Programação Científica em Julia: Fundamentos básicos e implementações computacionais

Technical Report · May 2021

DOI: 10.13140/RG.2.2.15300.76167

1 author:

Alexandre L M Levada
Universidade Federal de São Carlos
131 PUBLICATIONS **417** CITATIONS

SEE PROFILE

Departamento de Computação

Centro de Ciências Exatas e Tecnologia

Universidade Federal de São Carlos

Programação Científica em Julia

Fundamentos básicos e implementações computacionais

Prof. Alexandre Luis Magalhães Levada

Email: alexandre.levada@ufscar.br

Sumário

Prefácio....3
Prólogo: A linguagem Julia....4
Aula 1: O problema da precisão numérica....6
Aula 2: Recorrência Logística (Logistic map)....9
Aula 3: Autômatos celulares e o game of life....15
Aula 4: O método de Newton....23
Aula 5: O método da secante....27
Aula 6: O método de Simpson....33
Aula 7: Sistemas lineares – o método da eliminação de Gauss....37
Aula 8: Sistemas lineares – métodos iterativos....41
Aula 9: Regressão linear....47
Aula 10: Regressão quadrática....56
Aula 11: Regressão polinomial....59
Aula 12: Interpolação polinomial....63
Aula 13: Convolução e a filtragem linear de imagens....74
Aula 14: Filtragem de mínimos quadrados – o filtro de Wiener....87
Aula 15: Filtragem não local de imagens – o filtro NLM....93
Aula 16: Análise de Componentes Principais (PCA)....97
Bibliografia....106
Sobre o autor....107

“Experiência não é o que acontece com um homem; é o que ele faz com o que lhe acontece”
(Aldous Huxley)

Prefácio

Esta apostila sobre programação científica em Julia é um material recomendado para estudantes de cursos da área de exatas que desejam aprender fundamentos básicos de matemática, computação e estatística aplicadas. Cada aula é composta por uma parte teórica, em que são discutidos os conceitos mais abstratos e uma parte prática, em que são apresentados implementações em linguagem Julia sobre cada um dos métodos discutidos. Não há exigência de conhecimento prévio, de modo que o material pode ser utilizado tanto por iniciantes quanto pesquisadores de outras áreas da ciência.

O foco deste material não é ensinar como programar em Julia, mas sim utilizar essa linguagem como uma poderosa ferramenta computacional para implementar os métodos numéricos aqui apresentados. Para os interessados em aprender a programar em Julia, recomendamos a apostila:

Introdução a Computação em Julia: Problemas e Aplicações.
Disponível em: dx.doi.org/10.13140/RG.2.2.15777.53605/1

A justificativa da escolha da linguagem Julia se deve a dois motivos principais: 1) Julia é uma linguagem de programação com uma sintaxe e recursos muito bons para programação matemática, sendo mais intuitiva do que Python, uma vez que vetores, matrizes e funções anônimas são estruturas diretamente incorporadas nas definições básicas da linguagem; e 2) os programas escritos em Julia são muito mais rápidos do que os scripts nativos em Python, fazendo com que o tempo de execução dos métodos seja significativamente menor.

Por fim, todo conteúdo aqui apresentado é fruto de mais de 10 anos de notas de aula produzidas pelo professor durante sua carreira como docente no Departamento de Computação da Universidade Federal de São Carlos, de modo que pode ser utilizado sem fins lucrativos por qualquer pessoa interessada.

Alexandre L. M. Levada.

Prólogo: A linguagem Julia

Julia *é uma linguagem de programação dinâmica de alto nível projetada para atender os requisitos da computação de alto desempenho numérico e científico, sendo também eficaz para a programação de propósito geral.* (Wikipedia)

Nesse curso, optamos pela linguagem Julia, 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. Sua sintaxe é agradável e intuitiva para iniciantes que não estão acostumados com a rigidez sintática das linguagens de programação, sendo bastante próxima da linguagem Python, que também é uma excelente escolha de linguagem para um curso introdutório de programação. Em comparação com a linguagem Python, a principal vantagem de Julia é o desempenho, ou seja, sua velocidade de execução. Os programas escritos em Julia, em geral, são mais rápidos do que os scripts em Python. Isso se deve ao fato de que enquanto a linguagem Python é interpretada, a linguagem Julia utiliza um esquema de compilação Just-In-Time (JIT). Na prática, isso significa que a primeira vez que o código é carregado no ambiente, leva-se um tempo considerável para executá-lo, porém depois a execução torna-se muito mais rápida, podendo ser comparada à velocidade obtida pelos programas construídos utilizando a linguagem C.

