. Mas, surpreendentemente, temos a formação do seguinte padrão, conhecido como o atrator do modelo.



Podemos repetir a análise anterior, mas agora plotando o ponto (x_n, x_{n+1}) no plano. Conforme discutido anteriormente, vimos que $x_{n+1} = f(x_n)$ é uma função quadrática, ou melhor, uma parábola. O experimento prático apenas comprova a teoria. Note que o gráfico obtido pelo script a seguir é exatamente a parábola. Conforme a teoria, note que o ponto de máximo ocorre em $x_n = 0.5$, e o valor da função nesse ponto, $f(x_n)$, é praticamente 1 ($r/4 = 3.99/4$).

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

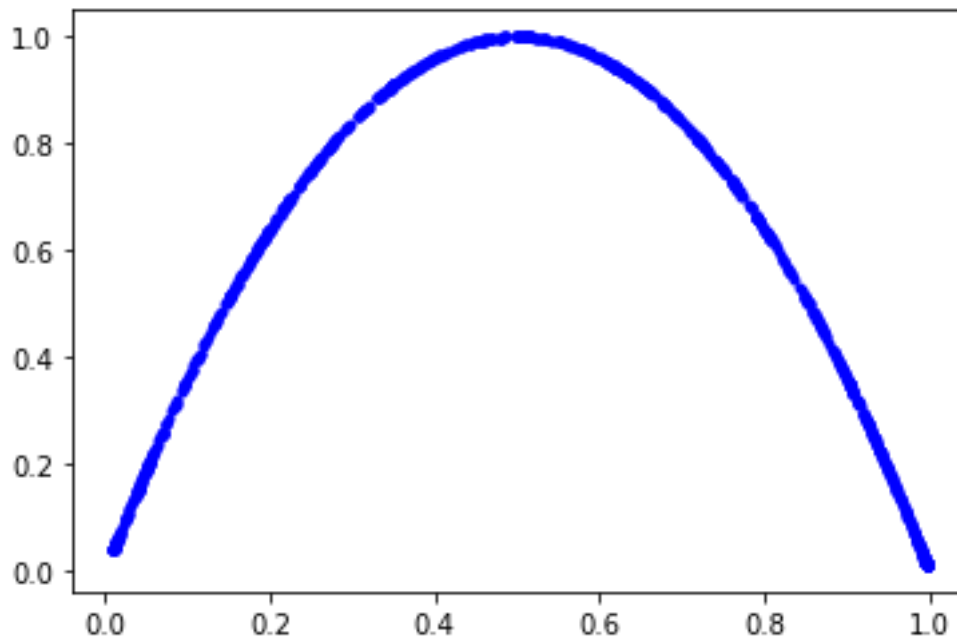
r = 3.99
x = np.random.random()          # a população é x minúsculo!
X = [x]                          # a lista é X maiúsculo!

for i in range(1000):
    x = r*x*(1 - x)
    X.append(x)

# Plota o gráfico da sequência
plt.figure(1)
plt.plot(X)
plt.show()
```

```
A = X[:len(X)-1]
B = X[1:len(X)]

#Plota atrator em 3D
plt.figure(2)
plt.plot(A, B, '.', c='blue')
plt.show()
```



Interessante, não é mesmo? Dentro do caos, há ordem. Muitos fenômenos que observamos no mundo real parecem ser aleatórios, mas na verdade exibem comportamento caótico. A pergunta que fica é justamente essa: como distinguir um sistema aleatório de um sistema caótico? Como identificar os padrões que nos permitem enxergar a ordem em um sistema caótico? Para responder a esse questionamento precisamos mergulhar fundo na matemática dos sistemas complexos e da teoria do caos.

Sistemas Lineares

Muitos problemas decorrentes de várias áreas da ciência podem ser expressos na forma de sistemas lineares de equações. Encontrar soluções para tais sistemas é portanto fundamental para resolver tais problemas. Um sistema linear de equações possui forma geral dada por:

$$\begin{array}{cccccc} a_{11}x_1 & + & a_{12}x_2 & + \cdots + & a_{1n}x_n & = b_1 \\ a_{21}x_1 & + & a_{22}x_2 & + \cdots + & a_{2n}x_n & = b_2 \\ \vdots & & \vdots & & \vdots & \vdots \\ a_{m1}x_1 & + & a_{m2}x_2 & + \cdots + & a_{mn}x_n & = b_m. \end{array}$$

onde x_1, x_2, \dots, x_n são variáveis, $a_{11}, a_{12}, a_{13}, \dots, a_{mn}$ são coeficientes do sistema (conhecidos) e b_1, b_2, \dots, b_m são termos constantes. Note que é possível expressar o sistema linear na forma matricial como $A\vec{x} = \vec{b}$, onde

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}$$

Uma solução para um sistema linear de equações consiste em uma atribuição de valores para x_1, x_2, \dots, x_n tal que todas as equações sejam simultaneamente satisfeitas. Um sistema linear de equações pode se comportar de 3 formas distintas:

1. O sistema possui infinitas soluções
2. O sistema possui uma única solução.
3. O sistema não admite solução.

Obs: Em geral, um sistema com menos equações que incógnitas (sistema subdeterminado) possui infinitas soluções (mas pode não ter solução em certos casos).

Em geral, um sistema com o mesmo número de equações e incógnitas possui uma única solução.

Em geral, um sistema com mais equações do que incógnitas (sistema sobredeterminado) não admite solução.

O método da eliminação de Gauss

Este método consiste em aplicar sucessivas operações elementares em um sistema linear, para o transformar num sistema de mais fácil resolução, tendo este as mesmas soluções que o original.

Operações elementares

Existem três operações básicas que podem ser aplicadas a qualquer tipo de sistema linear, sem que se altere as soluções dos mesmos:

1. Somar a uma linha um múltiplo de outra linha.
2. Trocar duas linhas entre si.
3. Multiplicar todos os elementos de uma linha por uma constante não-nula.

Usando essas operações, uma matriz sempre pode ser transformada em uma matriz triangular superior (forma escalonada) e, posteriormente, ser posta em sua forma escalonada reduzida. Esta forma final, por sua vez, é única e independente da sequência de operações de linha usadas, sendo mais fácil de resolver que a versão original da matriz. Também cabe ressaltar que estas operações elementares são reversíveis, sendo possível retornar ao sistema inicial aplicando a sequência de operações novamente, mas na ordem inversa.

Problema geral

Deseja-se, a partir da utilização de operações de linha, converter uma matriz a sua forma escalonada reduzida, e assim, resolver mais facilmente o sistema de equações associado àquela matriz. Para este fim, utilizamos o método de Eliminação de Gauss, sendo este composto por duas fases:

a) Fase de eliminação: cujo objetivo é empregar operações elementares na matriz aumentada, a fim de obter uma correspondente a um sistema triangular superior.

b) Fase de substituição: começa-se resolvendo a última equação, cuja solução é substituída na penúltima, a qual resolve-se na penúltima variável, e assim até obter-se a solução final.

Algoritmo

Seja $A\vec{x}=\vec{b}$ um sistema linear. O método da eliminação de Gauss para se encontrar a solução do sistema consiste nas seguintes etapas:

Etapa 1: Obter a matriz aumentada $[A|b]$ que representa o sistema de equações

Etapa 2: Transformar a matriz aumentada $[A|b]$ em uma matriz $[\bar{A}|\bar{b}]$ onde \bar{A} é uma matriz triangular superior

Etapa 3: Resolver o sistema linear $[\bar{A}|\bar{b}]$ da etapa 2 por substituição regressiva

A seguir o algoritmo é detalhado para um caso genérico.

Considere o sistema linear de 3 equações a seguir:

Etapa 1:

$$a_{11}x_1 + a_{12}x_2 + a_{13}x_3 = b_1 \quad (L_1)$$

$$a_{21}x_1 + a_{22}x_2 + a_{23}x_3 = b_2 \quad (L_2)$$

$$a_{31}x_1 + a_{32}x_2 + a_{33}x_3 = b_3 \quad (L_3)$$

A matriz aumentada do sistema é:

$$[A|b]^{(0)} = \left[\begin{array}{ccc|c} a_{11} & a_{12} & a_{13} & b_1 \\ a_{21} & a_{22} & a_{23} & b_2 \\ a_{31} & a_{32} & a_{33} & b_3 \end{array} \right] \text{ Etapa 2:}$$

Fase 1

Deseja-se zerar todos os elementos da primeira coluna abaixo da diagonal principal. Assim, sendo o pivô $a_{11} \neq 0$, definem-se as constantes $k = \frac{a_{21}}{a_{11}}$ e $w = \frac{a_{31}}{a_{11}}$ e faz-se as seguintes operações:

$$L_2^{(1)} \leftarrow L_2 - k \cdot L_1$$

$$L_3^{(1)} \leftarrow L_3 - w \cdot L_1$$

obtendo-se

$$[A|b]^{(1)} = \left[\begin{array}{ccc|c} a_{11}^{(1)} & a_{12}^{(1)} & a_{13}^{(1)} & b_1^{(1)} \\ 0 & a_{22}^{(1)} & a_{23}^{(1)} & b_2^{(1)} \\ 0 & a_{32}^{(1)} & a_{33}^{(1)} & b_3^{(1)} \end{array} \right]$$

Fase 2

Agora deseja-se zerar todos os elementos da segunda coluna abaixo da diagonal principal. Sendo o pivô $a_{22} \neq 0$, define-se uma nova constante $v = \frac{a_{32}}{a_{22}}$. Realizando a operação

$$L_3^{(2)} \leftarrow L_3^{(1)} - v \cdot L_2^{(1)}$$

obtem-se

$$[A|b]^{(2)} = \left[\begin{array}{ccc|c} a_{11}^{(2)} & a_{12}^{(2)} & a_{13}^{(2)} & b_1^{(2)} \\ 0 & a_{22}^{(2)} & a_{23}^{(2)} & b_2^{(2)} \\ 0 & 0 & a_{33}^{(2)} & b_3^{(2)} \end{array} \right]$$

Etapa 3: Resolve-se o sistema anterior por substituição regressiva

$$x_3 = b_3^{(2)} / a_{33}^{(2)}, \quad a_{33}^{(2)} \neq 0$$

$$x_2 = (b_2^{(2)} - (a_{23}^{(2)} x_3)) / a_{22}^{(2)}$$

$$x_1 = (b_1^{(2)} - (a_{12}^{(2)} x_2) - a_{13}^{(2)} x_3) / a_{11}^{(2)}$$

Assim, encontra-se a solução $\vec{x} = [x_1, x_2, x_3]$ que é a mesma solução do sistema $[A|b]$

Obs: O método da eliminação de Gauss só poderá ser usado para resolver sistemas lineares associados a matrizes escalonadas reduzidas com elementos das suas diagonais principais não-nulos, ou seja, $a_{11}^{(1)}, a_{22}^{(2)}, a_{33}^{(3)}, \dots, a_{nn}^{(n)} \neq 0$

A seguir veremos e discutiremos uma implementação em Python do algoritmo em questão.

```

import numpy as np

#####
# implementa o método da eliminação de Gauss
# problema: pode sofrer de instabilidade numérica
#####
def eliminacao_gauss(A, b):
    n = A.shape[0]          # número de linhas/colunas de A
    x = np.zeros(n)

    #####
    # Primeira etapa: transformar sistema na forma triangular
    #####
    # varre as colunas de A da primeira ate a penúltima
    # lembrar que devemos parar uma coluna antes do final
    for i in range(n-1):
        # dentro de cada coluna varre as linhas abaixo da diagonal
        for j in range(i+1, n):
            # calcula multiplicador usado para zerar variavel
            m = A[j][i]/A[i][i]
            # multiplica equação i por -m e soma com equação j
            # somente de i para frente (pivô)
            for k in range(i, n):
                A[j][k] = A[j][k] - m*A[i][k]
            b[j] = b[j] - m*b[i]

    #####
    # Segunda etapa: resolver sistema triangular de equações
    #####
    # percorre colunas (de tras para frente)
    for j in range(n-1,-1,-1):
        x[j] = b[j]/A[j][j]
        # percorre elementos dentro de uma coluna (baixo para cima)
        for i in range(j-1,-1,-1):
            b[i] = b[i] - A[i][j]*x[j]

    return x

# imprime o sistema linear na tela
def imprime_sistema(A, b):
    print("Sistema Linear:")
    for i in range(A.shape[0]):
        linha = []
        for j in range(A.shape[1]):
            linha.append('{0}*x{1}'.format(A[i, j], j + 1))
        print(" + ".join(linha), "=", b[i])
    print() # pula linha

# Início do script
MAX_ITERATION = 100

# inicializa a matriz A (4 equações)
A = np.array([[10., -1., 2., 0.],
              [-1., 11., -1., 3.],
              [2., -1., 10., -1.],
              [0.0, 3., -1., 8.]])

```

```
# inicializa o vetor b
b = np.array([6., 25., -11., 15.])

# imprime sistema na tela
imprime_sistema(A, b)

# resolve sistema linear
x = eliminacao_gauss(A, b)
print(x)
```

Algoritmos iterativos

Existem métodos analíticos para resolver um sistema linear obtendo sua solução exata (pelo menos teoricamente), combinando suas linhas para gerar um sistema linear equivalente e mais simples de ser resolvido. Um exemplo é o método da Eliminação de Gauss. Há, entretanto, algumas dificuldades quando devemos calcular a solução de um sistema de grandes proporções. O grande volume de operações de multiplicação e de divisões agrega, a cada passo, erros de truncamento que, somados ao longo do processo, podem nos levar a soluções absolutamente falsas. Assim sendo, a grande simplicidade de tais métodos perde-se na possibilidade do acúmulo incontrolável de erros.

A maneira de se superar essa dificuldade é definir novas famílias de métodos, agora iterativos, à semelhança do que é feito na determinação de raízes de uma função algébrica $f(x)$, isto é, na solução de $f(x) = 0$. Vamos, pois, generalizar o procedimento usado nesse caso, passando de uma função algébrica a uma equação matricial $f(\vec{x}) = 0$ da forma:

$$f(\vec{x}) = A\vec{x} - \vec{b}$$

onde A é uma matriz $n \times n$, \vec{x} é o vetor solução a ser obtido e \vec{b} é o vetor de termos constantes. Da mesma forma usada para a solução da equação $f(x) = 0$ transformamos a equação anterior na sua forma iterativa (relação de recorrência):

$$\vec{x}^{(k)} = \Phi(\vec{x}^{(k-1)}) \quad \text{com} \quad \Phi(\vec{x}) = H\vec{x} + \vec{k}$$

onde H é chamada de matriz de iteração do método proposto.

O método de Jacobi

Trata-se de um método numérico e iterativo para a resolução de sistemas lineares. Seja $A\vec{x} = \vec{b}$ um sistema linear de n equações e n incógnitas, onde

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}.$$

Então, a matriz A pode ser decomposta em uma matriz diagonal D e uma matriz resto R:

$$A = D + R \quad \text{where} \quad D = \begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{bmatrix} \quad \text{and} \quad R = \begin{bmatrix} 0 & a_{12} & \cdots & a_{1n} \\ a_{21} & 0 & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & 0 \end{bmatrix}.$$

Então, podemos escrever que:

$$(D+R)\vec{x}=\vec{b}$$

Expandindo o produto temos:

$$D\vec{x}+R\vec{x}=\vec{b}$$

Isolando os termos:

$$D\vec{x}=\vec{b}-R\vec{x}$$

o que nos leva a:

$$\vec{x}=D^{-1}(\vec{b}-R\vec{x})$$

Assim, podemos interpretar a equação acima como uma relação de recorrência em que dado um vetor \vec{x}_0 inicial computa a solução através da iteração:

$$\vec{x}_{k+1}=D^{-1}(\vec{b}-R\vec{x}_k)$$

A seguir veremos um exemplo de função que implementa o método de Jacobi em Python utilizando a biblioteca Numpy.

```
def imprime_sistema(A, b):
    print("Sistema Linear:")
    for i in range(A.shape[0]):
        linha = []
        for j in range(A.shape[1]):
            linha.append('{0}*x{1}'.format(A[i, j], j + 1))
        print(" + ".join(linha), "=", b[i])
    print() # pula linha

def jacobi(A, b):
    x = np.zeros(len(b))
    for k in range(MAX_ITERATION):
        print('Solução atual: ', x)
        x_novo = np.zeros(len(x))
        D = np.diag(np.diag(A))
        R = A - D
        x_novo = np.dot(np.linalg.inv(D), b - np.dot(R, x))
        if np.allclose(x, x_novo, atol=10**(-10)):
            break
        x = x_novo
    return x
```

```

# Início do script
MAX_ITERATION = 100

# inicializa a matriz A (4 equações)
A = np.array([[10., -1., 2., 0.],
              [-1., 11., -1., 3.],
              [2., -1., 10., -1.],
              [0.0, 3., -1., 8.]])

# inicializa o vetor b
b = np.array([6., 25., -11., 15.])

# imprime sistema na tela
imprime_sistema(A, b)

# resolve sistema linear
x = jacobi(A, b)
print(x)

```

O método de Gauss-Seidel

O método de Gauss-Seidel é muito similar ao algoritmo de Jacobi. A diferença está na forma em que a matriz A é decomposta:

$$A = L_* + U \quad \text{where} \quad L_* = \begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ a_{21} & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}, \quad U = \begin{bmatrix} 0 & a_{12} & \cdots & a_{1n} \\ 0 & 0 & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix}.$$

sendo L_* a matriz triangular inferior e U a matriz triangular superior estrita, isto é, sem a diagonal principal. Da mesma forma podemos escrever que:

$$(L_* + U)\vec{x} = \vec{b}$$

Expandindo o produto temos:

$$L_*\vec{x} + U\vec{x} = \vec{b}$$

Isolando os termos:

$$L_*\vec{x} = \vec{b} - U\vec{x}$$

o que nos leva a seguinte relação de recorrência:

$$\vec{x}_{k+1} = L_*^{-1}(\vec{b} - U\vec{x}_k)$$

A seguir veremos um exemplo de função que implementa o método de Gauss-Seidel em Python utilizando a biblioteca Numpy.

```

# implementa o método de Gauss_Siedel
def gauss_siedel(A, b):
    x = np.zeros(len(b))
    for k in range(MAX_ITERATION):
        print('Solução atual: ', x)
        x_novo = np.zeros(len(x))
        L = np.tril(A)
        U = A - L
        # computa relação de recorrência
        x_novo = np.dot(np.linalg.inv(L), b - np.dot(U, x))
        # verifica se a diferença entre x e x_novo é desprezível
        if np.allclose(x, x_novo, atol=10**(-10)):
            break
        x = x_novo