- Porque Julia?

De acordo com a página da linguagem Julia na Wikipedia e do site oficial da linguagem Julia, as principais características da linguagem são:

- Despacho múltiplo (*multiple dispatch*): provê capacidade de definir o comportamento da função através de muitas combinações de tipos de argumento.
- Tipagem dinâmica: tipos de documentação, otimização e despacho.
- Boa performance, que se aproxima de linguagens com tipagem estática, exemplo a linguagem C.
- Possui um gerenciador de pacotes prático e simples de usar.
- Possui macros como Lisp e outros pacotes de meta programação.
- Fornece chamada para funções da linguagem Python: para isso utiliza-se o pacote PyCall
- Possui APIs especiais para chamada de funções em C diretamente.
- Projetado para paralelismo e computação distribuída.
- Detém uma geração eficiente de código, especializado para diferentes tipos de argumentos.
- Apresenta uma forma elegante e extensivo para tipos numéricos.
- Inclui suporte eficiente para Unicode, incluindo UTF-8
- Licença pela MIT, livre e open source.

Dentre as vantagens de se aprender Julia, podemos citar a boa variedade de bibliotecas existentes para a linguagem, tornando-a uma linguagem ideal para computação científica. É possível desenvolver aplicações que vão desde métodos matemáticos numéricos (cálculo numérico e otimização principalmente), processamento de sinais e imagens (pois é muito fácil trabalhar com matrizes em Julia), aprendizado de máquina (pois a base é toda construída em cima de operações matemáticas da álgebra linear e da estatística) até aplicações mais comerciais, como sistemas de bancos de dados e redes de computadores.

Instalando Julia

Para instalar o compilador JIT para Julia, mais o REPL, um ambiente integrado que funciona como um interpretador da linguagem e nos permite interagir em tempo real com comandos, basta acessar:

https://julialang.org/downloads/

Recomenda-se instalar a última versão estável, que na data em que esse material foi desenvolvido era a versão 1.6.0, bastando escolher a versão compatível com o seu sistema operacional: Windows, macOS ou Linux, e a versão 32 ou 64 bits (se seu computador não é extremamente antigo, ele é 64 bits).

Para editar os programas, é possível utilizar qualquer editor de texto básico que nos permita salvar um arquivo com a extensão .jl

Uma alternativa muito interessante que vem sendo considerada por muitos programadores como a melhor ferramenta de desenvolvimento em Julia é o Visual Studio Code, ou VSCODE. Ele pode ser baixado sem custos em:

https://code.visualstudio.com/

Para que ele reconheça o compilador JIT Julia e o REPL, é preciso realizar algumas configurações básicas no ambiente. Alguns vídeos didáticos que ensinam como configurar o VSCODE para a linguagem Python podem ser encontrados a seguir:

https://www.youtube.com/watch?v=oi5dZxPGNlk (inglês)

https://www.youtube.com/watch?v=C3ro2b5tQws (inglês)

https://www.youtube.com/watch?v=J5uMzyaniag (português)

Plataformas Julia para desenvolvimento

Para desenvolver aplicações científicas em Julia, é conveniente instalar um ambiente de programação em que as principais bibliotecas para computação científica estejam presentes. A principal plataforma Julia para computação científica, nesse momento é a plataforma JuliaPro, que é composta pelo editor Juno e uma série de bibliotecas auxiliares.

JuliaPro: essa plataforma pode ser baixada sem custos (na versão padrão) em

https://juliacomputing.com/products/juliapro/

Repl.it e JuliaHub

Uma opção muito interessante são as plataformas Julia na nuvem: Repl.it e JuliaHub
Você pode desenvolver e armazenar seus códigos de maneira totalmente online sem a necessidade de instalar em sua máquina um ambiente de desenvolvimento local.

Aula 1: O problema da precisão numérica

A representação de números fracionários por computadores digitais pode levar a problemas de precisão numérica. Sabemos que um número na base 10 (decimal) é representado como:

$23457 = 2 \times 10^4 + 3 \times 10^3 + 4 \times 10^2 + 5 \times 10^1 + 7 \times 10^0$

Analogamente, um número binário (base 2) pode ser representado como:

$110101 = 1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 53$

O bit mais a direita é o menos significativo e portanto o seu valor é $1 \times 2^0 = 1$
O segundo bit a partir da direita tem valor de $0 \times 2^1 = 0$
O terceiro bit a partir da direita tem valor de $1 \times 2^2 = 4$
O quarto bit a partir da direita tem valor de $0 \times 2^3 = 0$
O quinto bit a partir da esquerda tem valor de $1 \times 2^4 = 16$
Por fim, o bit mais a esquerda tem valor de $1 \times 2^5 = 32$
Somando tudo temos: $1 + 4 + 16 + 32 = 5 + 48 = 53$.

Essa é a regra para convertermos um número binário para sua notação decimal.

Veremos agora o processo inverso: como converter um número decimal para binário. O processo é simples. Começamos dividindo o número decimal por 2:

53 / 2 = 26 e sobra resto **1** → esse 1 será nosso bit mais a direita (menos significativo no binário)

Continuamos o processo até que a divisão por 2 não seja mais possível:

26 / 2 = 13 e sobra resto **0** → esse 0 será nosso segundo bit mais a direita no binário

13 / 2 = 6 e sobra resto **1** → esse 1 será nosso terceiro bit mais a direita no binário

6 / 2 = 3 e sobra resto **0** → esse 0 será nosso quarto bit mais a direita no binário

3 / 2 = 1 e sobra resto **1** → esse 1 será nosso quinto bit mais a direita no binário

1 / 2 = 0 e sobra resto **1** → esse 1 será o nosso último bi (mais a esquerda)

Note que de agora em diante não precisamos continuar com o processo pois

0 / 2 = 0 e sobra 0
0 / 2 = 0 e sobra 0

ou seja a esquerda do sexto bit teremos apenas zeros, e como no sistema decimal, zeros a esquerda não possuem valor algum. Portanto, 53 em decimal equivale a 110101 em binário.

Com números fracionários, a ideia é similar:

Na base 10:
$456{,}78 = 4 \times 10^2 + 5 \times 10^1 + 6 \times 10^0 + 7 \times 10^{-1} + 8 \times 10^{-2}$

Na base 2:
$101{,}101 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-2} = 4 + 1 + 0.5 + 0.125 = 5{,}625$

Na base 10, dividir por 10 significa deslocar a vírgula uma casa para a esquerda e multiplicar por 10 significa deslocar a vírgula para direita, acrescentando ou removendo zeros quando for o caso.

2317 / 10 = 231,7
231,7 / 10 = 23,17
23,17 / 10 = 2,317

Com números binários a ideia é a mesma. Mover a vírgula para a esquerda significa divisão por 2 e mover a vírgula para direita significa multiplicação por 2

28 = 11100
14 = 1110
7 = 111
3,5 = 11,1
1,75 = 1,11

Para armazenar números reais em computadores, é preciso representá-los na base 2 (binário)

No caso de números inteiros, a conversão é direta

25 = 11001 pois

25 / 2 resulta em 12 com resto 1
12 / 2 resulta em 6 com resto 0
6 / 2 resulta em 3 com resto 0
3 / 2 resulta em 1 com resto 1
1 / 2 resulta em 0 com resto 1

No caso de números reais, o processo é similar

5,625

Primeiramente, devemos dividir o número em 2 partes: parte inteira e parte fracionária
A conversão é feita independentemente para cada parte. Assim, primeiro devemos converter o número 5

5 / 2 = 2 com resto 1
2 / 2 = 1 com resto 0
1 / 2 = 1 com resto 1

Então, temos que 5 = 101

Em seguida iremos trabalhar com a parte fracionária: 0,625. Nesse caso ao invés de dividir por 2, iremos multiplicar por 2 a parte fracionária e tomar a parte inteira do resultado (a esquerda da vírgula), repetindo o processo até que não se tenha mais casas decimais depois da vírgula.