    return x

```

Métodos iterativos: convergência

Os métodos iterativos de Jacobi e Gauss-Seidel podem ser expressos na forma padrão

$$\vec{x}^{(k)} = H \vec{x}^{(k-1)} + \vec{k}$$

Jacobi: $\vec{x}^{(n+1)} = -D^{-1} R \vec{x}^{(n)} + D^{-1} \vec{b}$

Gauss-Seidel: $\vec{x}^{(n+1)} = -L^{-1} U \vec{x}^{(n)} + L^{-1} \vec{b}$

A partir de uma aproximação inicial $\vec{x}^{(0)}$, fazemos sucessivamente as iterações:

$$\vec{x}^{(1)} = H \vec{x}^{(0)} + \vec{k}$$

$$\vec{x}^{(2)} = H \vec{x}^{(1)} + \vec{k}$$

...

$$\vec{x}^{(n)} = H \vec{x}^{(n-1)} + \vec{k}$$

Da mesma forma que no problema dos zeros da função $f(x)$ precisamos de condições gerais de convergência para a aplicação de métodos iterativos a sistemas lineares. Vejamos, inicialmente, como se comporta a propagação do erro nesses casos. Seja \vec{u} a solução exata do sistema dado. Então, por definição (\vec{u} é um ponto fixo do operador, ou seja, aplicar a iteração não muda nada):

$$\vec{u} = H \vec{u} + \vec{k}$$

Assim, o erro no passo k é dado por:

$$e^{(k)} = \vec{u} - \vec{x}^{(k)} = (H \vec{u} + \vec{k}) - (H \vec{x}^{(k-1)} + \vec{k}) = H \vec{u} - H \vec{x}^{(k-1)} = H (\vec{u} - \vec{x}^{(k-1)}) = H \vec{e}^{(k-1)}$$

Aplicando o mesmo raciocínio, podemos expandir o termo do erro no passo $k-1$:

$$H e^{(k-1)} = H (\vec{u} - \vec{x}^{(k-1)}) = H [(H \vec{u} + \vec{k}) - (H \vec{x}^{(k-2)} + \vec{k})] = H [H \vec{u} - H \vec{x}^{(k-2)}] = H H (\vec{u} - \vec{x}^{(k-2)}) = H H \vec{e}^{(k-2)}$$

de modo que fazendo todas as recursões de k até zero, temos:

$$\vec{e}^{(k)} = H H \dots H (\vec{u} - \vec{x}^{(0)}) = H^k (\vec{u} - \vec{x}^{(0)}) = H^k \vec{e}^{(0)}$$

onde $\vec{e}^{(0)}$ é o erro inicial.

Portanto, o acúmulo de erro em cda passo cresce com a potência da matriz H , ou seja, se cometemos um erro inicial $\vec{e}^{(0)}$, o erro final na k -ésima iteração será $H^k \vec{e}^{(0)}$. Assim, se H^k tender a zero podemos esperar que $\vec{e}^{(0)}$ diminua cada vez mais. Iremos examinar a condição necessária para que $\lim_{k \rightarrow \infty} H^k = 0$.

Decomposição em autovalores e autovetores (Eigendecomposition)

Seja A é uma matriz quadrada $n \times n$ com n autovetores. Então, A pode ser decomposta em:

$$A = Q \Lambda Q^{-1} \quad \text{onde}$$

$$Q = \begin{bmatrix} | & | & \dots & | \\ \vec{v}_1 & \vec{v}_2 & \dots & \vec{v}_n \\ | & | & \dots & | \end{bmatrix} \quad \text{é a matriz em que cada coluna é um dos } n \text{ autovetores e}$$

$$\Lambda = \begin{bmatrix} \lambda_1 & 0 & \dots & 0 \\ 0 & \lambda_2 & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & \lambda_n \end{bmatrix} \quad \text{é a matriz diagonal dos autovalores}$$

Então, aplicando essa decomposição a k -ésima potência da matriz de iteração, H^k , temos:

$$H^k = (Q \Lambda Q^{-1})(Q \Lambda Q^{-1}) \dots (Q \Lambda Q^{-1}) = Q \Lambda^k Q^{-1}$$

uma vez que para todos os termos intermediários $Q Q^{-1} = I$. Assim, se $\lim_{k \rightarrow \infty} \Lambda^k = 0$ teremos que $\lim_{k \rightarrow \infty} H^k = 0$. Como

$$\Lambda^k = \begin{bmatrix} \lambda_1^k & 0 & \dots & 0 \\ 0 & \lambda_2^k & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & \lambda_n^k \end{bmatrix}$$

o limite acima é satisfeito somente se $|\lambda_i| < 1, i=1,2,\dots,n$. Portanto, os autovalores devem ser, em módulo, menores que 1. Isso é equivalente a dizer que o raio espectral de H , que é o maior autovalor de H , deve ser menor que 1, ou seja, $|\rho(H)| < 1$.

Portanto, no método de Jacobi devemos ter $|\rho(-D^{-1}R)| < 1$ e no método de Gauss-Seidel devemos ter $|\rho(-L^{-1}U)| < 1$.

Exemplo: Suponha um sistema linear em que a matriz dos coeficientes é dada por:

$$A = \begin{bmatrix} 1 & 2 & -2 \\ 1 & 1 & 1 \\ 2 & 2 & 1 \end{bmatrix}$$

Determine se haverá convergência:

a) no método de Jacobi

$$D = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad R = \begin{bmatrix} 0 & 2 & -2 \\ 1 & 0 & 1 \\ 2 & 2 & 0 \end{bmatrix} \quad H = -D^{-1}R = \begin{bmatrix} 0 & -2 & 2 \\ -1 & 0 & -1 \\ -2 & -2 & 0 \end{bmatrix}$$

$$|\lambda_1| = 1.23321 \times 10^{-5}$$

$$|\lambda_2| = 1.23321 \times 10^{-5} \quad \text{Ok, iteração converge!}$$

$$|\lambda_3| = 1.23322 \times 10^{-5}$$

b) no método de Gauss-Seidel

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 2 & 2 & 1 \end{bmatrix} \quad U = \begin{bmatrix} 0 & 2 & -2 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} \quad H = -L^{-1}U = \begin{bmatrix} 0 & -2 & 2 \\ 0 & 2 & 3 \\ 0 & 0 & 2 \end{bmatrix}$$

$$|\lambda_1| = 0$$

$$|\lambda_2| = 2$$

$$|\lambda_3| = 2$$

Exercício: Uma matriz de Hilbert é definida como sendo a matriz quadrada H tal que o elemento $h[i, j]$ é dado por:

$$h[i, j] = \frac{1}{i+j-1}$$

Faça um script em Python que gere matrizes de Hilbert H de ordem $n \times n$ (onde n é um parâmetro conhecido pelo algoritmo) e gere o vetor b em que cada elemento $b[i]$ corresponde a soma dos elementos da i -ésima linha da matriz H .

a) Utilizando os critérios de convergência analise o que acontece quando $n = 3$ e $n = 30$.

b) Aplique os algoritmos de Jacobi e Gauss-Seidel para resolver um sistema linear definido por uma matriz de Hilbert de ordem 3 e ordem 30. Compare os resultados obtidos com os 2 métodos. O que acontece?

Convolução e filtragem linear

Para que possamos entender a operação de convolução, primeiramente é necessário compreender o que são sinais e sistemas.

O que é um sinal?

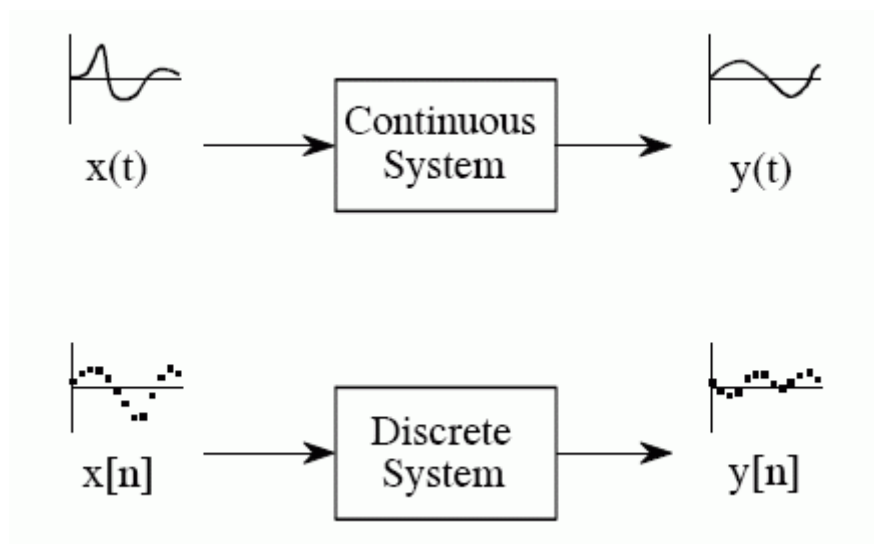
- Descrição de como um parâmetro varia no tempo (temperatura no dia, variação do dólar no mês, valor de uma ação na bolsa ao longo de horas, número de mortos pelo COVID-19 nos dias, etc.)

O que é um sistema?

- Qualquer processo que produz um sinal de saída em resposta a um sinal de entrada (tipicamente um filtro pois realiza um processo de filtragem no sinal de entrada)

Um sinal $f(t)$ é dito contínuo o domínio t é o conjunto dos números reais (há infinitas amostras)

Um sinal $f(n)$ é dito discreto se o domínio n é o conjunto dos números naturais (há um número finito de amostras)



Usaremos a notação $y(t) = H[x(t)]$ para dizer que o sistema H produz como saída o sinal $y(t)$ ao receber como entrada o sinal $x(t)$.

Um sistema é linear se satisfaz as propriedades de homogeneidade e aditividade, ou seja:

$$H[a x_1(t) + b x_2(t)] = a H[x_1(t)] + b H[x_2(t)] = a y_1(t) + b y_2(t)$$

Um sistema é invariante no tempo quando seu funcionamento não se altera ao longo do tempo. Matematicamente, para qualquer entrada $x(t)$ e deslocamento t_0 :

$$y(t) = H[x(t)] \rightarrow y(t - t_0) = H[x(t - t_0)]$$

ou seja, a mesma entrada em tempos distintos provoca a mesma saída apenas deslocada no tempo.

Um sistema linear e invariante no tempo (SLIT) é completamente caracterizado por sua resposta impulsiva $h(t)$, que define qual é a resposta do sistema a um impulso unitário.

No caso contínuo a relação entre um sinal de entrada $x(t)$, a resposta impulsiva do sistema $h(t)$ (comumente chamada de filtro) e a saída do sistema $y(t)$ é dada pela integral de convolução:

$$y(t) = x(t) * h(t) = \int_{-\infty}^{\infty} x(\tau) h(t - \tau) d\tau$$

ou seja, a integral acima nos permite calcular a saída de qualquer SLIT para qualquer entrada $x(t)$. A seguir veremos alguns exemplos em que é possível calcular analiticamente a saída para várias entradas em vários SLIT's, cada um definido pela sua resposta impulsiva $h(t)$.

Ex: Dado a entrada $x(t)=u(t)$, calcule a saída do SLIT definido por $h(t)=e^{-t}u(t)$, onde $u(t)$ denota a função degrau unitário, ou seja, $u(t)=0$ se $t<0$ e $u(t)=1$ se $t\geq 0$. Em outras palavras, quem é $y(t)$?

Gráfico de $u(t)$

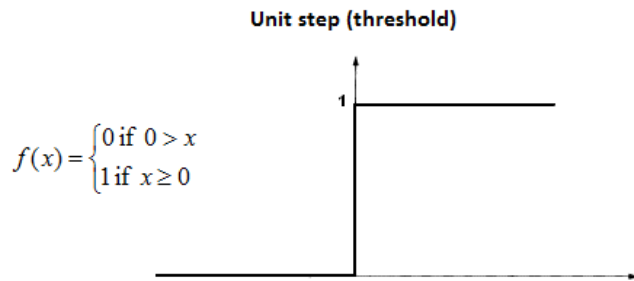
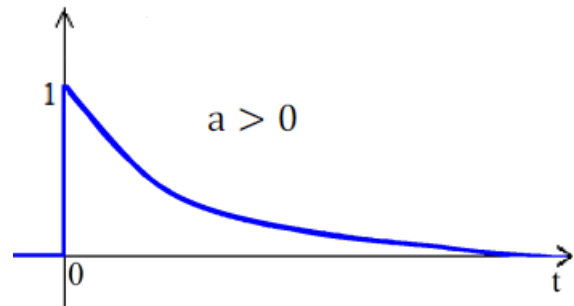


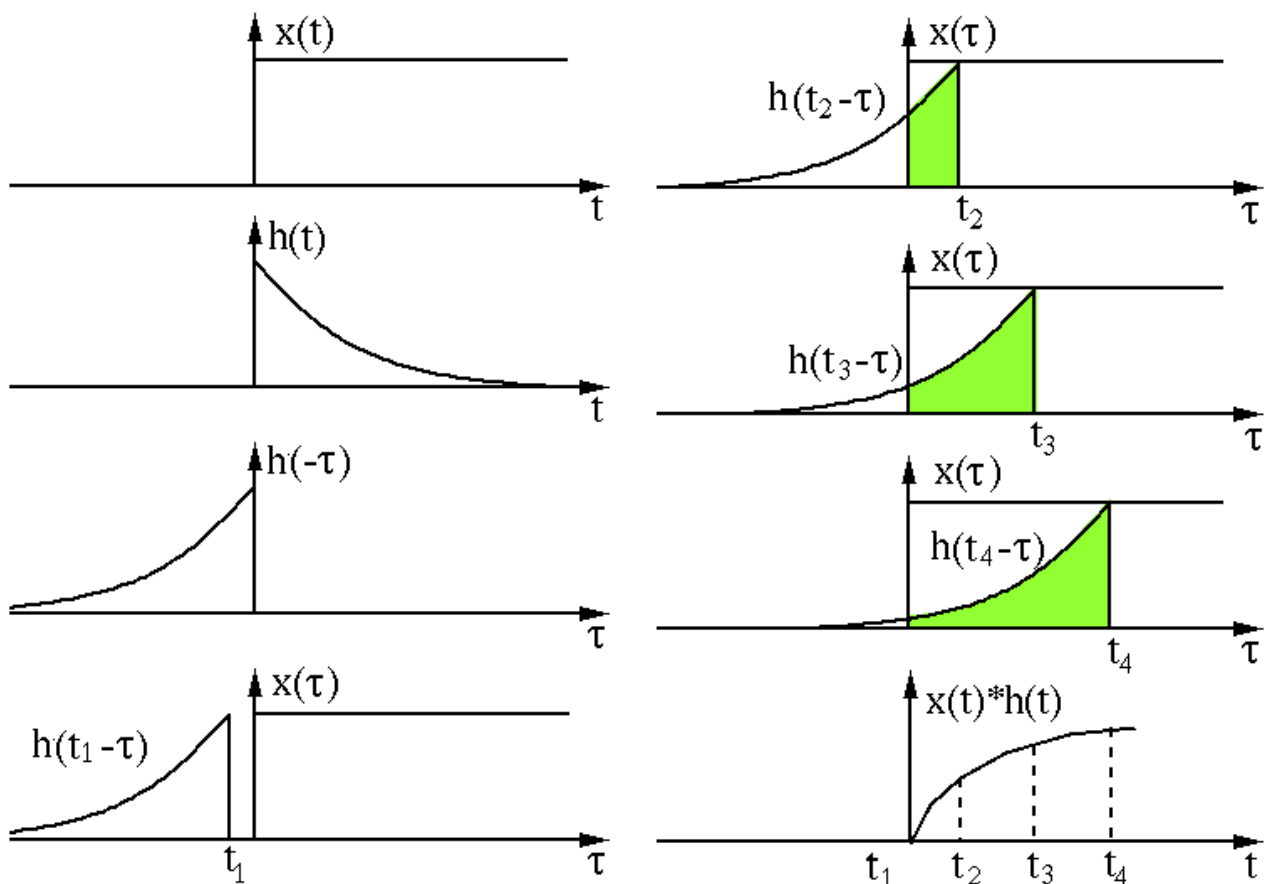
Gráfico de $h(t)=e^{-t}u(t)$



Devemos obter $y(t)$ como a solução da integral:

$$y(t) = \int_{-\infty}^{\infty} x(\tau)h(t-\tau)d\tau$$

Para isso, devemos visualizar o gráfico de $h(t-\tau)$ como função de τ . Primeiramente, note que o gráfico de $h(-\tau)$ é a versão refletida de $h(\tau)$ no eixo y. Note também que para valores negativos de t (t_1) a função rebatida se desloca para a esquerda enquanto que para valores positivos de t (t_2, t_3, t_4) a função rebatida se desloca para a direita, conforme ilustra a figura a seguir:



A integral de convolução na verdade computa portanto a área da intersecção quando se passa a versão rebatida do filtro por sobre o sinal $x(t)$, como função do deslocamento t . Conforme

$h(t-\tau)$ desliza por sobre $x(t)$, a área da intersecção varia. A integral de convolução mede a variação dessa área. A função resultante $y(t)$ retorna o valor da área da intersecção para cada possível valor de t . Note que para valores de t menores que zero, o resultado da convolução é zero. Voltando aos cálculos, temos que $h(-\tau) = e^{-(-\tau)}u(-\tau)$ cujo gráfico está representado na imagem acima. Assim, $h(t-\tau) = e^{-(t-\tau)}u(t-\tau)$, cujos gráficos para diferentes valores de t são apresentados na figura anterior. Portanto, a integral fica:

$$y(t) = \int_{-\infty}^{\infty} x(\tau)h(t-\tau)d\tau = \int_0^t 1 \cdot e^{-(t-\tau)}d\tau = \int_0^t e^{-t}e^{\tau}d\tau = e^{-t} \int_0^t e^{\tau}d\tau = e^{-t}e^{\tau} \Big|_0^t = e^{-t}[e^t - 1] = 1 - e^{-t}$$

Ex2: Dado a entrada $x(t) = e^{-t}u(t)$, calcule a saída do SLIT definido por $h(t) = e^{-t}u(t)$, onde $u(t)$ denota a função degrau unitário, ou seja, $u(t) = 0$ se $t < 0$ e $u(t) = 1$ se $t \geq 0$. Em outras palavras, quem é $y(t)$?

Gráfico de $h(t) = e^{-t}u(t)$

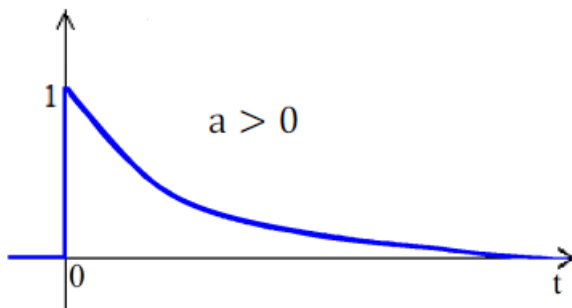
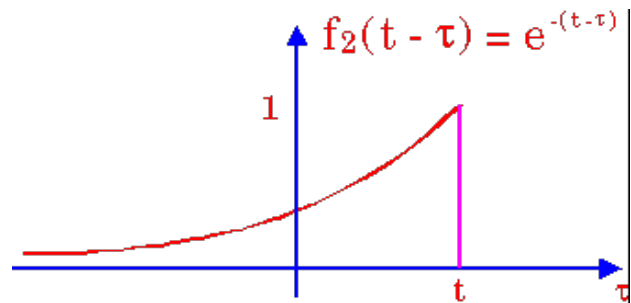


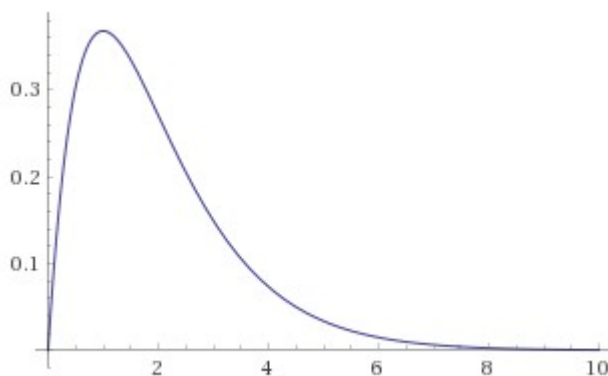
Gráfico de $h(t-\tau) = e^{-(t-\tau)}u(t-\tau)$



Assim, a integral de convolução fica:

$$y(t) = \int_{-\infty}^{\infty} x(\tau)h(t-\tau)d\tau = \int_0^t e^{-\tau} \cdot e^{-(t-\tau)}d\tau = \int_0^t e^{-t}d\tau = e^{-t} \int_0^t d\tau = e^{-t}\tau \Big|_0^t = te^{-t}$$

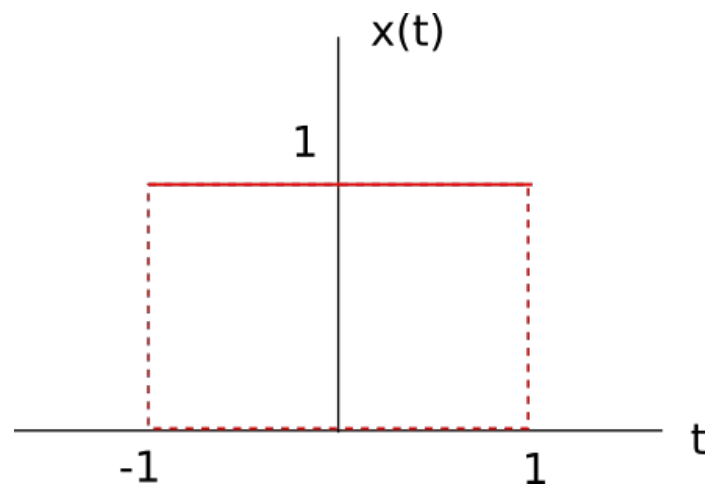
O gráfico da função resultante é:



Ex: Dado que tanto a entrada $x(t)$ quanto $h(t)$ são definidos por um pulso quadrado de largura 2 centrado em zero calcule a saída do SLIT. Em outras palavras, quem é $y(t)$?

$$x(t) = h(t) = \begin{cases} 0, & t < -1 \\ 1, & -1 \leq t \leq 1 \\ 0, & t > 1 \end{cases}$$

O gráfico da função retângulo $x(t)$ e $h(t)$ é ilustrado a seguir:

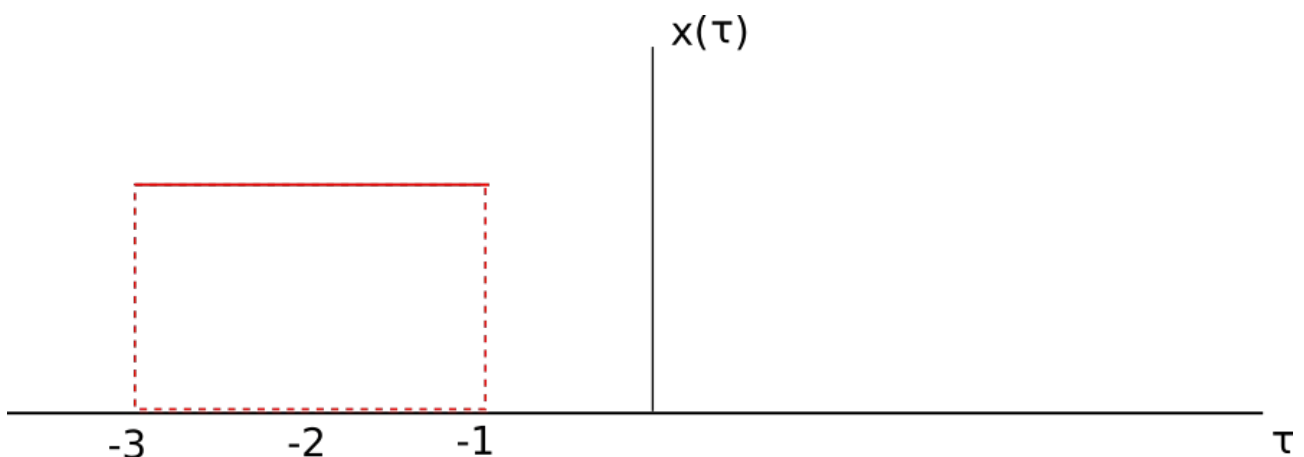


Dessa forma podemos definir $h(t-\tau)$ como:

$$h(t-\tau) = \begin{cases} 0, & t-\tau < -1 \\ 1, & -1 \leq t-\tau \leq 1 \\ 0, & t-\tau > 1 \end{cases} \quad \text{ou seja} \quad h(t-\tau) = \begin{cases} 0, & \tau > t+1 \\ 1, & t-1 \leq \tau \leq t+1 \\ 0, & \tau < t-1 \end{cases}$$

Para valores de t menores que zero o pulso é deslocado para a esquerda e para valores de t maiores que zero o pulso é deslocado para direita. A figura a seguir ilustra a função $h(-2-\tau)$

$$h(-2-\tau) = \begin{cases} 0, & \tau > -1 \\ 1, & -3 \leq \tau \leq -1 \\ 0, & \tau < -3 \end{cases}$$

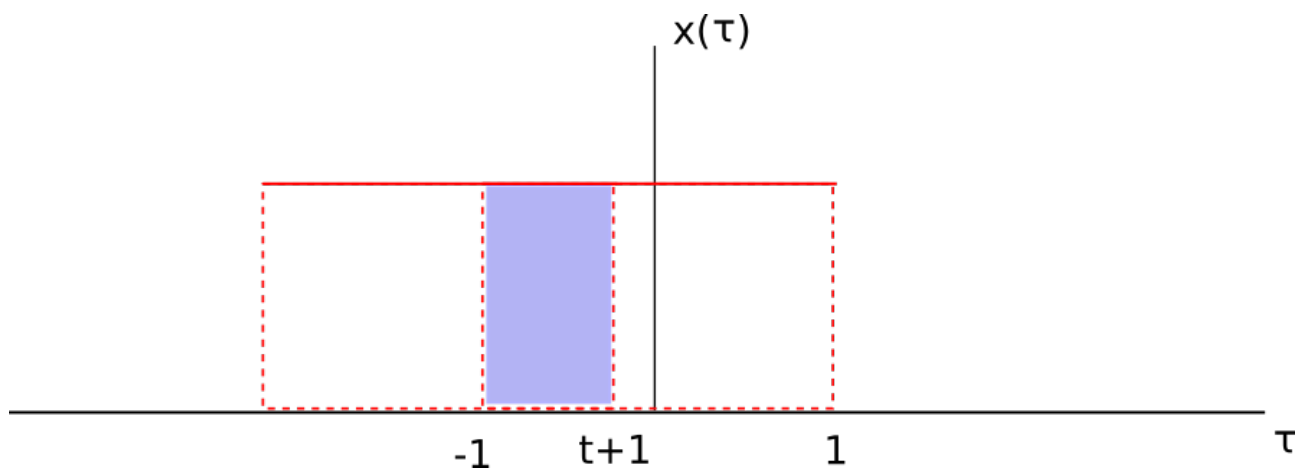


Assim, temos que para $t < -2$ não há intersecção de $h(t)$ com $x(t)$ e portanto a integral resulta em zero. No total temos que considerar 4 casos:

a) para $t \leq -2$ a intersecção é vazia e $y(t) = \int x(\tau)h(t-\tau)d\tau = 0$

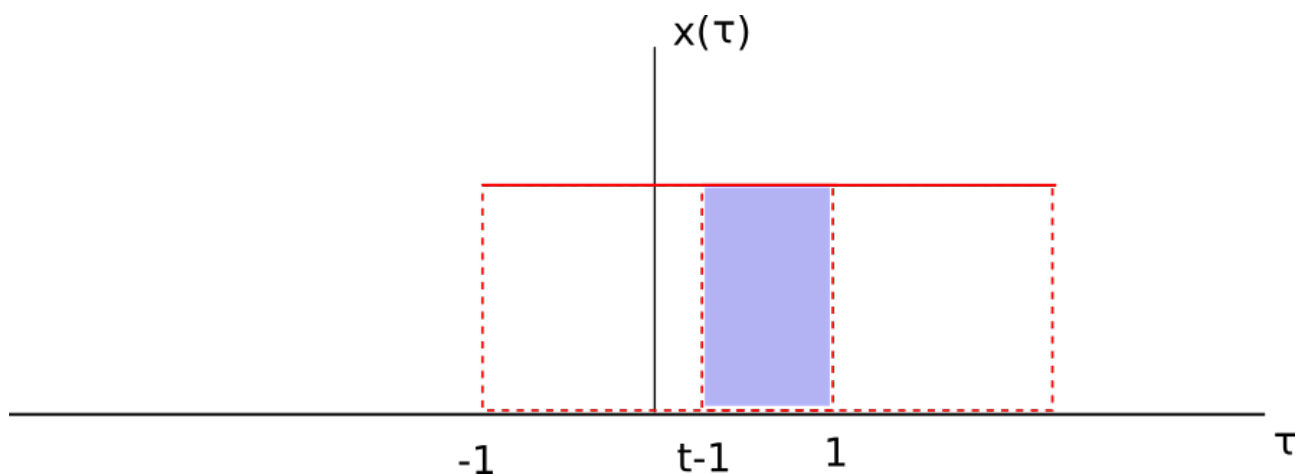
b) para $-2 < t \leq 0$, o pulso $h(t-\tau)$ está entrando em $x(t)$ e

$$y(t) = \int_{-1}^{t+1} x(\tau) h(t-\tau) d\tau = \int_{-1}^{t+1} 1 \cdot 1 d\tau = \tau \Big|_{-1}^{t+1} = (t+1) - (-1) = t+2$$



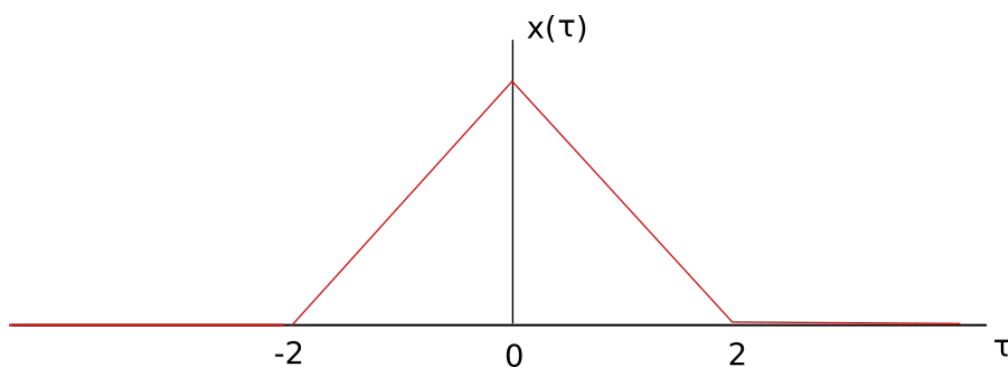
c) para $0 < t \leq 2$, o pulso $h(t-\tau)$ está saindo de $x(t)$ e

$$y(t) = \int_{t-1}^1 x(\tau) h(t-\tau) d\tau = \int_{t-1}^1 1 \cdot 1 d\tau = \tau \Big|_{t-1}^1 = 1 - (t-1) = 2-t$$



d) para $t > 2$, novamente não há intersecção entre os pulsos e $y(t) = \int x(\tau) h(t-\tau) d\tau = 0$

Portanto, a função $y(t)$ resultante é definida por:
$$y(t) = \begin{cases} 0, & t \leq -2 \\ t+2, & -2 < t \leq 0 \\ 2-t, & 0 < t \leq 2 \\ 0, & t > 2 \end{cases}$$



Ex: Calcule o resultado da convolução de $x(t)=e^{-at}u(t)$ e $h(t)=e^{-bt}u(t)$ com $a \neq b$.

Convolução discreta e filtragem

A convolução é a operação matemática que implementa a filtragem de sinais. Por exemplo, suponha que observamos uma série temporal que apresenta um ruído aleatório. Gostaríamos de conseguir filtrar esse sinal, removendo ou atenuando essas perturbações indesejadas. Esse processo de filtragem é definido em termos da operação de convolução do sinal discreto $x[n]$ com um filtro $h[n]$. A ideia geral consiste em ter como entrada 2 vetores: x , que define o sinal e h , que define o filtro a ser aplicado. Em geral, o filtro ou máscara é conhecido a priori e possui tamanho ímpar (3, 5, 7, 9, etc). A expressão matemática da convolução discreta é análoga a sua versão contínua, com a substituição da integral por um somatório:

$$y[n] = \sum_{k=0}^w h[k] x[n-k]$$

onde:

w = número de coeficientes do filtro conhecido como suporte de $h[\]$ (ímpar)

$h[\]$ = filtro

$x[\]$ = sinal de entrada

$y[\]$ = sinal de saída (filtrado)

Observe nesta equação que a soma é realizada nos índices k e não nos índices n . Além disso, esse índice é invertido no filtro h (tem-se $h[-k]$). Desse modo, pode-se resumir as operações envolvidas na convolução pelos seguintes passos:

- i. Rebate-se $h[k]$ para se obter $h[-k]$ (em geral h é simétrico e esse passo não é necessário)
- ii. Desloca-se $h[-k]$ por n_0 amostras a direita se n_0 é positivo.
- iii. Multiplica-se cada elemento $x[k]$ por $h[n_0-k]$
- iv. Somam-se todos os valores da sequência produto para se obter $y[n_0]$
- v. Repetem-se os passos acima para todos os valores possíveis de n de modo a obter $y[n]$

Considere como exemplo o seguinte caso em que o filtro h é simétrico:

$$h[\] = \left[\frac{1}{4}, \frac{1}{2}, \frac{1}{4} \right]$$

$$x[\] = [2, 4, 8, 4, 2]$$

Vamos proceder com o cálculo da convolução pelo primeiro elemento de y :

$$y[0] = h[0]x[0] + h[1]x[-1] + h[2]x[-2] = \frac{1}{4} \cdot 2 + 0 + 0 = \frac{1}{2}$$

Note que como ambos $x[-1]$ e $x[-2]$ não são definidos, eles assumem valor zero.

A seguir, temos o segundo elemento do sinal de saída:

$$y[1] = h[0]x[1] + h[1]x[0] + h[2]x[-1] = 1 + 1 + 0 = 2$$

De forma análoga, podemos computar todo o restante, gerando o sinal de saída $y[]$:

$$y[2]=h[0]x[2]+h[1]x[1]+h[2]x[0]=\frac{1}{4}*8+\frac{1}{2}*4+\frac{1}{4}*2=\frac{9}{2}$$

$$y[3]=h[0]x[3]+h[1]x[2]+h[2]x[1]=\frac{1}{4}*4+\frac{1}{2}*8+\frac{1}{4}*4=6$$

$$y[4]=h[0]x[4]+h[1]x[3]+h[2]x[2]=\frac{1}{4}*2+\frac{1}{2}*4+\frac{1}{4}*8=\frac{9}{2}$$

$$y[5]=h[0]x[5]+h[1]x[4]+h[2]x[3]=\frac{1}{4}*0+\frac{1}{2}*2+\frac{1}{4}*4=2$$

$$y[6]=h[0]x[6]+h[1]x[5]+h[2]x[4]=\frac{1}{4}*0+\frac{1}{2}*0+\frac{1}{4}*2=\frac{1}{2}$$

Portanto, temos que $y[] = \{0.5, 2, 4.5, 6, 4.5, 2, 0.5\}$. Note que como tanto o sinal quanto o filtro são simétricos, o sinal de saída $y[]$ também é. Note que o tamanho do sinal de entrada $x[]$ é $M = 5$, o tamanho do filtro é $N = 3$ e o tamanho da saída $y[]$ é 7.

Dentre os filtros mais conhecidos para suavização de sinais estão o filtro da média e o filtro gaussiano. O filtro da média é definido por:

$$h[] = \frac{1}{n} [1, 1, \dots, 1]$$

Para ordens 3, 5 e 7 temos

$$h_3[] = \left[\frac{1}{3}, \frac{1}{3}, \frac{1}{3} \right], \quad h_5[] = \left[\frac{1}{5}, \frac{1}{5}, \frac{1}{5}, \frac{1}{5}, \frac{1}{5} \right] \quad \text{e} \quad h_7[] = \left[\frac{1}{7}, \frac{1}{7}, \frac{1}{7}, \frac{1}{7}, \frac{1}{7}, \frac{1}{7}, \frac{1}{7} \right]$$

Outro filtro muito conhecido é o Gaussiano, cujos pesos definem aproximadamente uma distribuição normal. Uma forma de gerar um filtro Gaussiano de ordem n é dada pelo seguinte algoritmo:

1. Gerar a $(n-1)$ -ésima linha do triângulo de Pascal
2. Normalizar os coeficientes pela soma dos pesos

Ex:

Linha 0:	1								
Linha 1:	1	1							
Linha 2:	1	2	1						
Linha 3:	1	3	3	1					
Linha 4:	1	4	6	4	1				
Linha 5:	1	5	10	10	5	1			
Linha 6:	1	6	15	20	15	6	1		
Linha 7:	1	7	21	35	35	21	7	1	
Linha 8:	1	8	28	56	70	56	28	8	1

onde o k-ésimo elemento da linha n é dado por $\binom{n}{k} = \frac{n!}{k!(n-k)!}$

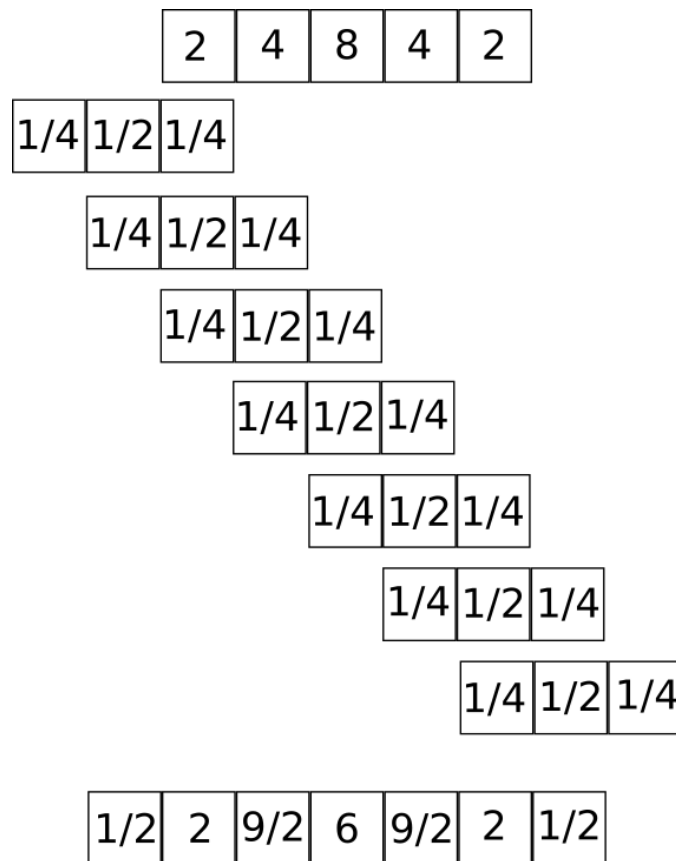
Assim, o filtro Gaussiano de ordem 9 é dado por:

$$h[] = \left[\frac{1}{256}, \frac{8}{256}, \frac{28}{256}, \frac{56}{256}, \frac{70}{256}, \frac{56}{256}, \frac{28}{256}, \frac{8}{256}, \frac{1}{256} \right]$$

$$h[] = [0.0039, 0.03125, 0.109375, 0.21875, 0.2734375, 0.21875, 0.109375, 0.03125, 0.0039]$$

Ilustração gráfica

A equação de convolução implementa o processo de deslizar uma máscara sobre o sinal/imagem, substituindo o elemento central pela soma dos produtos de cada peso da máscara por seu respectivo elemento no sinal. Trata-se de um processo de média móvel ponderada.



OBS: Se o suporte de $h[]$ é M e o suporte do sinal $x[]$ é N , a saída será um sinal de suporte:

$$L = N + M - 1$$

É comum em aplicações práticas, desconsiderar as bordas do resultado devido ao espalhamento da convolução para que a saída e a entrada possuam a mesma dimensão.

O processo de convolução computacionalmente consiste na implementação de uma janela deslizante sobre os elementos do sinal, fazendo a multiplicação de cada elemento do sinal com seu respectivo elemento da janela, somando e atribuindo o resultado ao pixel central de uma cópia do sinal. O processo nada mais é que uma média móvel ponderada pelos pesos da máscara/filtro.

A seguir é apresentado um script em Python que implementa a filtragem por convolução. São utilizados sinais simulados e séries temporais reais para ilustrar os resultados obtidos.

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

# Função que realiza a convolução do sinal com o filtro h
def convolucao(sinal, h):
    filtrado = np.zeros(len(sinal))
    # filtra sinal
    for i in range(len(sinal)):
        for j in range(len(h)):
            filtrado[i] = filtrado[i] + h[j]*sinal[i-j]
    return filtrado

# Função que calcula o binomial de N sobre K
def combinacao(n, k):
    return (math.factorial(n)/(math.factorial(k)*math.factorial(n-k)))

def filtro_gaussiano(n):
    h = np.zeros(n+1)
    for i in range(n+1):
        h[i] = (combinacao(n,i))
    return h/sum(h)

def filtro_media(n):
    h = (1/n)*np.ones(n) # filtro da média
    return h

h = filtro_media(13)          # gera o filtro h

# cria sinal senoidal corrompido por ruído gaussiano
t = np.linspace(0, 2*np.pi, 1000)
sinal = np.sin(t) + np.random.normal(0, 0.05, 1000)

plt.figure(1)
plt.plot(t, sinal, 'r')
plt.axis([0, 2*np.pi, -1.5, 1.5])
plt.show()

# realiza filtragem
filtrado = convolucao(sinal, h)

# plota resultados
plt.figure(2)
plt.plot(t, filtrado)
plt.axis([0, 2*np.pi, -1.5, 1.5])
plt.show()

# Lê dados do arquivo (série temporal)
a = np.loadtxt('soi.txt')

plt.figure(3)
plt.plot(a, 'r')
plt.show()
```

```

filtrado2 = convolucao(a, h)

plt.figure(4)
plt.plot(filtrado2)
plt.axis([0, 600, -8, 6])
plt.show()