0,625 x 2 = 1,25 → 1 (primeira casa fracionária)
0,25 x 2 = 0,5 → 0 (segunda casa)
0,5 x 2 = 1,0 → 1 (terceira casa)

Assim, temos que 0,625 = 0,101 e portanto 5,625 = 101,101

Porém, em alguns casos, alguns problemas podem surgir. Por exemplo, suponha que desejamos armazenar num computador o número 0,8 na base 10. Para isso, devemos proceder da forma descrita anteriormente.

0,8 x 2 = 1,6 → 1
0,6 x 2 = 1,2 → 1
0,2 x 2 = 0,4 → 0
0,4 x 2 = 0,8 → 0
0,8 x 2 = 1,6 → 1
0,6 x 2 = 1,2 → 1
0,2 x 2 = 0,4 → 0
0,4 x 2 = 0,8 → 0
0,8 x 2 = 1,6 → 1
0,6 x 2 = 1,2 → 1
0,2 x 2 = 0,4 → 0
0,4 x 2 = 0,8 → 0
…

Portanto, 0,8 = 0,110011001100110011001100….

Infinitas casas decimais. Porém, como na prática temos um número finito de bits, deve-se truncar o número para uma quantidade finita. Isso implica numa aproximação. Por exemplo, qual é o erro cometido ao se representar 0,8 como 0,11001100?

½ + ¼ + 1/32 + 1/64 = 51/64 = 0.796875, o que implica num erro de 0,003125.

O problema pode ser amplificado ao se realizar operações matemáticas com esse valor (é como se o erro fosse sendo propagado nos cálculos). Existem outros valores para que isso ocorre: 0,2, 0,4, 0,6

Ex: Forneça as representações computacionais na base 2 (binário) para os seguintes números. Quais são os erros cometidos se considerarmos apenas 8 bits para a parte fracionária?

a) 11,6
b) 27,4
c) 53,6
d) 31,2

Aula 2: Recorrência Logística (Logistic map)

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:

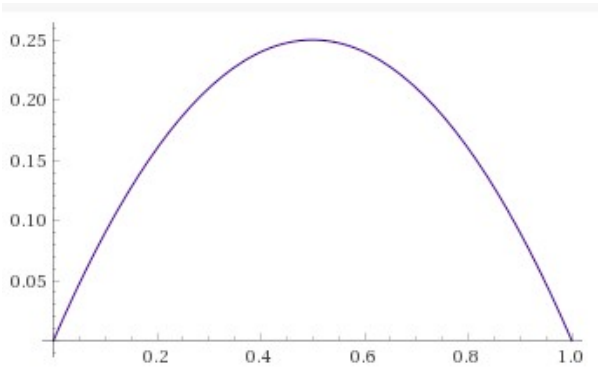
$$x_{n+1}=f(x_n)=-r\,x_n^2+r\,x_n$$

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:

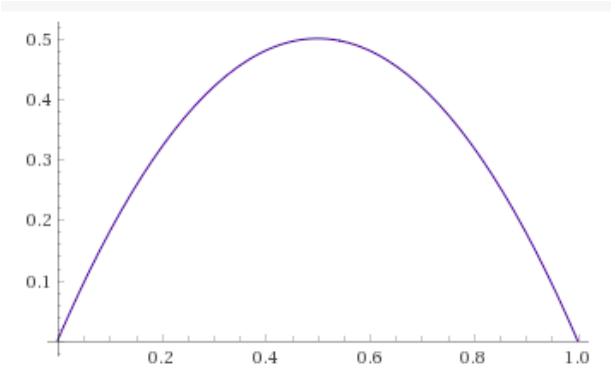
$$-2r\,x_n+r=0$$

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

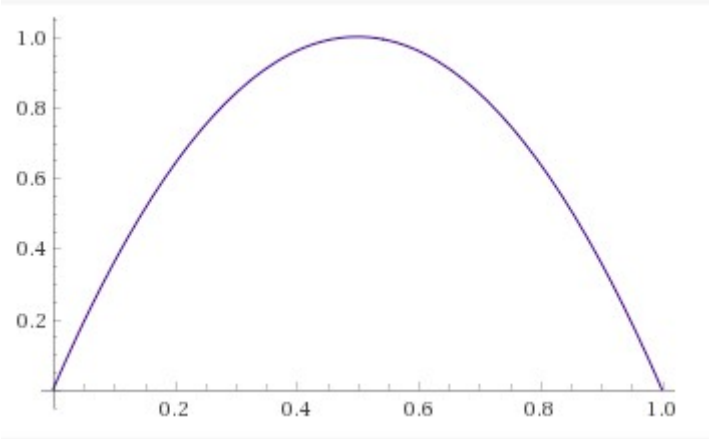
Note também que como c = 0, $f(0)=0$, ou seja, a parábola passa pela origem. Note ainda que $f(1)=0$, ou seja a parábola corta o eixo x no ponto x = 1. De forma gráfica temos para alguns valores de r as seguintes parábolas:



r = 1



r = 2



r = 4

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

```
using Plots

# Função que calcula a população ao longo de n iterações
function populacao(r, x, n)

     populacao = [x]

     for i in 2:n
          x = r*x*(1-x)
          push!(populacao, x)
     end

     println("População final: $(populacao[end])")

     eixo_x = collect(1:n)
     plot(eixo_x, populacao)

end
```

Lembrando que é necessário instalar o pacote Plots. Para isso, pressione backspace no REPL para entrar no modo de gerenciamento de pacotes e entre com o comando: `add Plots`

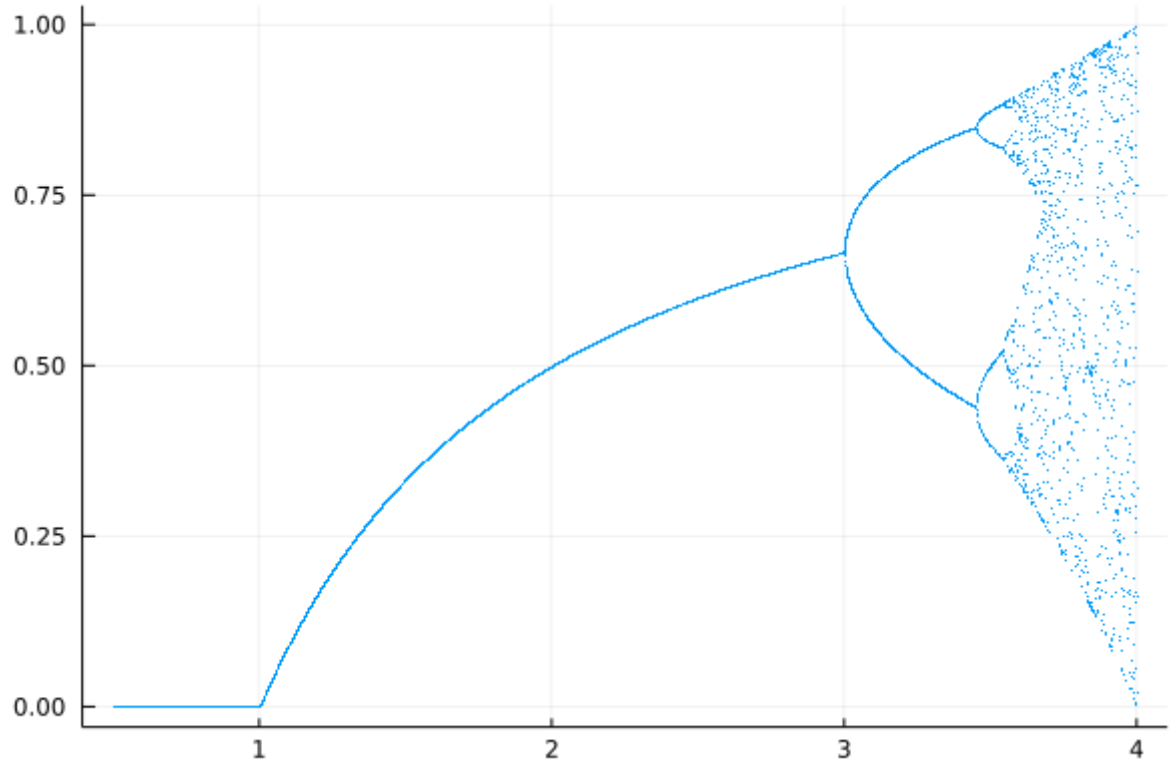
Execute a função no REPL e veja o que acontece para as entradas a seguir:

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. A função em Julia a seguir mostra a implementação computacional dessa análise.

```
# Função que cria bifurcation map
function bifurcation_map()
      # Cria 10000 pontos no intervalo de 0.5 a 4
      R = LinRange(0.5, 4, 10000)

      X = []
      Y = []

      for r in R
            push!(X, r)
            # Gera um número aleatório emtre 0 e 1
            x = rand()
            # Calcula população de equilíbrio para x
            for i in 1:1000
                  x = x*r*(1-x)
            end
            push!(Y, x)
      end

      scatter(X, Y, marker=1, legend=false)

end
```

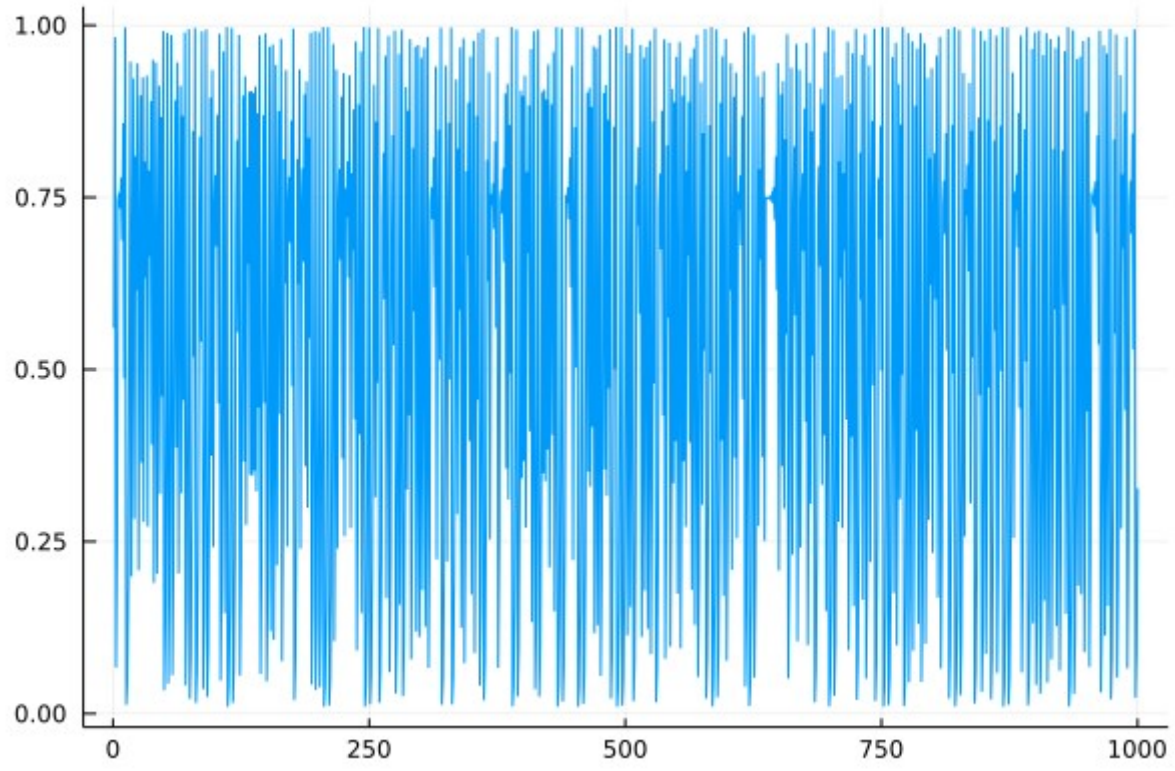
O gráfico plotado pelo script acima é conhecido como *bifurcation map*. Esse fenômeno da bifurcação ocorre como uma manifestação do comportamento caótico da população de equilíbrio para valores de r maiores que 3. Na prática, o que temos é que para um valor de r = 3.49999, a população de equilíbrio é 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 furação, conhecido também como o efeito borboleta.