```

Convolução 2D e filtragem de imagens

No caso 2D a generalização da integral de convolução é direta. A diferença primordial é que agora estamos no domínio do espaço e não mais no domínio do tempo. Dadas um sinal de entrada $f(x, y)$ e um sistema cuja resposta impulsiva seja $h(x, y)$, então a saída $g(x, y)$ do Sistema Linear e Invariante no Espaço (SLIE) é dada pela integral de convolução 2D:

$$g(x, y) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(\alpha, \beta) h(x - \alpha, y - \beta) d\alpha d\beta$$

Note porém, que a resolução analítica da integral no caso 2D não é tão simples quanto no caso 1D, necessitando de uma interpretação geométrica um pouco mais elaborada. Tipicamente, funções de 2 variáveis no domínio do espaço são interpretadas como imagens analógicas em tons de cinza. No caso digital, a imagem é dividida em pixels, que são amostras discretas da imagem analógica.

No caso de filtragem de imagens, o processo de convolução é generalizado para o caso bidimensional. Nesse caso, temos como entrada uma matriz, que representa uma imagem em tons de cinza por exemplo, e um filtro ou máscara 3 x 3, 5 x 5, 7 x 7, etc. A saída que também será uma matriz, representa a imagem filtrada. A expressão para a convolução discreta 2D é dada por:

$$y[n][m] = \sum_{k=0}^w \sum_{l=0}^w h[k][l] x[n-k][m-l]$$

onde

$h[k][l]$: filtro (mascara/filtro)

$x[n][m]$: imagem de entrada

$y[n][m]$: imagem filtrada

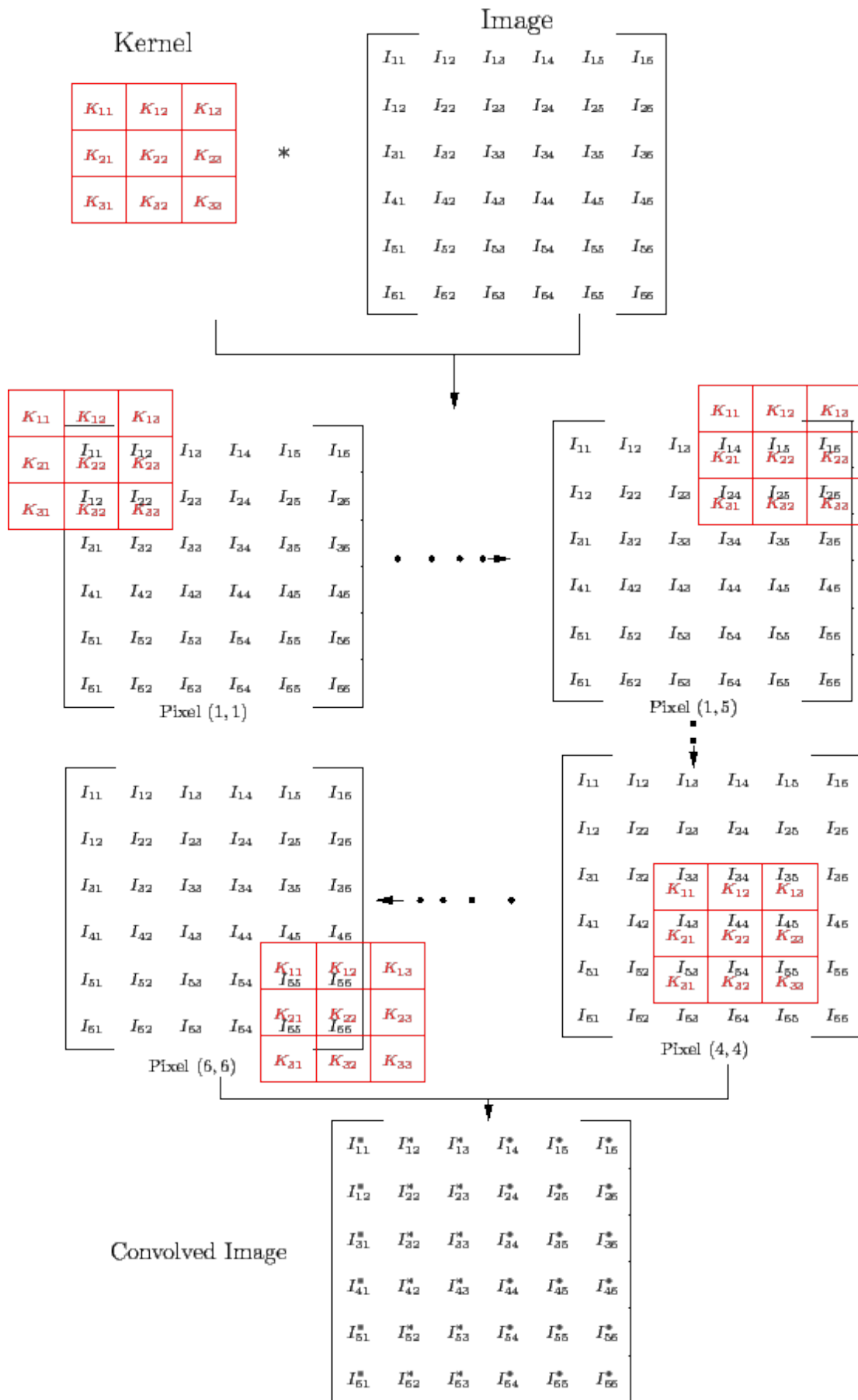
w: dimensão da máscara (3x3, 5x5, 7x7, ...)

A interpretação da fórmula consiste em deslizar a máscara sobre todos os pontos da imagem (sinal bidimensional) e efetuar a operação de convolução sobre cada elemento.

OBS: Em geral para lidar com o problema de valor de contorno (filtragem das bordas), recomenda-se realizar uma de duas opções:

- i) implementar um processo circular (borda superior ligada com a borda inferior e borda esquerda ligada com borda direita). Imagem é vista topologicamente como um toro (não existem bordas).
- ii) espelhar as bordas da imagem (rebate os pixels das bordas)

Para fins didáticos, o diagrama mostrado na figura a seguir ilustra o processo de convolução de uma máscara 3 x 3 com uma imagem 6 x 6. Note que é preciso deslizar a mascara por todos os pontos da matriz, o que requer uma varredura completa.



A seguir é apresentado um script em Python que lê uma imagem em tons de cinza, adiciona ruído e aplica o filtro da média. Note como o resultado é uma versão borrada da imagem de entrada.

```
# realiza a filtragem de imagens por convolução
import numpy as np
import matplotlib.pyplot as plt
import skimage.util as skiu

# Função que realiza a convolução 2D da imagem com filtro h
def convolucao2D(img, h):
    num_linhas = img.shape[0]
    num_colunas = img.shape[1]
    filtrada = np.zeros((num_linhas, num_colunas))
    # filtra imagem
    for i in range(num_linhas):
        for j in range(num_colunas):
            for k in range(h.shape[0]):
                for l in range(h.shape[1]):
                    filtrada[i,j] += h[k,l]*img[i-k, j-l]
    return filtrada

# Lê, adiciona ruído e plota imagem de entrada
img = plt.imread('lena.jpg')
ruidosa = skiu.random_noise(img, 'gaussian', mean=0, var=0.008)
plt.figure(1)
plt.imshow(ruidosa, cmap='gray')
plt.savefig('ruidosa.jpg')
plt.show()

# Cria máscara (filtro da média)
n = 5
h = (1/n**2)*np.ones((n, n))

# Filtra imagem de entrada
saida = convolucao2D(ruidosa, h)

# Plota imagem de saída
plt.figure(2)
plt.imshow(saida, cmap='gray')
plt.savefig('saida.jpg')
plt.show()
```

O Filtro da Mediana

O filtro da mediana é uma técnica não linear para a remoção de ruídos em imagens. É um filtro muito utilizado em processamento de imagens, pois sob certas condições ele remove o ruído preservando bordas e detalhes finos (não borra tanto a imagem). É especialmente eficaz no caso de ruído impulsivo do tipo sal e pimenta. Uma imagem contendo esse tipo de ruído apresenta pixels escuros em regiões claras e pixels claros em regiões escuras, simulando “dead pixels” em um monitor LCD (pontos pretos ou brancos defeituosos). A figura a seguir ilustra uma imagem corrompida por ruído deste tipo.



O funcionamento do filtro da mediana consiste basicamente em mover uma janela de tamanho 3 x 3, 5 x 5, 7 x 7, etc. pela imagem, substituindo o pixel central pela mediana dos pixels da janela. Para isso, é necessário ordenar os elementos da janela em um vetor e escolher o valor que encontra-se na posição central (meio do vetor). A figura a seguir ilustra a ideia. Durante o processo de filtragem o pixel 150 será substituído pelo valor 124. Esse é filtro considerado robusto pois consegue eliminar outliers, isto é, pontos muito extremos e longes da média.

	123	125	126	130	140
	122	124	126	127	135
	118	120	150	125	134
	119	115	119	123	133
	111	116	110	120	130

Neighbourhood values:

115, 119, 120, 123, 124,
125, 126, 127, 150

Median value: 124

A seguir, será apresentado um script em Python que implementa a filtragem da mediana em uma imagem corrompida por ruído impulsivo do tipo sal e pimenta. Como método de ordenação escolhemos o algoritmo Selection sort, mas outros podem ser utilizados.

```

# Aplica o filtro da mediana para remover ruído sal e pimenta
import numpy as np
import matplotlib.pyplot as plt
import skimage.util as skiu

# Função que ordena um vetor
def SelectionSort(vetor):
    for i in range(len(vetor)):
        menor = i
        for k in range(i+1, len(vetor)):
            if vetor[k] < vetor[menor]:
                menor = k

        tmp = vetor[menor]
        vetor[menor] = vetor[i]
        vetor[i] = tmp

# percorre a imagem img e para cada bloco de tamanho 2n+1, o transforma
num vetor, ordena e obtém a mediana
def filtro_mediana(img):
    num_linhas = img.shape[0]
    num_colunas = img.shape[1]
    filtrada = np.zeros((num_linhas, num_colunas))

    # filtra imagem
    for i in range(num_linhas):
        for j in range(num_colunas):
            bloco = img[i-2:i+3, j-2:j+3]
            vetor = np.resize(bloco, 25)
            SelectionSort(vetor) # ordena elementos da janela
            mediana = vetor[12] # elemento central do vetor
            filtrada[i,j] = mediana

    return filtrada

# Início do script
img = plt.imread('lena.jpg')
ruidosa = skiu.random_noise(img, 's&p') # adiciona ruído

plt.figure(1)
plt.imshow(ruidosa, cmap='gray')
plt.savefig('ruidosa.jpg')
plt.show()

# Filtra imagem de entrada
saida = filtro_mediana(ruidosa)

# Plota imagem de saída
plt.figure(2)
plt.imshow(saida, cmap='gray')
plt.savefig('saida.jpg')
plt.show()

```

Autômatos celulares

Modelos de autômatos celulares definem ferramentas computacionais muito importantes para a simulação e estudo de sistemas complexos. Um sistema é dito complexo quando suas propriedades não são uma consequência natural de seus elementos constituintes vistos isoladamente. As propriedades emergentes de um sistema complexo decorrem em grande parte da relação não-linear entre as partes. Costuma-se dizer de um sistema complexo que o todo é mais que a soma das partes. Exemplos de sistemas complexos incluem redes sociais, colônias de animais, o clima e a economia.

Uma pergunta recorrente no estudo de tais sistemas é: porque e como padrões complexos emergem a partir da interação entre os elementos? Como esses padrões evoluem com o tempo? Respostas a essas perguntas não são totalmente conhecidas, mas o estudo de modelos de autômatos celulares nos auxiliam no estudo e análise de tais sistemas. Aplicações práticas são muitas e incluem:

- Autômatos celulares e composição musical
- Autômatos celulares e modelagem urbana
- Autômatos celulares e propagação de epidemias
- Autômatos celulares e crescimento de câncer

Os primeiros modelos de autômatos celulares foram propostos originalmente na década de 40 por John Von Neumann e tinham como objetivos principais:

- Representar matematicamente a evolução natural em sistemas complexos
- Desenvolver máquinas de auto-replicação, através de um conjunto de regras matemáticas objetivas
- Estudar a auto-organização em sistemas complexos

Segundo Wolfram, autômatos celulares são formados por uma rede de células que possuem seus estados alterados num tempo discreto de acordo com seu estado anterior e o estado de suas células vizinhas. Algumas características importantes e comuns a todos os autômatos celulares são:

- Homogeneidade: as regras são iguais para todas as células
- Estados discretos: cada célula pode estar em um dos finitos estados
- Interações locais: o estado de uma célula depende apenas das células mais próximas (vizinhas)
- Processo dinâmico: a cada instante de tempo as células podem sofrer uma atualização de estado

Def: Um autômato celular é definido por uma 5-tupla de elementos

$$A = (R, S, S_0, V, F) \quad \text{onde}$$

R é a grade de células (pode ser 1D, 2D, 3D,...)

S é o conjunto de estados de uma célula c_i

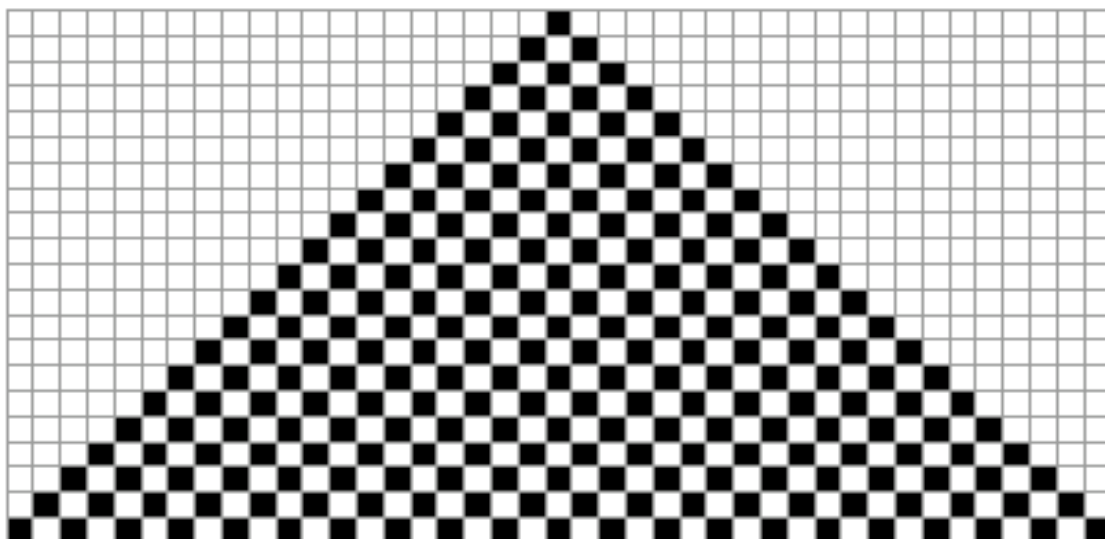
S_0 é o estado inicial do sistema

V é conjunto vizinhança (define quem são os vizinhos de cada célula)

F é a função de transição (regras de governam a evolução do sistema no tempo)

Autômatos Celulares Elementares

Um autômato celular é dito elementar se o reticulado de células R é unidimensional (ou seja, pode ser representado por um vetor), o conjunto de estados $S = \{0, 1\}$ e o conjunto vizinhança engloba apenas duas células: a anterior ($i-1$) e a posterior ($i+1$). Tipicamente, se uma célula assume estado 0 dizemos que ela está morta e se ela assume estado 1 dizemos que está viva. A figura a seguir ilustra as primeiras 20 gerações de um autômato celular elementar, em que no início apenas uma célula está viva (preto = vivo, branco = morto)

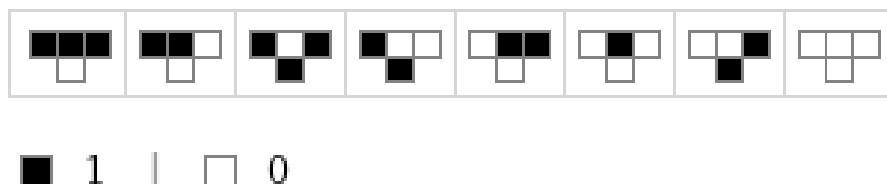


A questão é: como evoluir uma configuração de forma a construir esse padrão? Que regras são aplicadas para definir quais células vivem ou morrem na próxima geração?

A função de transição do autômato da figura é dada pela seguinte tabela.

$x_t(i-1)$	$x_t(i)$	$x_t(i+1)$	$x_{t+1}(i)$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

Uma forma de resumir toda essa tabela é através da seguinte representação:



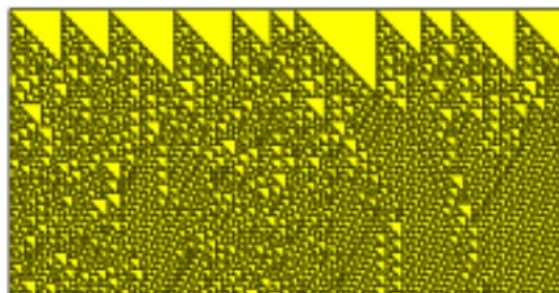
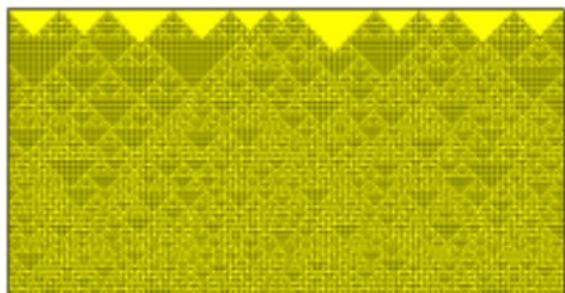
O que essa regra nos diz pode ser sumariado em 8 fatos:

- 1) sempre que a célula i for morta e a $(i-1)$ e $(i+1)$ também forem, a célula i permanecerá morta na próxima geração.
- 2) sempre que a célula i for morta, a $(i-1)$ for morta e a $(i+1)$ for viva, a célula i viverá na próxima geração.
- 3) sempre que a célula i for viva e a $(i-1)$ e a $(i+1)$ forem mortas, a célula i morrerá na próxima geração.
- 4) sempre que a célula i for viva, a $(i-1)$ for morta e a $(i+1)$ for viva, a célula i morrerá na próxima geração.

- 5) sempre que a célula i for morta, a $(i-1)$ for viva e a $(i+1)$ for morta, a célula i viverá na próxima geração.
- 6) sempre que a célula i for morta e ambas $(i-1)$ e $(i+1)$ forem vivas, a célula i viverá na próxima geração.
- 7) sempre que a célula i for viva, a $(i-1)$ for viva e a $(i+1)$ for morta, a célula i morrerá na próxima geração.
- 8) sempre que a célula i for viva e ambas $(i-1)$ e $(i+1)$ forem vivas, a célula i morrerá na próxima geração.

Note que não existem mais combinações possíveis de 0's e 1's usando apenas 3 bits, pois conseguimos contar em binário de 0 a 7, o que resulta em 8 possibilidades. Essa regra tem um nome: é a regra 50, pois o número binário correspondente a última coluna da função de transição vale 00110010, que em binário é justamente o número 50. Sendo assim, quantas possíveis regras existem para um autômato celular elementar? Basta computar 2^8 , que resulta em 256. Portanto, o número total de regras distintas é 256. Por essa razão dizemos que existem 256 autômatos celulares elementares distintos, um para cada regra. O interessante é estudar e simular o que acontece com cada um desses autômatos durante sua evolução. De acordo com Wolfram, existem 4 classes de regras para um autômato celular elementar:

- Classe 1: Estado Homogêneo
Todas as células chegarão num mesmo estado após um número finito de estados
- Classe 2: Estável simples
As células não possuirão todas o mesmo estado, mas eles se repetem com a evolução temporal
- Classe 3: Padrão irregular
Não possui padrão reconhecível
- Classe 4: Estrutura complexa
Estruturas complexas que evoluem imprevisivelmente



As regras mais interessantes são as da classe 4, pois definem um sistema complexo com propriedades dinâmicas interessantes, sendo algumas delas capazes até de simular máquinas de Turing, que são modelos computacionais capazes de serem programadas para realizar diferentes tarefas computacionais. Um exemplo de regra com essa característica é a regra 110. (Referências: https://en.wikipedia.org/wiki/Rule_110, <http://www.complex-systems.com/pdf/15-1-1.pdf>)

Exercício: Construa a função de transição do autômato celular elementar definido pela regra 30. Aplique a regra para evoluir a condição inicial idêntica a da figura da regra 50 (apenas uma célula viva) por 20 gerações. Repita o exercício mas agora para a regra 110.

A seguir é apresentado um algoritmo para a simulação de autômatos celulares elementares.

```
geração = vetor(N) (N é o número de células)

nova_geração = vetor(N)

evolução = matriz(MAX, N) (MAX é o número de gerações)

Inicializar geração (setar configuração inicial)

para i = 1 até MAX
    evolução[i,:] = geração
    # Percorre cada célula da geração atual
    para j = 1 até N
        Aplicar regra de transição na célula j, gerando nova_geração
    geração = nova_geração

Plotar resultados
```

Ex: Baseado no algoritmo anterior, implementar um script em Python que, dado uma regra (número de 0 a 255), evolua uma configuração inicial de tamanho N = 1000 até a geração 500.