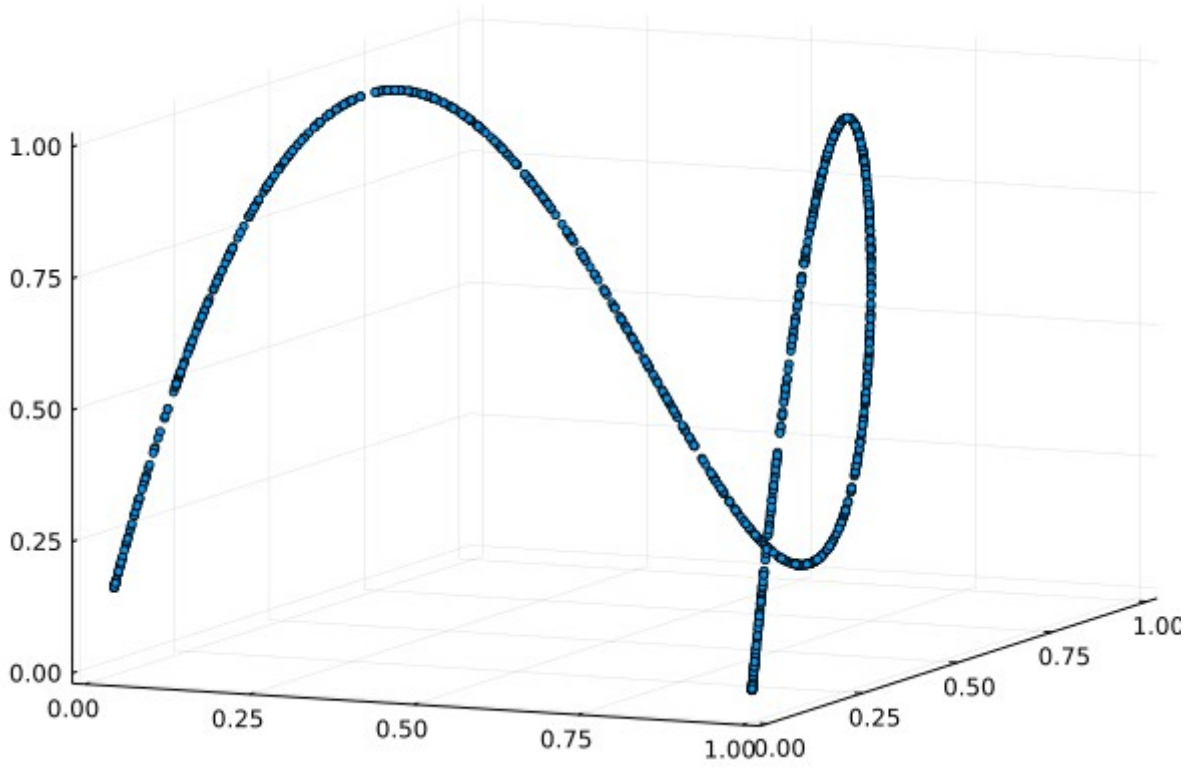
Uma das propriedades do caos é que é possível encontrar ordem e padrões em comportamentos caóticos. Por exemplo, a seguir iremos desenvolver um script em Python para plotar uma sequencia de populações, começando de uma população inicial arbitrária e utilizando o valor de r = 3.99.

```
# Função que constrói o atrator do modelo
function atrator(r)

      x = rand()
      X = [x]

      for i in 1:999
            x = r*x*(1-x)
            push!(X, x)
      end

      A = X[1:end-2]
      B = X[2:end-1]
      C = X[3:end]

      p1 = plot(X, legend=false, reuse=false)
      savefig("./Sequencia.png")

      p2 = scatter(A, B, C, marker=3, legend=false)
      savefig("./Atrator.png")

end
```