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

# converte um número inteiro para sua representação binária (0-255)
def converte_binario(numero):
    binario = bin(numero)
    binario = binario[2:]
    if len(binario) < 8:
        zeros = [0]*(8-len(binario))
        binario = zeros + list(binario)
    return list(binario)

# Início do script
MAX = 500
g = np.zeros(1000)
ng = np.zeros(1000)

regra = int(input('Entre com o número da regra: '))
codigo = converte_binario(regra)
```

```

# Matriz em que cada linha armazena uma geração do autômato
matriz_evolucao = np.zeros((MAX, len(g)))

# Define geração inicial
g[len(g)//2] = 1

# Laço principal: atualiza as gerações
for i in range(MAX):

    matriz_evolucao[i,:] = g

    # Percorre células da geração atual
    for j in range(len(g)):

        if (g[j-1] == 0 and g[j] == 0 and g[(j+1)%len(g)] == 0):
            ng[j] = int(codigo[7])
        elif (g[j-1] == 0 and g[j] == 0 and g[(j+1)%len(g)] == 1):
            ng[j] = int(codigo[6])
        elif (g[j-1] == 0 and g[j] == 1 and g[(j+1)%len(g)] == 0):
            ng[j] = int(codigo[5])
        elif (g[j-1] == 0 and g[j] == 1 and g[(j+1)%len(g)] == 1):
            ng[j] = int(codigo[4])
        elif (g[j-1] == 1 and g[j] == 0 and g[(j+1)%len(g)] == 0):
            ng[j] = int(codigo[3])
        elif (g[j-1] == 1 and g[j] == 0 and g[(j+1)%len(g)] == 1):
            ng[j] = int(codigo[2])
        elif (g[j-1] == 1 and g[j] == 1 and g[(j+1)%len(g)] == 0):
            ng[j] = int(codigo[1])
        elif (g[j-1] == 1 and g[j] == 1 and g[(j+1)%len(g)] == 1):
            ng[j] = int(codigo[0])

    g = ng.copy() # se não usar copy ambos vetores tornam-se o mesmo

# plota matriz resultante como imagem
plt.figure(1)
plt.axis('off')
plt.imshow(matriz_evolucao, cmap='gray')
plt.savefig('Automata.png', dpi=300)
plt.show()

```

Autômatos celulares 2D

O Jogo da Vida

O autômato 2D mais conhecido, sem dúvida, é o jogo da vida, criado por Conway para simular a evolução de sistemas complexos a partir de regras determinísticas. O reticulado 2D é representado computacionalmente por uma matriz geralmente quadrada de células que podem estar vivas ou mortas. A função de vizinhança é definida pela vizinhança de Moore, ou seja, pelas 8 células mais próximas a uma dada célula i , conforme ilustra a figura a seguir.

A função de transição do jogo da vida tem como conceito imitar processos de nascimento e morte. A ideia básica é que um ser vivo necessita de outros seres vivos para sobreviver e procriar, mas um excesso de densidade populacional provoca a morte do ser vivo devido à escassez de recursos.

Two-dimensional cellular automata

1	0	1	0	1	0
0	0	1	0	1	1
1	1	1	0	1	1
1	0	1	0	1	0
0	0	0	1	1	0
1	1	0	0	1	0
1	1	1	0	0	0
1	0	1	1	1	1

a neighborhood
of 9 cells

São 4 regras básicas:

R1 (Sobrevivência) – uma célula viva com 2 ou 3 células vizinhas vivas, permanece viva na próxima geração.

R2 (Morte por isolamento) – uma célula viva com 0 ou 1 vizinho vivo morre de solidão na próxima geração.

R3 (Morte por sufocamento) – uma célula viva com 4 ou mais vizinhos vivos morre por sufocamento na próxima geração.

R4 (Renascimento) – uma célula morta com exatamente 3 vizinhos vivos, renasce na próxima geração.

Isso nos leva a regra de transição conhecida como B3S23, uma vez que no código proposto B significa born (nascer) e S significa survive (sobreviver). Em outras palavras, nesse autômato em particular, uma célula nasce sempre que possui 3 vizinhas vivas ao redor e sobrevive se possui 2 ou 3 vizinhas vivas ao redor. Outras variantes de regras incluem: B15S257, B147S256, B34S4567, etc. Cada uma das regras define um autômato diferente. Ao autômato cuja regra é B3S23 dá-se o nome de Jogo da Vida.

É interessante perceber que a regra B3S23 a partir de diversas inicializações simples exibe um comportamento altamente complexo, onde padrões complexos e “seres vivos” passam a interagir de maneira bastante inesperada. Trata-se de um conjunto de regras totalmente determinísticas que levam a um comportamento completamente imprevisível (ordem x caos).

Para uma coletânea de condições iniciais verifique o link:

<https://jakevdp.github.io/blog/2013/08/07/conways-game-of-life/>

Um problema comum que afeta simulações computacionais do jogo da vida é o chamado problema de valor de contorno. Isso nada mais é que uma falha ao se definir a função de transição para células na borda do sistema. Para se evitar essa indefinição, considera-se que o reticulado é na verdade um toro. Isso significa que não existem bordas, uma vez que a borda da esquerda é ligada a borda da direita, assim como a inferior é ligada a superior.

Ex: Implementar um script em Python que, dada uma configuração inicial, simule o jogo da Vida num tabuleiro de dimensões 100 x 100 por 200 gerações.

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

# cria inicialização: rpentomino nas coordenadas i, j
def init_config(tabuleiro, padrao, i, j):
    linhas = padrao.shape[0]
    colunas = padrao.shape[1]
    tabuleiro[i:i+linhas, j:j+colunas] = padrao

# Início do script
inicio = time.time()
MAX = 200
SIZE = 100

geracao = np.zeros((SIZE, SIZE))
nova_geracao = np.zeros((SIZE, SIZE))

# Cubo de dados em que cada fatia representa uma geração
matriz_evolucao = np.zeros((SIZE, SIZE, MAX))

# Define geração inicial

unbounded = np.array([[1, 1, 1, 0, 1],
                      [1, 0, 0, 0, 0],
                      [0, 0, 0, 1, 1],
                      [0, 1, 1, 0, 1],
                      [1, 0, 1, 0, 1]])

glider = np.array([[1, 0, 0],
                  [0, 1, 1],
                  [1, 1, 0]])

r_pentomino = np.array([[0, 1, 1],
                       [1, 1, 0],
                       [0, 1, 0]])

diehard = np.array([[0, 0, 0, 0, 0, 0, 1, 0],
                    [1, 1, 0, 0, 0, 0, 0, 0],
                    [0, 1, 0, 0, 0, 1, 1, 1]])

acorn = np.array([[0, 1, 0, 0, 0, 0, 0],
                 [0, 0, 0, 1, 0, 0, 0],
                 [1, 1, 0, 0, 1, 1, 1]])

init_config(geracao, r_pentomino, SIZE//2, SIZE//2)
```

```

# Laço principal (atualiza gerações)
for k in range(MAX):

    print('Processando geração %d...' %k)
    matriz_evolucao[:, :, k] = geracao

    # Laço principal: atualiza as gerações
    for i in range(SIZE):

        for j in range(SIZE):

            vivos = geracao[i-1, j-1] + geracao[i-1, j] + \
                    geracao[i-1, (j+1)%SIZE] + geracao[i, j-1] + \
                    geracao[i, (j+1)%SIZE] + geracao[(i+1)%SIZE, j-1] + \
                    geracao[(i+1)%SIZE, j] + geracao[(i+1)%SIZE, (j+1)%SIZE]

            if (geracao[i,j] == 1):
                if (vivos == 2 or vivos == 3):
                    nova_geracao[i,j] = 1
                else:
                    nova_geracao[i,j] = 0
            else:
                if (vivos == 3):
                    nova_geracao[i,j] = 1

        geracao = nova_geracao.copy()

fim = time.time()
print('Tempo gasto na simulação: %.2f s' %(fim-inicio))

# Gera animação da evolução do sistema
fig = plt.figure(1)
plt.axis('off')
lista = []
for i in range(MAX):
    im = plt.imshow(matriz_evolucao[:, :, i], cmap='gray')
    lista.append([im])

ani = animation.ArtistAnimation(fig, lista, interval=100, blit=True,
                                repeat_delay=1000)

ani.save('r_pentomino.gif', writer='imagemagick')

plt.show()

```

Filtragem de mínimos quadrados

A formulação matemática do filtro de Wiener é realizada em termos estatísticos.

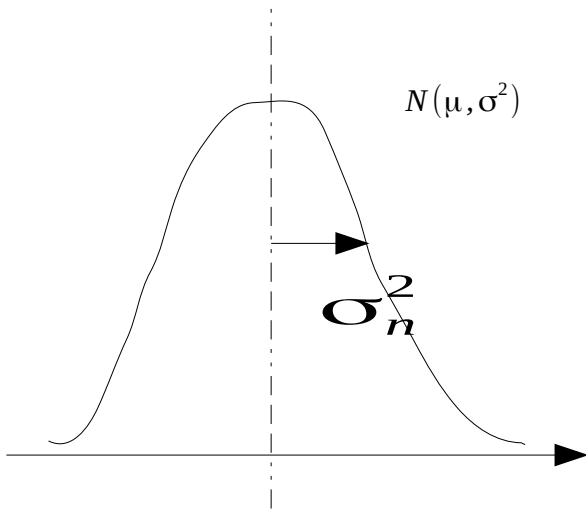
Objetivo: Encontrar o estimador linear ótimo para um sinal corrompido por ruído gaussiano aditivo e independente do sinal.

$$f(n) = s(n) + \eta(n)$$

$f(n)$ = observação (pixel ruidoso da imagem observada)

$s(n)$ = pixel livre de ruído (o que quero descobrir)

$\eta(n)$ = ruído \rightarrow Variável Aleatória Normal $(0, \sigma_n^2)$ - perturbação aleatória



$$p(n; \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma_n^2}} \exp\left\{\frac{-1}{2\sigma_n^2} n^2\right\}$$

Estimador linear: $\hat{s}(n) = \alpha f(n) + \beta \rightarrow$ Função linear da observação (equação da reta $y = ax + b$)

Hipótese: Sinal e ruído não correlacionados; Ruído é gaussiano com média zero e variância σ_n^2 .

Então:

$$E[f(n)] = E[s(n)] \text{ ou seja } \mu_f = \mu_s \text{ e}$$

$$Var[f(n)] = Var[s(n)] + Var[\eta(n)] \text{ ou seja } \sigma_f^2 = \sigma_s^2 + \sigma_n^2$$

Objetivo: Escolher α e β da melhor maneira possível (minimizando o erro quadrático médio entre o sinal original $s(n)$ e sua estimativa $\hat{s}(n)$).

$$J(\alpha, \beta) = E[(\hat{s}(n) - s(n))^2] = E[(\alpha f(n) + \beta - s(n))^2]$$

Derivando em relação a α e β :

$$1) \quad \frac{\partial}{\partial \alpha} J(\alpha, \beta) = E[(\alpha f(n) + \beta - s(n)) f(n)] = 0$$

$$\text{II)} \quad \frac{\partial}{\partial \beta} J(\alpha, \beta) = E[(\alpha f(n) + \beta - s(n))] = 0$$

Resolvendo I temos:

$$\begin{aligned} E[\alpha f(n)^2 + \beta f(n) - s(n)f(n)] &= 0 \\ \alpha E[f(n)^2] + \beta E[f(n)] - E[s(n)f(n)] &= 0 \\ \alpha E[f(n)^2] + \beta E[f(n)] - E[s(n)(s(n) + \eta(n))] &= 0 \end{aligned}$$

Note que $\text{Var}[x] = E[x^2] - E^2[x]$ o que implica em $E[x^2] = \text{Var}[x] + E^2[x]$. Assim, temos:

$$\begin{aligned} \alpha(\sigma_f^2 + \mu_f^2) + \beta\mu_f - (\sigma_s^2 + \mu_s^2) &= 0 \\ \alpha(\sigma_f^2 + \mu_f^2) + \beta\mu_f - \sigma_s^2 - \mu_s^2 &= 0 \end{aligned}$$

Resolvendo II temos:

$$\begin{aligned} \alpha E[f(n)] + \beta - E[s(n)] &= 0 && (\text{note que } \mu_f = \mu_s) \\ \alpha\mu_f + \beta - \mu_f &= 0 \end{aligned}$$

$$\beta = \mu_f - \alpha\mu_f = (1 - \alpha)\mu_f$$

Substituindo β na equação I:

$$\begin{aligned} \alpha(\sigma_f^2 + \mu_f^2) + (1 - \alpha)\mu_f\mu_f - \sigma_s^2 - \mu_s^2 &= 0 && (\text{note que } \mu_s^2 = \mu_f^2) \\ \alpha\sigma_f^2 + \alpha\mu_f^2 + \mu_f^2 - \alpha\mu_f^2 - \sigma_s^2 - \mu_f^2 &= 0 \\ \alpha = \frac{\sigma_s^2}{\sigma_f^2} = \frac{\sigma_s^2}{(\sigma_s^2 + \sigma_n^2)} \end{aligned}$$

Portanto o estimador $\hat{s}(n)$ fica:

$$\hat{s}(n) = \alpha f(n) + \beta = \frac{\sigma_s^2}{(\sigma_s^2 + \sigma_n^2)} f(n) + \left(1 - \frac{\sigma_s^2}{(\sigma_s^2 + \sigma_n^2)}\right) \mu_f$$

$$\hat{s}(n) = \mu_f + \frac{\sigma_s^2}{(\sigma_s^2 + \sigma_n^2)} (f(n) - \mu_f), \text{ onde } \frac{\sigma_s^2}{(\sigma_s^2 + \sigma_n^2)} \text{ é o ganho do filtro de Wiener}$$

$$\sigma_s^2 = \sigma_f^2 - \sigma_n^2 \quad (\text{porque ruído é aditivo})$$

sendo que é possível estimar μ_f e σ_f^2 diretamente do sinal observado

$$\begin{aligned} \mu_f &= \frac{1}{N} \sum_n f(n) \\ \sigma_f^2 &= \frac{1}{N} \sum_n (f(n) - \mu_f)^2 \end{aligned} \quad (\text{média e variância amostrais})$$

Essa estimação é feita localmente usando uma janela adaptativa ao redor do pixel atual.

$f(n)$ é o ponto central da janela no pixel n da img ruidosa (tipicamente 3 x 3, 5 x 5, 7 x 7, ...)

μ_f é a média dos pixels da janela

σ_f^2 é a variância dos pixels da janela

σ_n^2 consiste numa constante referente ao nível de ruído (quanto mais ruído maior a variância).

Filtro de Wiener Adaptativo Pontual

Sabendo que

$$f(n) = s(n) + \eta(n)$$

o melhor estimador pontual possível em termos de mínimos quadrados é dado por:

$$\hat{s}(n) = \mu_f + \frac{\sigma_s^2}{(\sigma_s^2 + \sigma_n^2)} (f(n) - \mu_f)$$

onde μ_f : média local numa janela (3x3, 5x5, 7x7, etc.) na imagem observada e σ_s^2 é a variância local na imagem livre de ruído.

A pergunta que fica é: como obter σ_s^2 ? Há basicamente duas maneiras:

- 1) $\sigma_s^2 = \min\{(\sigma_f^2 - \sigma_n^2), 0\}$ (em geral essa é preferida)
- 2) Pré-suavização da img. ruidosa observada f via um filtro passa-baixa (filtro de média, gaussiano)

Algoritmo

Parâmetros: Tamanho da janela (5 x 5, 7 x 7, ...) e variância do ruído (conhecida em simulações)

1. Ler imagem do arquivo para memória
2. Normalizar imagem: dividir o valor de cada pixel pelo máximo da imagem
3. Criar uma imagem s da mesma dimensão de f com zeros
4. Para todo pixel $p(i,j)$ da imagem

Computar a média dos valores da janela ao redor de $p(i,j)$: μ_f

Computar a variância dos valores da janela ao redor de $p(i,j)$: σ_f^2

Computar $\sigma_s^2 = \min\{(\sigma_f^2 - \sigma_n^2), 0\}$

Fazer $\hat{s}(n) = \mu_f + \frac{\sigma_s^2}{(\sigma_s^2 + \sigma_n^2)} (f(n) - \mu_f)$

5. Salvar imagem s em arquivo

Comportamento do Filtro

Há basicamente dois casos limites para ser analisados:

→ Se $\frac{\sigma_s^2}{(\sigma_s^2 + \sigma_n^2)} \Rightarrow 0, \sigma_s^2 \ll \sigma_n^2$ ou seja $\hat{s}(n) = \mu_f$ (Tende à média, filtra bastante pois ele entende que só há ruído e quase nada de sinal)

→ Se $\frac{\sigma_s^2}{(\sigma_s^2 + \sigma_n^2)} \Rightarrow 1, \sigma_s^2 \gg \sigma_n^2$ ou seja $\hat{s}(n) = f(n)$ (Tende à imagem observada, não filtra pois ele entende que não há ruído, apenas sinal)

A seguir é apresentado um script em Python que implementa o filtro de Wiener adaptativo pontual.

```
import skimage.io
import skimage.util
import skimage.measure
import matplotlib.pyplot as plt
import numpy as np

...
PARÂMETROS:
    img - imagem a ser filtrada (entre 0 e 1)
    w - tamanho da janela (3, 5, 7, ...)
    var_ruído - variância do ruído
...
def filtro_wiener(img, janela, var_ruído):

    # Aloca matriz de saída
    restored = np.zeros((img.shape[0], img.shape[1]))

    # Problema de valor de contorno (duplica bordas)
    img = skimage.util.pad(img, ((w, w), (w, w)), 'symmetric')

    # Define janelamento adaptativo (3x3 -> K=1, 5x5 -> K=2, ...)
    K = janela//2

    # Aplica filtragem
    for i in range(w, img.shape[0]-w):
        for j in range(w, img.shape[1]-w):

            # Estima média e variância locais
            media = np.mean(img[i-K:i+K+1, j-K:j+K+1])
            variancia = np.var(img[i-K:i+K+1, j-K:j+K+1])

            # Verifica se ganho de Wiener é negativo
            var_f = variancia - var_ruído
            if var_f < 0:
                var_f = 0

            restored[i-w, j-w] = media+(var_f/variancia)*(img[i,j]-media)

    return restored

##### INÍCIO DO SCRIPT
image = skimage.io.imread('lena.png')
varn = 0.008      # esse parâmetro controla a força do ruído

image_n=skimage.util.random_noise(image, mode='gaussian', mean=0, var=varn)
image_w = filtro_wiener(image_n, 7, varn)

pn = skimage.measure.compare_psnr(skimage.util.img_as_float(image), image_n)
print('PSNR: %f' %pn)

pw = skimage.measure.compare_psnr(skimage.util.img_as_float(image), image_w)
print('PSNR: %f' %pw)
```

```
plt.figure(1)
skimage.io.imshow(image_n)
skimage.io.imsave('Noisy.png', image_n)

plt.figure(2)
skimage.io.imshow(image_w)
skimage.io.imsave('Wiener.png', image_w)
```

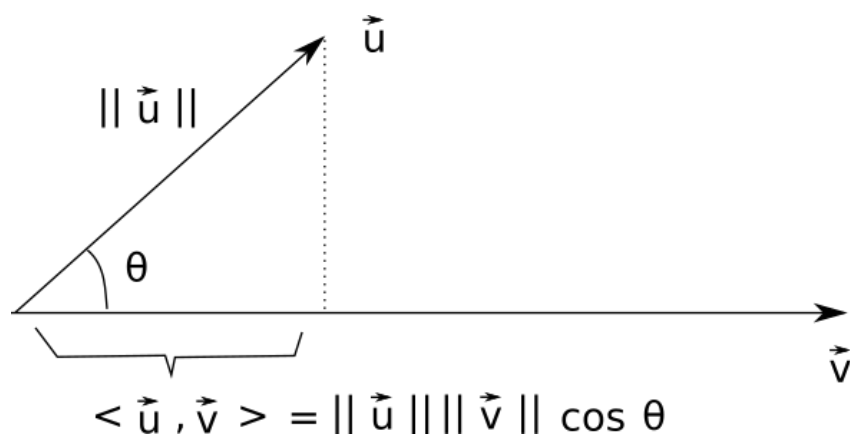
Autovalores e autovetores

Seja V um espaço vetorial de n dimensões equipado com produto interno. Então, podemos definir:

$$\langle \vec{u}, \vec{v} \rangle = \vec{u} \cdot \vec{v} = \sum_{i=1}^n u_i v_i$$

$$\langle \vec{v}, \vec{v} \rangle = \vec{v} \cdot \vec{v} = \sum_{i=1}^n v_i^2 = \|\vec{v}\|^2$$

Geometricamente:



Um operador linear é uma matriz que mapeia um vetor de entrada \vec{v} para um vetor de saída \vec{u}

$$\vec{u} = P\vec{v} \quad (\text{analogia com } y = f(x))$$

Pergunta motivadora: Dado um operador P , para quais vetores \vec{v} a saída $\vec{u} = P\vec{v}$ aponta para a mesma direção da entrada, apenas sendo esticado ou encolhido? Em termos matemáticos temos:

$$\vec{u} = P\vec{v} = \lambda \vec{v} \quad (*)$$

→ Todos os vetores \vec{v} que satisfazem a equação (*) são chamados de autovetores de P

→ Todos os escalares λ que satisfazem a equação (*) são chamados de autovalores de P .

λ é o fator de escala da redução ou incremento de tamanho do vetor

Def: Dizemos que $\vec{v} \in \mathbb{R}^n$ é um autovetor de $P_{n \times n}$ com autovalor λ se $\vec{v} \neq \vec{0}$ e:

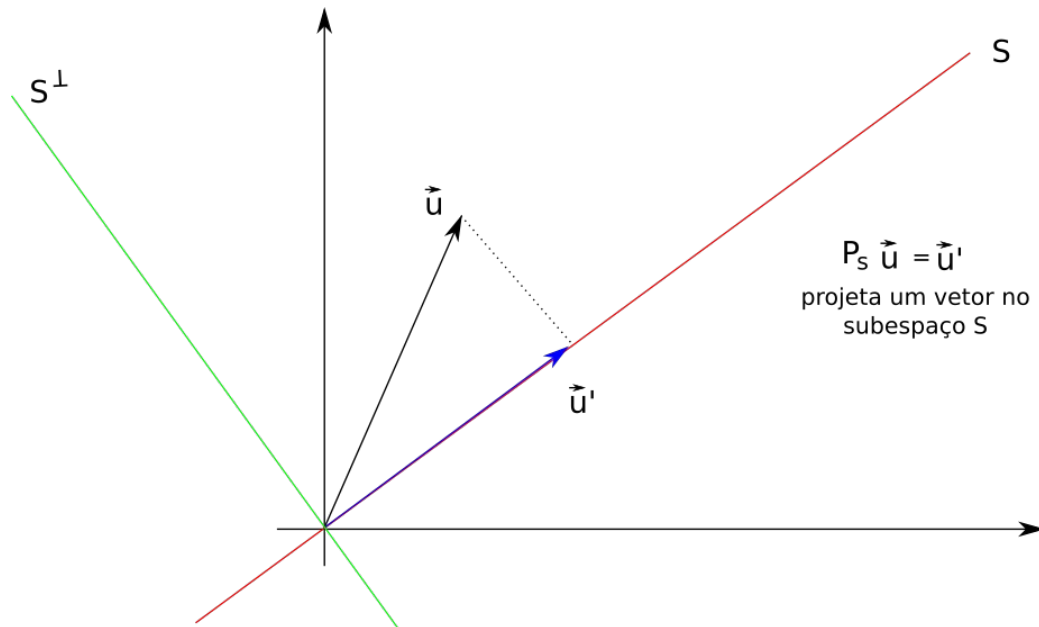
$$P\vec{v} = \lambda \vec{v}$$

Ex: Seja $P = I_{n \times n}$ o operador identidade

$$I_{n \times n} \vec{v} = 1 \vec{v}$$

ou seja, $\forall \vec{v} \in R^n$, \vec{v} é autovetor de $I_{n \times n}$ com autovalor igual a 1

Ex: operador projeção em R^2



Seja $\vec{w} \in S$, então $P_S \vec{w} = \vec{w}$, portanto todo vetor que pertence a reta S é um autovetor de P_S com autovalor $\lambda = 1$. Seja $\vec{x} \in S^\perp$, então $P_S \vec{x} = 0 \vec{x}$, portanto todo vetor que pertence a S^\perp é um autovetor de P_S com autovalor $\lambda = 0$.

Como calcular os autovalores e autovetores de uma matriz A?

Da equação (*) sabemos que:

$$\begin{aligned} A \vec{v} &= \lambda \vec{v} \\ A \vec{v} - \lambda I \vec{v} &= 0 \\ (A - \lambda I) \vec{v} &= 0 \end{aligned}$$

Teorema: O sistema linear $A \vec{v} = 0$ admite solução não nula se e somente se $\det(A) = 0$

Portanto, devemos ter que $\det(A - \lambda I) = 0$

Ex: Encontrar os autovalores e autovetores da matriz $A = \begin{bmatrix} 3/2 & -1/2 \\ -1/2 & 3/2 \end{bmatrix}$

$$A - \lambda I = \begin{bmatrix} 3/2 - \lambda & -1/2 \\ -1/2 & 3/2 - \lambda \end{bmatrix}$$

Sabemos que $(A - \lambda I) \vec{v} = 0$ admite solução se $\det(A - \lambda I) = 0$ ou seja:

$$\left(\frac{3}{2} - \lambda\right)^2 - \frac{1}{4} = 0$$

o que implica em:

$$\frac{9}{4} - 2 \cdot \frac{3}{2} \lambda + \lambda^2 - \frac{1}{4} = 0 \rightarrow \lambda^2 - 3\lambda + 2 = 0$$

cujas soluções são $\lambda_1 = \frac{3-1}{2} = 1$ e $\lambda_2 = \frac{3+1}{2} = 2$

Para obter o autovetor associado ao autovalor $\lambda_1 = 1$ temos:

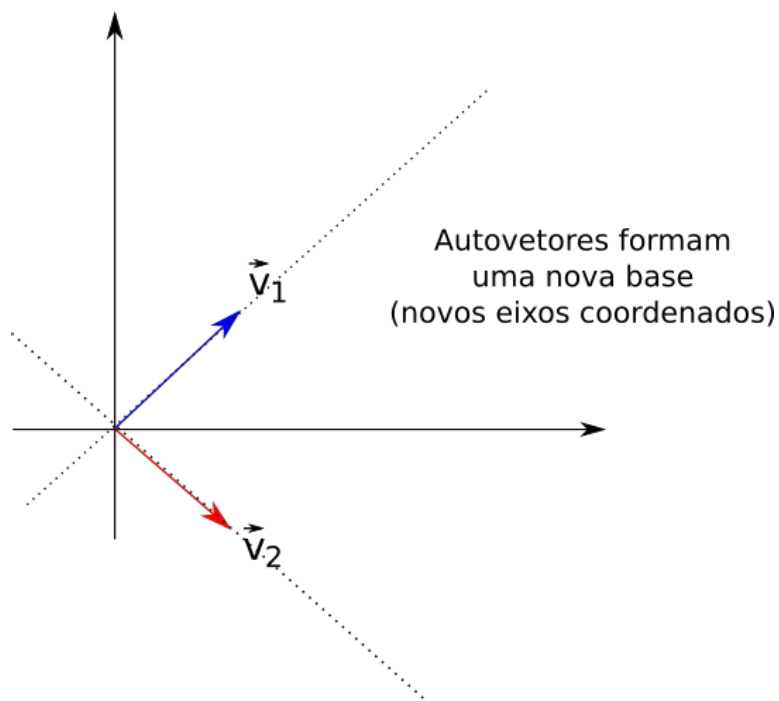
$$\begin{bmatrix} 1/2 & -1/2 \\ -1/2 & 1/2 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \rightarrow \begin{cases} \frac{1}{2}v_1 - \frac{1}{2}v_2 = 0 \\ -\frac{1}{2}v_1 + \frac{1}{2}v_2 = 0 \end{cases} \quad \text{o que implica em}$$

$$v_1 = v_2 \rightarrow \vec{v}^{(1)} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} \quad (\text{norma unitária})$$

Para obter o autovetor associado ao autovalor $\lambda_2 = 2$ temos:

$$\begin{bmatrix} -1/2 & -1/2 \\ -1/2 & -1/2 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \rightarrow \begin{cases} -\frac{1}{2}v_1 - \frac{1}{2}v_2 = 0 \\ -\frac{1}{2}v_1 - \frac{1}{2}v_2 = 0 \end{cases} \quad \text{o que implica em}$$

$$v_1 = -v_2 \rightarrow \vec{v}^{(2)} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} \end{bmatrix} \quad (\text{norma unitária})$$



Def: Se P é um operador linear que possui n autovalores distintos então os autovetores de P definem uma nova base para R^n

$$Q = \begin{bmatrix} | & | & \dots & | \\ \vec{v}_1 & \vec{v}_2 & \dots & \vec{v}_n \\ | & | & \dots & | \end{bmatrix} \quad (\text{matriz dos autovetores})$$

Obs: Uma matriz é dita positiva semidefinida se todos seus autovalores forem maiores ou iguais a 0

Def: Seja P um operador linear. Se a matriz Q dos autovetores de P define uma nova base para R^n , então $PQ = Q\Lambda$ onde Λ é a matriz diagonal dos autovalores.

$$\text{Note que } P = \begin{bmatrix} | & - & \vec{p}_1 & - & | \\ | & - & \vec{p}_2 & - & | \\ \dots & \dots & \dots & \dots & \dots \\ | & - & \vec{p}_n & - & | \end{bmatrix} \quad \text{e assim } PQ = \begin{bmatrix} \vec{p}_1 \vec{v}_1 & \vec{p}_1 \vec{v}_2 & \dots & \vec{p}_1 \vec{v}_n \\ \vec{p}_2 \vec{v}_1 & \vec{p}_2 \vec{v}_2 & \dots & \vec{p}_2 \vec{v}_n \\ \dots & \dots & \dots & \dots \\ \vec{p}_n \vec{v}_1 & \vec{p}_n \vec{v}_2 & \dots & \vec{p}_n \vec{v}_n \end{bmatrix}.$$

$$\text{Observando } PQ, \text{ podemos reescrever a matriz como } PQ = \begin{bmatrix} | & | & \dots & | \\ P\vec{v}_1 & P\vec{v}_2 & \dots & P\vec{v}_n \\ | & | & \dots & | \end{bmatrix}$$

$$\text{O lado direito é dado por } Q\Lambda = \begin{bmatrix} v_{11}\lambda_1 & v_{12}\lambda_2 & \dots & v_{1n}\lambda_n \\ v_{21}\lambda_1 & v_{22}\lambda_2 & \dots & v_{2n}\lambda_n \\ \dots & \dots & \dots & \dots \\ v_{n1}\lambda_1 & v_{n2}\lambda_2 & \dots & v_{nn}\lambda_n \end{bmatrix} = \begin{bmatrix} | & | & \dots & | \\ \lambda_1 \vec{v}_1 & \lambda_2 \vec{v}_2 & \dots & \lambda_n \vec{v}_n \\ | & | & \dots & | \end{bmatrix}$$

Portanto, o que temos definido em $PQ = Q\Lambda$ são n equações do tipo $P\vec{v}_i = \lambda_i \vec{v}_i$, $i=1, \dots, n$

Def (Eigendecomposition): Seja P uma matriz quadrada n x n, Q a matriz dos autovetores de P nas colunas e Λ a matriz diagonal dos autovalores de P. Então P pode ser decomposta como:

$$P = Q\Lambda Q^{-1}$$

Para verificar essa igualdade basta partir de $PQ = Q\Lambda$. Multiplicando ambos os lados pela inversa de Q, temos:

$$PQQ^{-1} = Q\Lambda Q^{-1}$$

$$PI = Q\Lambda Q^{-1}$$

o que nos leva ao resultado desejado.

Obs: Se P é uma matriz ortogonal, $Q^{-1} = Q^T$ e portanto $P = Q\Lambda Q^T$

Análise de Componentes Principais (PCA)

Análise de componentes principais é uma família de técnicas para tratar dados de alta dimensionalidade que utilizam as dependências entre as variáveis para representá-los de uma forma mais compacta sem perda de informação relevante. PCA é um dos métodos mais simples e robusto de realizar redução de dimensionalidade. É uma das técnicas mais antigas e foi redescoberta muitas vezes em diversas áreas da ciência, sendo conhecida também como Transformação de Karhunen-Loeve, Transformação de Hotelling ou decomposição em valores singulares.

- Método linear: assume hipótese de que dados encontram-se num subespaço Euclidiano do \mathbb{R}^n
- Método não supervisionado (não requer rótulos das classes)
- Decorrelaciona os dados de entrada eliminando redundâncias

$$\vec{x} \in \mathbb{R}^d = \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_d \end{bmatrix} \xRightarrow{W_{pca}} \vec{y} \in \mathbb{R}^k = \begin{bmatrix} y_1 \\ \dots \\ y_k \end{bmatrix}, k \ll d$$

PCA pela Maximização da Variância

Seja $Z = [T^T, S^T]$ uma base ortonormal para \mathbb{R}^d , como:

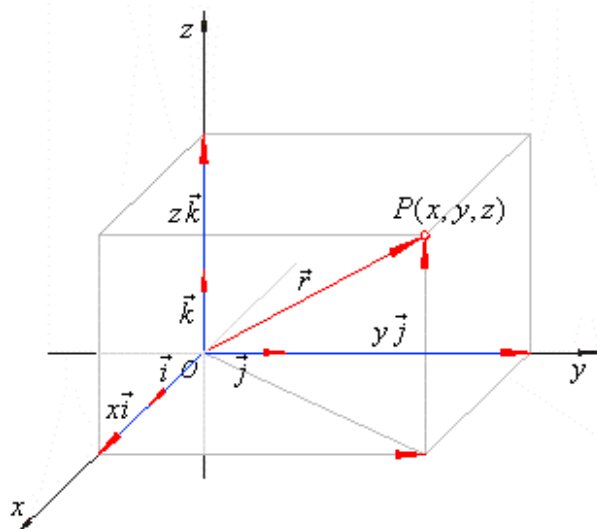
$T^T = [\vec{w}_1, \vec{w}_2, \dots, \vec{w}_k]$ (d componentes que desejamos reter no processo de redução)

$S^T = [\vec{w}_{k+1}, \vec{w}_{k+2}, \dots, \vec{w}_d]$ (componentes restantes que deve ser descartadas)

onde T denota o novo sistema de eixos coordenados (subespaço PCA) e S denota o subespaço eliminado durante redução de dimensionalidade.

O problema em questão pode ser resumido como: dado um espaço de entrada, deseja-se encontrar as direções \vec{w}_i que, ao projetar os dados, maximizam a variância retida na nova representação. Ou seja, queremos encontrar as direções em que o espalhamento dos dados é máximo. A pergunta que surge é: quais são essas direções? Primeiramente, note que podemos escrever $\vec{x} \in \mathbb{R}^d$ como (expansão na base ortonormal):

$$\vec{x} = \sum_{j=1}^d (\vec{x}^T \vec{w}_j) \vec{w}_j = \sum_{j=1}^d c_j \vec{w}_j \quad (c_j \text{ são os coeficientes da expansão})$$



Assim, o novo vetor $\vec{y} \in R^k$ pode ser obtido pela transformação:

$$\vec{y} = T \vec{x} \rightarrow \vec{y}^T = \vec{x}^T T^T = \sum_{j=1}^d c_j \vec{w}_j^T [\vec{w}_1, \vec{w}_2, \dots, \vec{w}_k] = [c_1, c_2, \dots, c_k]$$

uma vez que usando a propriedade de ortonormalidade (base ortonormal):

$$\vec{w}_i^T \vec{w}_j = \begin{cases} 1, & \text{se } i=j \\ 0, & \text{se } i \neq j \end{cases}$$

Dessa forma, busca-se uma transformação linear T que maximize a variância retida nos dados, ou seja, que maximize o seguinte critério:

$$J_1^{PCA}(T) = E[\|\vec{y}\|^2] = E[\vec{y}^T \vec{y}] = \sum_{j=1}^k E[c_j^2] \quad (*)$$

Note que $E[\vec{y}^T \vec{y}] = E[y_1^2] + E[y_2^2] + \dots + E[y_k^2]$ é justamente a soma das variâncias em cada eixo coordenado da nova representação. Como $c_j = \vec{x}^T \vec{w}_j$ (projeção de \vec{x} em \vec{w}_j), podemos escrever (substituindo c em *):

$$J_1^{PCA}(T) = \sum_{i=1}^k E[\vec{w}_j^T \vec{x} \vec{x}^T \vec{w}_j] = \sum_{i=1}^k \vec{w}_j^T E[\vec{x} \vec{x}^T] \vec{w}_j = \sum_{j=1}^k \vec{w}_j^T \Sigma_x \vec{w}_j \quad \text{sujeito a } \|\vec{w}_j\| = 1$$

em que $E[\vec{x} \vec{x}^T] = \Sigma_x$ denota a matriz de covariância dos dados observados.

O problema então consiste em resolver:

$$\underset{\vec{w}_j}{\operatorname{argmax}} \sum_{j=1}^k \vec{w}_j^T \Sigma_x \vec{w}_j \quad \text{sujeito a } \|\vec{w}_j\| = 1, \text{ para } j=1, 2, \dots, k$$

o que significa encontrar as k direções ortogonais \vec{w}_j que maximizam o somatório acima.

Trata-se portanto de um problema de otimização com restrições de igualdade. É sabido que a ferramenta matemática mais adequada para esse tipo de problema são os multiplicadores de Lagrange, utilizados para criar a função Lagrangiana, que incorpora as restrições diretamente na função objetivo.

Em matemática, em problemas de otimização, o método dos multiplicadores de Lagrange permite encontrar extremos (máximos e mínimos) de uma função de uma ou mais variáveis suscetíveis a uma ou mais restrições.

$$\max f(x, y) \quad \text{sujeito a } g(x, y) = c$$

O método consiste em utilizar essa nova variável (λ normalmente), chamada de multiplicador de Lagrange para definir uma nova função: a função Lagrangiana, assim definida:

$$L(x, y, \lambda) = f(x, y) - \lambda(g(x, y) - c)$$

Nesta função, o termo λ pode ser adicionado ou subtraído. Se $f(x,y)$ é um ponto de máximo para o problema original, então existe um λ tal que (x,y,λ) é um ponto estacionário para a função lagrangiana. Para múltiplas restrições no problema, a generalização é direta:

$$L(x,y,\lambda_1,\lambda_2,\dots,\lambda_m)=f(x,y)-\sum_{i=1}^m \lambda_i(g_i(x,y)-c_i)$$

Utilizando multiplicadores de Lagrange podemos reescrever a função objetivo do PCA como:

$$J_1^{PCA}(T,\lambda_j)=\sum_{j=1}^k \vec{w}_j^T \Sigma_x \vec{w}_j - \sum_{j=1}^k \lambda_j(\vec{w}_j^T \vec{w}_j - 1)$$

Derivando o funcional em relação a \vec{w}_j e igualando o resultado a zero, chega-se em:

$$\frac{\partial}{\partial \vec{w}_j} J(T,\lambda_j) = \Sigma_x \vec{w}_j - \lambda_j \vec{w}_j = 0$$

o que nos leva a $\Sigma_x \vec{w}_j = \lambda_j \vec{w}_j$ (problema de autovalores e autovetores)

Portanto, isso nos diz que os vetores \vec{w}_j da nova base devem ser autovetores da matriz de covariâncias.

→ \vec{w}_j : autovalores da matriz de covariância Σ_x

→ Vetores da base PCA são autovetores de Σ_x

Para otimizar o critério definido anteriormente, note que:

$$\underset{\vec{w}_j}{\operatorname{argmax}} \sum_{j=1}^k \vec{w}_j^T \Sigma_x \vec{w}_j = \underset{\vec{w}_j}{\operatorname{argmax}} \sum_{j=1}^k \vec{w}_j^T \lambda_j \vec{w}_j = \underset{\vec{w}_j}{\operatorname{argmax}} \sum_{j=1}^k \lambda_j \|\vec{w}_j\|^2 = \underset{\vec{w}_j}{\operatorname{argmax}} \sum_{j=1}^k \lambda_j$$

ou seja, devemos maximizar a soma dos k autovalores. Isso define a regra a ser utilizada pelo PCA: devemos escolher para compor a nova base os K autovetores da matriz de covariâncias associados aos K maiores autovalores.

Após aplicar PCA, os dados projetados na base PCA não exibem correlação, ou seja, a matriz de covariâncias torna-se diagonal.

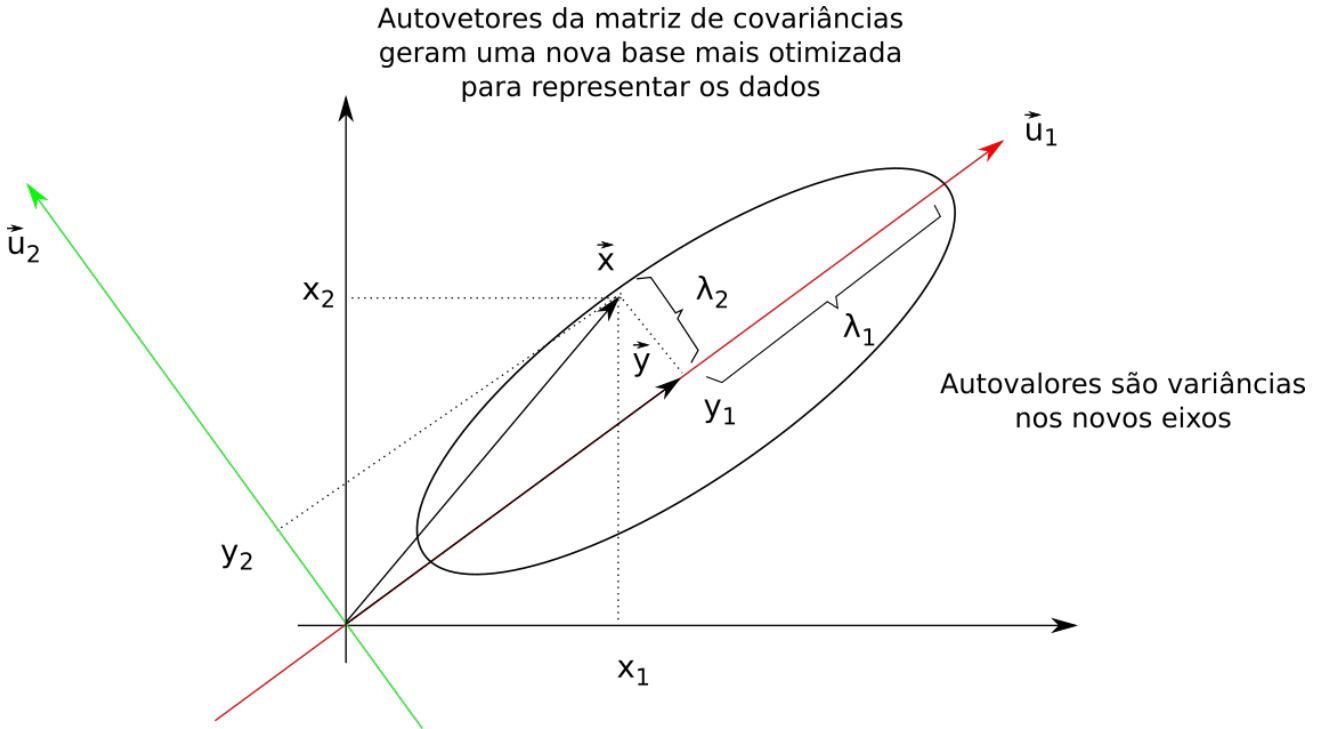
Def: Toda matriz A simétrica e positiva semidefinida possui uma decomposição $A = Q \Lambda Q^T$ onde Q é a matriz dos autovetores (nas colunas) e Λ é a matriz diagonal dos autovalores. Sabemos que matrizes de covariâncias são matrizes positivas semidefinidas.

Além disso, pode-se mostrar que se $\vec{y} = A \vec{x}$ (\vec{y} é uma transformação linear de \vec{x}) então a matriz de covariância de \vec{y} é transformada por $\Sigma_y = A^T \Sigma_x A$. Combinando os fatos definidos acima, a matriz de covariâncias dos dados transformados Σ_y é dada por: $\Sigma_y = A^T Q \Lambda Q^T A$ onde A é a matriz de projeção do PCA (autovetores de Σ_x), ou seja, é igual a Q e portanto:

$$\Sigma_y = Q^T Q \Lambda Q^T Q = \Lambda = \begin{bmatrix} \lambda_1 & 0 & \dots & 0 \\ 0 & \lambda_2 & \dots & 0 \\ 0 & 0 & \dots & 0 \\ 0 & 0 & \dots & \lambda_n \end{bmatrix}$$

sendo que $Q^T Q = I$ pois a base formada pelos autovetores é ortonormal. Dizemos portanto que após a transformação PCA, os dados encontram-se decorrelacionados (matriz de covariâncias é diagonal), o que significa que não há correlação entre os novos atributos gerados a partir do PCA (elimina as dependências existentes entre as variáveis)

Interpretação geométrica no caso 2D



PCA pela Minimização do erro quadrático médio

Um segundo critério otimizado pela representação PCA é a minimização do erro quadrático médio (MSE) entre os dados originais e a nova representação mais compacta. Em outras palavras, isso significa que o PCA é ótimo em termos de compactação, pois esse método descarta as componentes menos importantes no sentido de minimizar o erro de aproximação. Sendo assim, é possível definir o funcional a ser minimizado como:

$$J_2^{PCA}(T) = E[\|\vec{x} - \vec{y}\|^2] = E\left[\left\|\vec{x} - \sum_{j=1}^k (\vec{w}_j^T \vec{x}) \vec{w}_j\right\|^2\right]$$

Aplicando a definição de norma ao quadrado temos:

$$J_2^{PCA}(T) = E\left[\left(\vec{x} - \sum_{j=1}^k (\vec{w}_j^T \vec{x}) \vec{w}_j\right)^T \left(\vec{x} - \sum_{j=1}^k (\vec{w}_j^T \vec{x}) \vec{w}_j\right)\right]$$

Após a distributiva ser aplicada chegamos em:

$$J_2^{PCA}(T) = E\left[\vec{x}^T \vec{x} - \vec{x}^T \sum_{j=1}^k (\vec{w}_j^T \vec{x}) \vec{w}_j - \left(\sum_{j=1}^k (\vec{w}_j^T \vec{x}) \vec{w}_j\right)^T \vec{x} + \left(\sum_{j=1}^k (\vec{w}_j^T \vec{x}) \vec{w}_j\right)^T \left(\sum_{j=1}^k (\vec{w}_j^T \vec{x}) \vec{w}_j\right)\right]$$

Rearranjando os termos:

$$J_2^{PCA}(T) = E[\|\vec{x}\|^2] - E\left[\sum_{j=1}^k (\vec{w}_j^T \vec{x})(\vec{w}_j^T \vec{x})\right] - E\left[\sum_{j=1}^k \vec{w}_j^T (\vec{x}^T \vec{w}_j) \vec{x}\right] + E\left[\left(\sum_{j=1}^k \vec{w}_j^T (\vec{x}^T \vec{w}_j)\right) \left(\sum_{j=1}^k (\vec{w}_j^T \vec{x}) \vec{w}_j\right)\right]$$

Aplicando distributiva no último termo e rearranjando os valores esperados:

$$J_2^{PCA}(T) = E[\|\vec{x}\|^2] - \sum_{j=1}^k E[(\vec{w}_j^T \vec{x})^2] - \sum_{j=1}^k E[(\vec{w}_j^T \vec{x})^2] + \sum_{j=1}^k \sum_{l=1}^k E[(\vec{w}_j^T \vec{x})(\vec{x}^T \vec{w}_l) \vec{w}_j^T \vec{w}_l]$$

Notando que os vetores \vec{w}_j formam uma base ortonormal, o produto escalar é nulo para $j \neq l$

$$J_2^{PCA}(T) = E[\|\vec{x}\|^2] - \sum_{j=1}^k E[(\vec{w}_j^T \vec{x})^2] - \sum_{j=1}^k E[(\vec{w}_j^T \vec{x})^2] + \sum_{j=1}^k E[(\vec{w}_j^T \vec{x})^2]$$

chegamos finalmente em:

$$J_2^{PCA}(T) = E[\|\vec{x}\|^2] - \sum_{j=1}^k E[(\vec{w}_j^T \vec{x})^2]$$

Note que o primeiro termo não depende de \vec{w}_j , sendo uma constante. Então para minimizar o erro devemos maximizar o segundo termo, que nada mais é que a variância retida em cada novo eixo coordenado da base T:

$$J_2^{PCA}(T) = E[\|\vec{x}\|^2] - \sum_{j=1}^k E[\vec{w}_j^T \vec{x} \vec{x}^T \vec{w}_j] = E[\|\vec{x}\|^2] - \sum_{j=1}^k \vec{w}_j^T E[\vec{x} \vec{x}^T] \vec{w}_j = E[\|\vec{x}\|^2] - \sum_{j=1}^k \vec{w}_j^T \Sigma_x \vec{w}_j$$

Ou seja, caímos no mesmo problema resolvido anteriormente de maximizar a variância. Como esse problema já foi resolvido, sabemos que a solução é dada na forma de uma equação de autovalores e autovetores $\Sigma_x \vec{w}_j = \lambda_j \vec{w}_j$. Isso significa que voltando ao funcional acima temos a equivalência:

$$J_2^{PCA}(T) = E[\|\vec{x}\|^2] - \sum_{j=1}^k \vec{w}_j^T \lambda_j \vec{w}_j = E[\|\vec{x}\|^2] - \sum_{j=1}^k \lambda_j$$

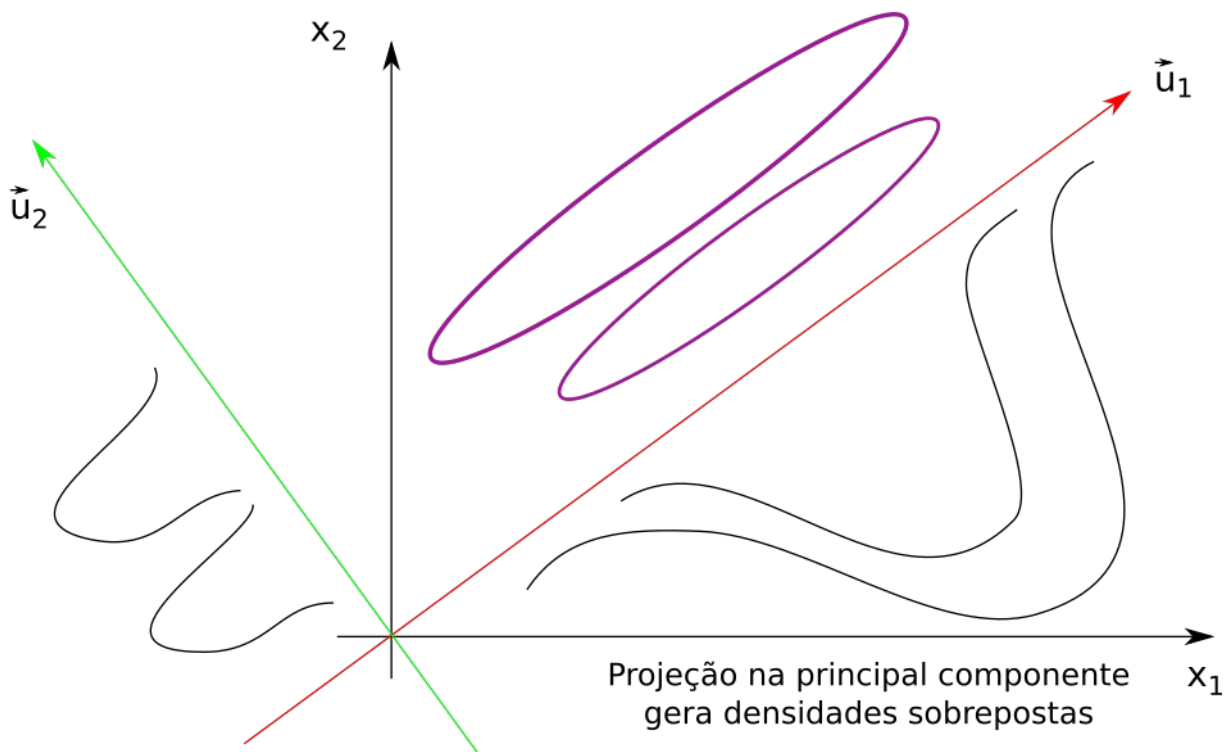
ou seja, para minimizar o erro quadrático médio, devemos selecionar os k autovetores associados aos k maiores autovalores da matriz de covariâncias, o que é exatamente a mesma conclusão obtida anteriormente.

Importante: Os dados precisam estar centralizados, ou seja, média deve ser nula para que a rotação dos eixos seja realizada de maneira correta (isso significa que $\vec{x}_i = \vec{x}_i - \vec{\mu}_x \quad \forall i$)

Limitações do PCA

Embora a redução de dimensionalidade implementada pelo PCA seja ótima do ponto de vista da compactação dos dados, pois minimiza o erro quadrático médio entre a representação original e a nova representação, ela não é ótima para a discriminação dos dados em classes. A figura a seguir ilustra um simples exemplo dessa limitação.

Projeção na componente de menor variância separa melhor



Algoritmo:

1. Ler dataset $X = \{\vec{x}_1, \vec{x}_2, \vec{x}_3, \dots, \vec{x}_n\}$ com $\vec{x}_i \in \mathbb{R}^d$
2. Computar o vetor média μ_x a matriz de covariâncias: $\Sigma_x = \frac{1}{n} \sum_{i=1}^n (\vec{x}_i - \mu_x)(\vec{x}_i - \mu_x)^T$
3. Obter autovalores e autovetores de Σ_x
4. Selecionar os k autovetores associados aos k maiores autovalores da matriz de covariâncias, denotados por $\vec{w}_1, \vec{w}_2, \dots, \vec{w}_k$
5. Definir a matriz de transformação $W_{PCA} = [\vec{w}_1, \vec{w}_2, \dots, \vec{w}_k]$ ($n \times k$)
6. Projetar dados na nova base: $\vec{y} = W_{PCA}^T \vec{x}_i$ com autovetores nas linhas de W_{PCA}^T , ou seja, a matriz transposta é $k \times n$ e \vec{x}_i vetor coluna $n \times 1$)

A seguir é apresentado um script em Python que lê um dataset 4D, computa e plota as duas componentes principais.

```
import numpy as np
import sklearn.datasets as skdata
import matplotlib.pyplot as plt

...
    Aplica PCA nos dados para reduzir dimensionalidade para d
    dados: matriz n x m, em que cada linha representa uma amostra
    d: inteiro menor que m
...
```

```
def PCA(dados, d):
    # Autovalores e autovetores da matriz de covariâncias
    v, w = np.linalg.eig(np.cov(dados.T))
    # Ordena os autovalores
    ordem = v.argsort()
    # Seleciona os d autovetores associados aos d maiores autovalores
    maiores_autovetores = w[:, ordem[-d:]]
    # Monta a matriz de projeção
    Wpca = maiores_autovetores
    # Realiza a projeção dos dados no subespaço PCA
    novos_dados = np.dot(Wpca.T, dados.T)

    return novos_dados

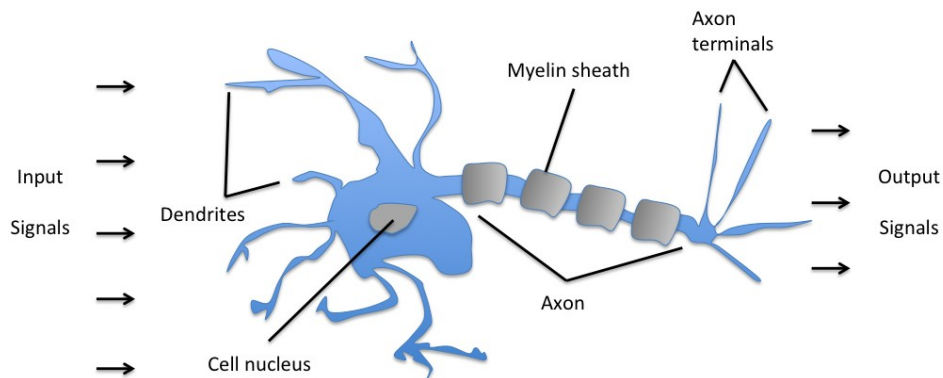
# Início do script
X = skdata.load_iris()
dados = X['data']

# Aplica PCA
dados_pca = PCA(dados, 2)

# Plota dados
plt.figure(1)
plt.plot(dados_pca[0,0:51], dados_pca[1,0:51], '*', color='blue')
plt.plot(dados_pca[0,51:101], dados_pca[1,51:101], '*', color='red')
plt.plot(dados_pca[0,101:151], dados_pca[1,101:151], '*', color='green')
plt.show()
```

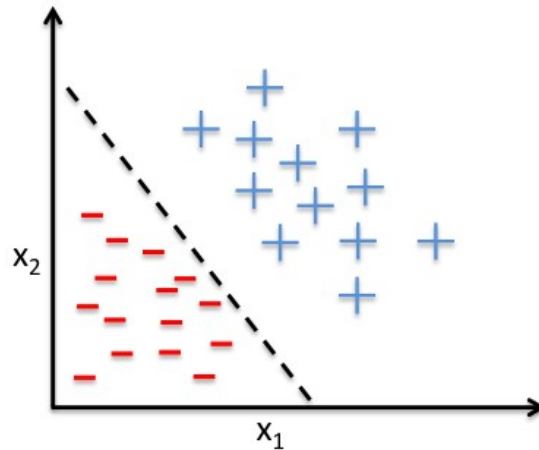
O Perceptron

Trata-se do primeiro modelo matemático baseado no conceito de que a unidade básica de processamento de informação no cérebro biológico são os neurônios. O estudo desse modelo primitivo de computação é importante pois ele define a base para o funcionamento de redes neurais artificiais utilizadas em tarefas de aprendizado profundo. É um classificador supervisionado binário. O Perceptron foi o primeiro algoritmo de aprendizado desenvolvido na história. Sua ideia inicial é baseada nos trabalhos pioneiros de McCulloch e Piits em 1943, que propuseram uma analogia entre neurônios biológicos e portas lógicas com saídas binárias. De maneira intuitiva, neurônios podem ser entendidos como as unidades básicas de redes neurais no cérebro biológico. O fluxo de informação é como segue: sinais chegam nos dendritos e são acumulados no corpo celular. Se o valor acumulado excede um determinado limiar, um sinal de saída é gerado pelo axônio. A figura a seguir mostra esse processo de maneira ilustrativa.



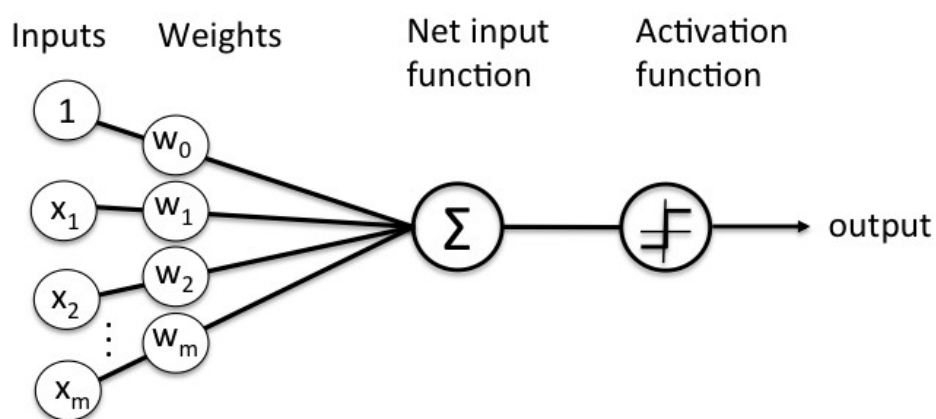
Schematic of a biological neuron.

Foi mais tarde, em 1957, que Rosenblatt publicou o primeiro conceito sobre a regra de treinamento de perceptrons. A ideia básica era desenvolver um algoritmo para aprender os pesos sinápticos do neurônio que eram multiplicados por um vetor de padrões de entrada para então decidir se o neurônio deve disparar ou não. No contexto de reconhecimento de padrões esse algoritmo seria útil para determinar se uma amostra x pertence ou não a uma determinada classe, ou seja, o neurônio dispararia se x pertencesse a classe 1 e não dispararia se pertencesse a classe 2.



Example of a linear decision boundary for binary classification.

Em resumo a ideia consiste em imitar como um neurônio no cérebro funciona: determinar condições matemáticas para que se possa decidir quando um ele deve disparar ou permanecer estático. Nas palavras, de Rosenblatt, o algoritmo perceptron fornece uma maneira objetiva de um neurônio aprender pesos sinápticos de modo a construir uma superfície de separação linear (hiperplano) que seja capaz de discriminar os dados em 2 classes ou categorias. A figura a seguir ilustra a arquitetura básica de um neurônio artificial do tipo perceptron.



Schematic of Rosenblatt's perceptron.

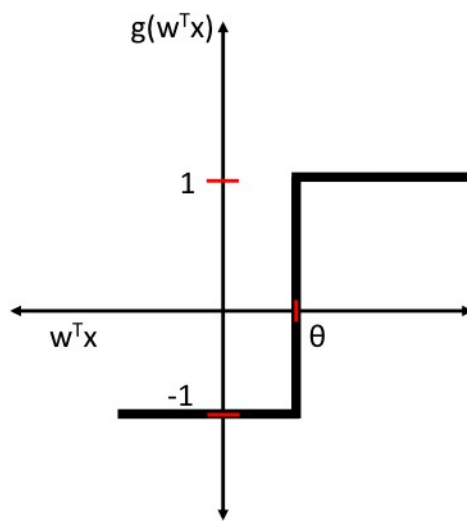
Fundamentação matemática

Iremos adotar a convenção de que no problema de classificação binário, a classe 1 ou positiva possui rótulo +1 enquanto que a classe 2 ou negativa possui rótulo -1.

A primeiro passo consiste na definição de uma função de ativação para o neurônio, responsável por dizer quando ele dispara ou não. Iremos definir a função $g(z)$ como sendo a função degrau unitário, ou seja, ela fornece como saída +1 se o valor de z é maior que θ e -1 se o valor de z é menor que θ , ou seja:

$$g(z) = \begin{cases} +1, & z \geq \theta \\ -1, & z < \theta \end{cases}$$

A figura a seguir ilustra essa função.



Unit step function.

O que o neurônio faz é multiplicar cada peso pela sua respectiva entrada e acumular. Em seguida, esse valor é passado para a função de ativação, que responde positivamente com +1 ou negativamente com -1. Sendo assim, temos:

$$z = w_1 x_1 + w_2 x_2 + \dots + w_k x_k = \sum_{i=1}^k w_i x_i = \vec{w}^T \vec{x}$$

onde $\vec{w}^T = [w_1, w_2, \dots, w_k]$ é o vetor de pesos e $\vec{x}^T = [x_1, x_2, \dots, x_k] \in R^k$ é um vetor de padrões (amostra do conjunto de dados a ser classificado).

Para simplificar a notação, é usual trazer θ para o lado esquerdo da equação e defini-lo como o componente zero do vetor de pesos, ou seja, $w_0 = -\theta$, fazendo $x_0 = 1$. Assim, a função $g(z)$ pode ser simplificada para:

$$g(z) = \begin{cases} +1, & z \geq 0 \\ -1, & z < 0 \end{cases}$$

e a variável z torna-se:

$$z = w_0 x_0 + w_1 x_1 + w_2 x_2 + \dots + w_k x_k = \sum_{i=0}^k w_i x_i = \vec{w}^T \vec{x}$$

Algoritmo de treinamento

Por convenção iremos adotar a seguinte notação:

$\{\vec{x}_i, d_i\}$ para $i=1, \dots, n$: denota o conjunto de treinamento com as amostras \vec{x}_i e os seus respectivos rótulos y_i
 $y_i = g(\vec{x}_i)$: denota a saída do perceptron para um vetor de entrada \vec{x}
 \vec{w} : denota o vetor de pesos

O algoritmo é sumarizado pelo seguinte conjunto de passos:

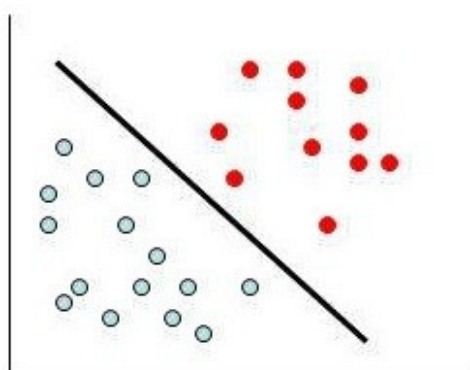
Passo 1: Inicialize o vetor de pesos com coeficientes aleatórios (uniforme $[0,1]$)

Passo 2: Para cada amostra \vec{x}_i do conjunto de treinamento faça:

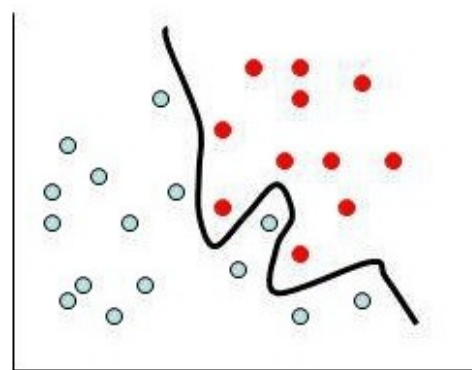
- Calcule a saída do neurônio: $y_i = g(\vec{x}_i)$
- Atualize os pesos sinápticos: $\vec{w} = \vec{w} + \alpha(d_i - y_i)\vec{x}_i$, onde $\alpha \in [0,1]$ é a taxa de aprendizado.

Passo 3: Repita o passo 2 um número finito de passos, ou até que o erro $\frac{1}{n} \sum_{j=1}^n |d_j - y_j|$ seja menor que um limiar γ pré estabelecido.

Pode-se mostrar que se o conjunto de dados é linearmente separável, o algoritmo perceptron converge para um vetor de pesos que define um hiperplano separador num número finito porém desconhecido de passos.



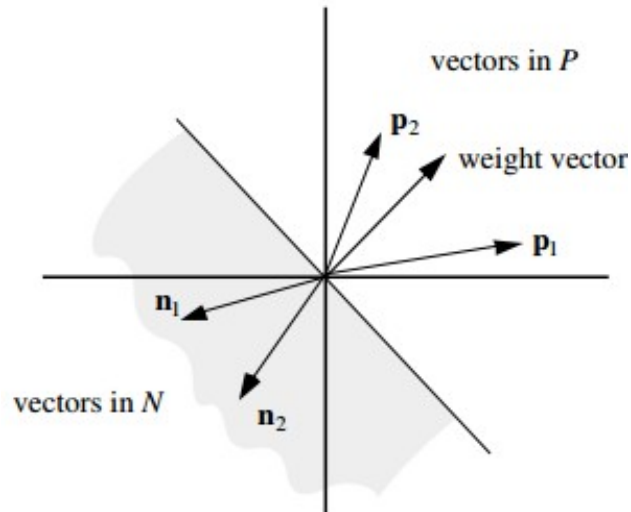
**Linearmente
separável**



**Não linearmente
separável**

Interpretação geométrica

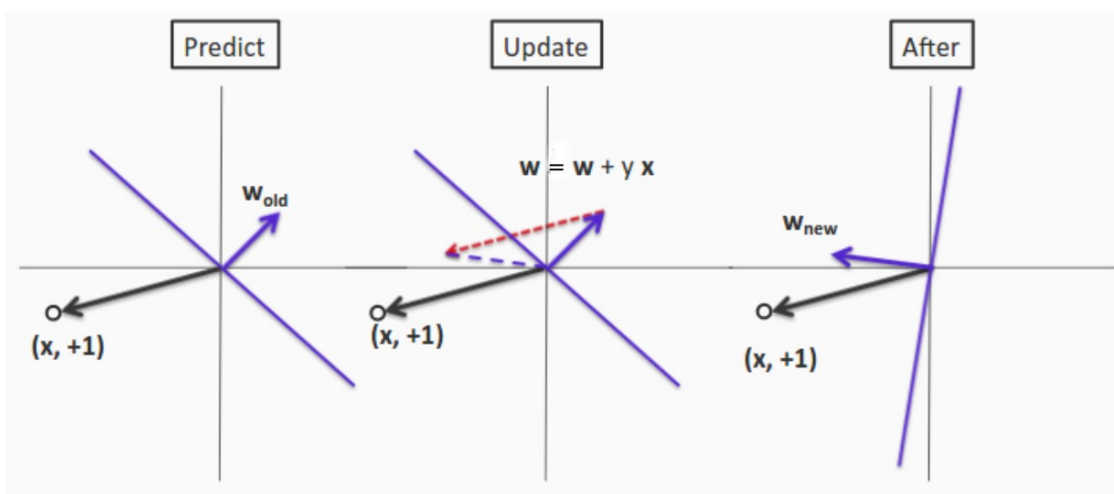
A equação $\vec{w}^T \vec{x} = w_1 x_1 + \dots + w_k x_k - \theta = 0$ define um hiperplano normal ao vetor de pesos \vec{w} e deslocado de θ em relação a origem (bias). A figura a seguir ilustra esse cenário em um caso em que $\theta = 0$ (hiperplano passa pela origem).



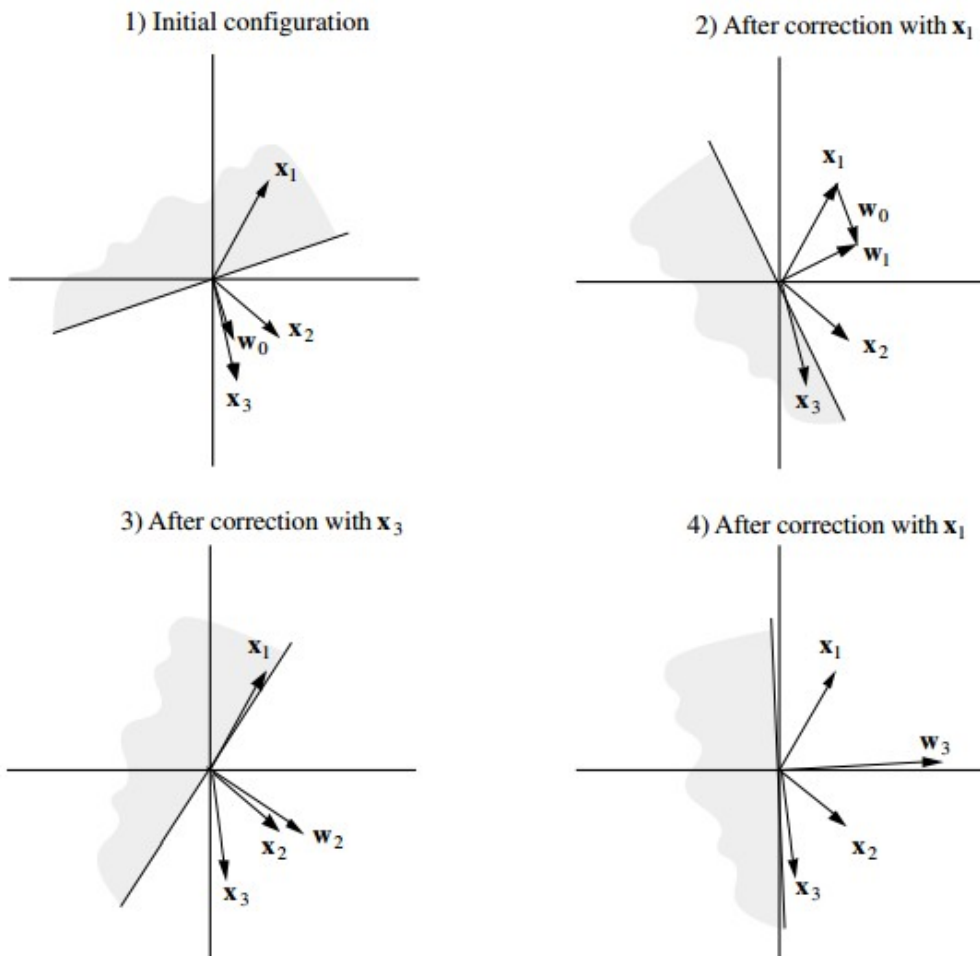
Suponha que $\alpha = 0.5$ de modo que a regra de atualização dos pesos é dada por:

$$\vec{w} = \vec{w} + 0.5(d_i - y_i)\vec{x}_i$$

Note que sempre que a saída do neurônio (y_i) é igual ao rótulo da amostra (d_i), o vetor de pesos permanece inalterado. Se a amostra é positiva, ou seja, $d_i = 1$ mas o neurônio prediz $y_i = -1$, então o novo vetor de pesos será $\vec{w} = \vec{w} + \vec{x}_i$, e o hiperplano será rotacionado de modo que a amostra em questão seja positiva. Por outro lado, se a amostra é negativa, ou seja, $d_i = -1$ mas o neurônio prediz $y_i = 1$, então o novo vetor de pesos será $\vec{w} = \vec{w} - \vec{x}_i$, e o hiperplano será rotacionado de modo que a amostra em questão seja negativa. A figura a seguir ilustra o processo de rotação do plano separador ao se classificar uma amostra de maneira errônea.



O processo segue até que todas as amostras positivas encontram-se de um lado do hiperplano e todas as amostras negativas encontram-se do outro lado.



O que percebemos ao final do algoritmo é que se \vec{w}^T é um hiperplano separador, então:

$$\vec{w}^T \vec{x} \geq 0 \quad \text{para todo } \vec{x} \in P \quad (\text{classe positiva, } y = 1)$$

$$\vec{w}^T \vec{x} < 0 \quad \text{para todo } \vec{x} \in N \quad (\text{classe negativa, } y = -1)$$

Teorema: Convergência do perceptron de incrementos fixos

Sejam os conjuntos de treinamento P e N linearmente separáveis. Considere que as entradas apresentadas ao perceptron são originadas desses conjuntos. Então, o perceptron converge depois de n_0 iterações no sentido de que

$$\vec{w}(n_0) = \vec{w}(n_0+1) = \vec{w}(n_0+2) = \dots$$

é um vetor solução para $n_0 \leq n_{max}$.

Prova:

Iremos considerar, sem perda de generalidade, que inicialmente o vetor de pesos é nulo, ou seja, $\vec{w}(0) = \vec{0}$. Suponha ainda que para $n = 1, 2, 3, \dots$, $\vec{w}^T(n) \vec{x}(n) < 0$ (perceptron prediz $y = -1$) e os vetores $\vec{x}(n)$ pertençam ao conjunto P (classe positiva). Isso significa que o perceptron

inicialmente classifica incorretamente os vetores $\vec{x}(1), \vec{x}(2), \dots, \vec{x}(n)$. Então, considerando a taxa de aprendizado $\alpha = 0.5$, a correção do vetor de pesos será:

$$\vec{w}(n+1) = \vec{w}(n) + \vec{x}(n) \quad \text{para } \vec{x}(n) \in P$$

Dado a condição inicial $\vec{w}(0) = \vec{0}$ é possível iterativamente resolver essa equação:

$$\begin{aligned} \vec{w}(1) &= \vec{x}(0) \\ \vec{w}(2) &= \vec{x}(1) + \vec{x}(0) \\ \vec{w}(3) &= \vec{x}(2) + \vec{x}(1) + \vec{x}(0) \\ &\dots \\ \vec{w}(n+1) &= \vec{x}(1) + \vec{x}(2) + \dots + \vec{x}(n) \quad (*) \end{aligned}$$

Como as classes P e N são linearmente separáveis, existe uma solução \vec{w}_o para a qual $\vec{w}_o^T \vec{x}(n) > 0$ para todos os vetores pertencentes ao conjunto P. Para essa solução \vec{w}_o podemos definir um número positivo α como:

$$\alpha = \min_{\vec{x}(n) \in P} \vec{w}_o^T \vec{x}(n) \quad (\text{menor valor de produto escalar para vetores em P é sempre positivo}) (**)$$

Assim, multiplicando ambos os lados de (*) por \vec{w}_o^T :

$$\vec{w}_o^T \vec{w}(n+1) = \vec{w}_o^T \vec{x}(1) + \vec{w}_o^T \vec{x}(2) + \dots + \vec{w}_o^T \vec{x}(n)$$

De acordo com a definição dada em (**), temos:

$$\vec{w}_o^T \vec{w}(n+1) \geq n\alpha \quad (3)$$

Em seguida, fazemos uso de uma desigualdade conhecida como desigualdade de Cauchy-Schwarz. Essa desigualdade nos diz que $\|\vec{u} \cdot \vec{v}\|^2 \leq \|\vec{u}\|^2 \cdot \|\vec{v}\|^2$. Aplicando em relação a \vec{w}_o^T e $\vec{w}(n+1)$:

$$\|\vec{w}_o\|^2 \|\vec{w}(n+1)\|^2 \geq [\vec{w}_o^T \vec{w}(n+1)]^2 \quad (4)$$

onde o produto $\vec{w}_o^T \vec{w}(n+1)$ é uma quantidade escalar. A partir das equações (3) e (4) é fácil perceber que por transitividade temos:

$$\|\vec{w}_o\|^2 \|\vec{w}(n+1)\|^2 \geq n^2 \alpha^2$$

ou de maneira equivalente

$$\|\vec{w}(n+1)\|^2 \geq \frac{n^2 \alpha^2}{\|\vec{w}_o\|^2} \quad (5)$$

Note que definimos um limite inferior para a norma do vetor de pesos. A princípio, pode parecer alarmante, pois quando o número de iterações cresce, a norma aumenta de maneira quadrática.

Agora, vamos seguir uma outra rota de desenvolvimento. Reescrevendo a primeira equação em termos da variável k:

$$\vec{w}(k+1) = \vec{w}(k) + \vec{x}(k) \quad \text{para } k = 1, 2, \dots, n \text{ e } \vec{x}(k) \in P$$

Tomando a norma ao quadrado de ambos os lados da equação anterior:

$$\|\vec{w}(k+1)\|^2 = \|\vec{w}(k)\|^2 + \|\vec{x}(k)\|^2 + 2\vec{w}^T(k)\vec{x}(k)$$

Mas $\vec{w}^T(k)\vec{x}(k) < 0$ pois $\vec{x}(k) \in P$. Então, podemos deduzir que ao remover uma quantidade negativa do lado direito temos:

$$\|\vec{w}(k+1)\|^2 \leq \|\vec{w}(k)\|^2 + \|\vec{x}(k)\|^2$$

ou equivalentemente

$$\|\vec{w}(k+1)\|^2 - \|\vec{w}(k)\|^2 \leq \|\vec{x}(k)\|^2$$

Adicionando as desigualdades para $k=0,1,2,\dots,n$, o lado esquerdo fica:

$$(\|\vec{w}(1)\|^2 - \|\vec{w}(0)\|^2) + (\|\vec{w}(2)\|^2 - \|\vec{w}(1)\|^2) + (\|\vec{w}(3)\|^2 - \|\vec{w}(2)\|^2) + \dots + (\|\vec{w}(n+1)\|^2 - \|\vec{w}(n)\|^2)$$

Note que após cancelar os termos positivos e negativos, a expressão simplifica-se para:

$$\|\vec{w}(n+1)\|^2 - \|\vec{w}(0)\|^2$$

Mas como a condição inicial é $\vec{w}(0) = \vec{0}$, temos que o lado esquerdo fica $\|\vec{w}(n+1)\|^2$. O lado esquerdo da desigualdade fica:

$$\|\vec{x}(0)\|^2 + \|\vec{x}(1)\|^2 + \|\vec{x}(2)\|^2 + \dots + \|\vec{x}(n)\|^2$$

Como $\vec{x}(0)$ não é definida, o lado esquerdo da desigualdade fica:

$$\sum_{k=1}^n \|\vec{x}(k)\|^2$$

e finalmente chegamos na desigualdade:

$$\|\vec{w}(n+1)\|^2 \leq \sum_{k=1}^n \|\vec{x}(k)\|^2$$

Definindo β como

$$\beta = \max_{\vec{x}(k) \in P} \|\vec{x}(k)\|^2$$

podemos escrever

$$\|\vec{w}(n+1)\|^2 \leq n\beta \quad (6)$$

o que mostra que a norma Euclidiana do vetor de pesos é sempre inferior a uma constante vezes o número de iterações n . A equação (6) define um limite superior para a norma do vetor de pesos.

Note que a equação (5) define um limite inferior enquanto que a equação (6) define um limite superior. Observe que ambas as equações devem ser satisfeitas simultaneamente, mas a pergunta que surge é: para quais valores de n elas são consistentes?

É impossível satisfazer as equações (5) e (6) para valores muito grandes de n . Para tais valores o limite inferior será muito alto e o limite superior será muito baixo, gerando uma inconsistência.

Portanto, devemos ter um valor máximo n_{max} para o qual (5) e (6) são simultaneamente satisfeitas com igualdade:

$$\frac{n_{max}^2 \alpha^2}{\|\vec{w}_o\|^2} = n_{max} \beta$$

Resolvendo a expressão em n_{max} dado um vetor solução \vec{w}_o temos:

$$n_{max} = \frac{\beta \|\vec{w}_o\|^2}{\alpha^2}$$

Provamos que existe um valor de n finito, porém desconhecido a priori, após o qual o vetor de pesos não mais é atualizado, ou seja, na iteração n_{max} atinge-se a correta classificação dos padrões.

A seguir é apresentado um script em Python que implementa a regra de treinamento do perceptron e aplica para um problema de classificação linearmente separável 2D, plotando as amostras bem como o plano separador.

```
from pylab import rand, plot, show, norm

''' Implementar funções do perceptron
    * response: dada uma entrada calcula a saída
    * updateWeights: atualiza os pesos
    * train: treinamento
'''

""" Calcula saída do perceptron """
def response(w, x):
    y = x[0]*w[0]+x[1]*w[1]      # produto interno de w e x
    if y >= 0:
        return 1
    else:
        return -1

"""
    Atualiza os pesos, w com a regra
    w(t+1) = w(t) + learningRate*(d-r)*x
    onde d é a saída desejada e r a resposta do perceptron
    iterError é a diferença (d-r)
"""
def updateWeights(w, x, iterError):
    w[0] += learningRate*iterError*x[0]
    w[1] += learningRate*iterError*x[1]
```

```

"""
Treina o perceptron com os vetores
Cada vetor deve conter 3 elementos, sendo que
o terceiro elemento (x[2]) deve ser o rótulo (saída desejada)
"""
def train(w, data):
    learned = False
    iteration = 0

    while not learned:
        globalError = 0.0

        for x in data: # para cada amostra
            r = response(w, x)

            if x[2] != r: # se temos um resposta errada
                iterError = x[2] - r # desejado - resposta
                updateWeights(w, x, iterError)
                globalError += abs(iterError)

            iteration += 1

        if globalError == 0.0 or iteration >= 100: # parada
            print('iterations', iteration)
            learned = True # encerra treinamento

"""
Gera um conjunto 2D linearmente separável com n amostras.
O terceiro elemento do vetor é a saída desejada (rótulo da classe)
"""
def generateData(n):
    xb = (rand(n)*2 - 1)/2 - 0.5
    yb = (rand(n)*2 - 1)/2 + 0.5
    xr = (rand(n)*2 - 1)/2 + 0.5
    yr = (rand(n)*2 - 1)/2 - 0.5
    inputs = []
    for i in range(len(xb)):
        inputs.append([xb[i],yb[i],1])
        inputs.append([xr[i],yr[i],-1])
    return inputs

""" INÍCIO DO SCRIPT """
""" Inicialização dos parâmetros do perceptron """
w = rand(2)*2-1 # vetor pesos inicial (aleatórios entre -1 e 1)
learningRate = 0.5 # taxa de aprendizado

trainset = generateData(30) # geração do conjunto de treinamento
train(w, trainset) # treina com os dados gerados
testset = generateData(20) # gera um conjunto de testes

# Etapa de teste do perceptron, após estar treinado
for x in testset:
    r = response(w, x)
    if r != x[2]: # se a resposta não é correta
        print('error')

```

```

    if r == 1:
        plot(x[0], x[1], 'ob')
    else:
        plot(x[0], x[1], 'or')

# plota a superfície de separação
# ortogonal ao vetor de pesos
n = norm(w)
ww = w/n
ww1 = [ww[1], -ww[0]]
ww2 = [-ww[1], ww[0]]
plot([ww1[0], ww2[0]], [ww1[1], ww2[1]], '--k')
show()

```

Bibliografia

- Coelho, F. C. Computação Científica com Python: Uma introdução à programação para cientistas, 2007.
- Robert Johansson. Scientific Computing in Python, 2014 - disponível em: <https://github.com/jrjohansson/scientific-python-lectures/blob/master/Scientific-Computing-with-Python.pdf>
- Python Scientific Lecture Notes, Editors: Valentin Haenel, Emmanuelle Gouillart, Gael Varoquaux, 2013 - disponível em <http://scipy-lectures.github.io/>
- Hans Fangohr. Introduction to Python for Computational Science and Engeneering: A Begginer's Guide, 2014 - disponível em <http://www.southampton.ac.uk/~fangohr/training/python/pdfs/Python-for-Computational-Science-and-Engineering.pdf>
- Michael T. Heath. Scientific Computing: An Introductory Survey, McGraw-Hill, New York, 2002.
- Hans Petter Langtangen. A Primer on Scientific Programming with Python (Texts in Computational Science and Engineering), Springer, 2009.
- Hans Petter Langtangen. Python Scripting for Computational Science (Texts in Computational Science and Engineering), Springer, 2009.
- M. A. Gomes Ruggiero, V. L. da Rocha Lopes. Cálculo Numérico - Aspectos Teóricos e Computacionais, 2ª edição, Editora Pearson, 1997.
- M.C. Cunha. Métodos Numéricos. 2a edição, Editora da Unicamp, 2000.
- N.B. Franco. Cálculo Numérico. Pearson Prentice Hall, 2007.
- Richard L. Burden e J. Douglas Faires, Análise Numérica, Cengage Learning, Tradução da 8. Ed. Americana, 2008
- A. Quarteroni, F. Saleri. Cálculo Científico - Com MATLAB e Octave.
- D. S. Watkins, Fundamentals of Matrix Computations, New Jersey: John Wiley & Sons, 3. Ed, 2010
- Oppenheim, A. V. Sinais e sistemas, Pearson, 2ª ed., 2010.
- R. O. Duda, P. E. Hart, and D. G. Stork, Pattern Classification, 2nd ed. (Wiley-Interscience, 2000) p. 688 pages.
- T. Hastie, R. Tibshirani, and J. Friedman, The Elements of Statistical Learning, 2nd ed. (Springer, 2009).
- T. Y. Young and T. W. Calvert, Classification, Estimation and Pattern Recognition (Elsevier, 1974).
- S. Haykin. Neural Networks and Learning Machines, Prentice Hall, 3ª ed., 2008.

Sobre o autor

Alexandre L. M. Levada é bacharel em Ciências da Computação pela Universidade Estadual Paulista “Júlio de Mesquita Filho” (UNESP), mestre em Ciências da Computação pela Universidade Federal de São Carlos (UFSCar) e doutor em Física Computacional pela Universidade de São Paulo (USP). Atualmente é professor adjunto no Departamento de Computação da Universidade Federal de São Carlos e seus interesses em pesquisa são: filtragem de ruído em imagens e aprendizado de métricas via redução de dimensionalidade para problemas de classificação de padrões. Para maiores detalhes: <https://sites.google.com/view/alexandre-levada/>

“Experiência não é o que acontece com um homem; é o que ele faz com o que lhe acontece”
(Aldous Huxley)