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

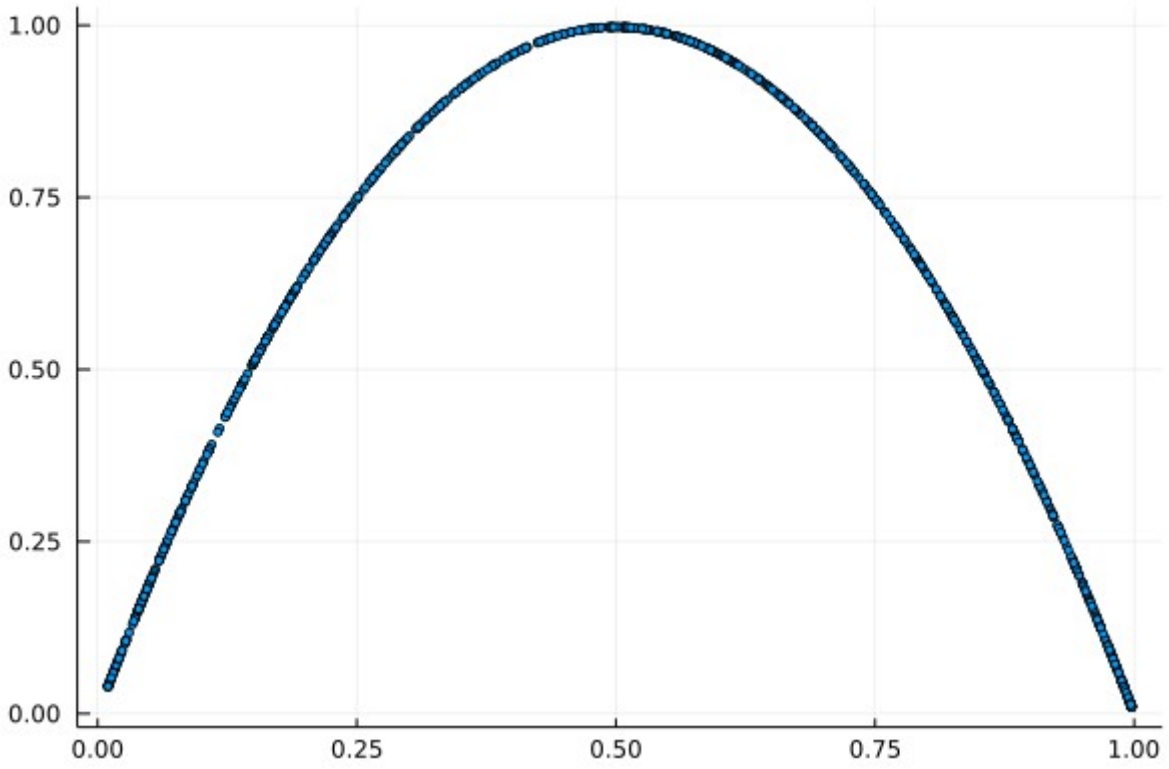


A seguir, iremos plotar cada subsequência (x_n, x_{n+1}, x_{n+2}) como um ponto no R^3. Na prática, isso significa que no eixo X iremos plotar a sequência original, no eixo Y iremos plotar a sequência deslocada de uma unidade e no eixo Z iremos plotar a sequência deslocada de duas unidades. Qual será 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).

```
# Função que constrói o atrator 2D do modelo
function atrator2D(r)

      x = rand()
      X = [x]

      for i in 1:999
            x = r*x*(1-x)
            push!(X, x)
      end

      A = X[1:end-1]
      B = X[2:end]

      p = scatter(A, B, marker=3, legend=false)
      savefig("./Atrator2D.png")

end
```



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.

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

Aula 3: Autômatos celulares e o game of life

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 disscreto 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

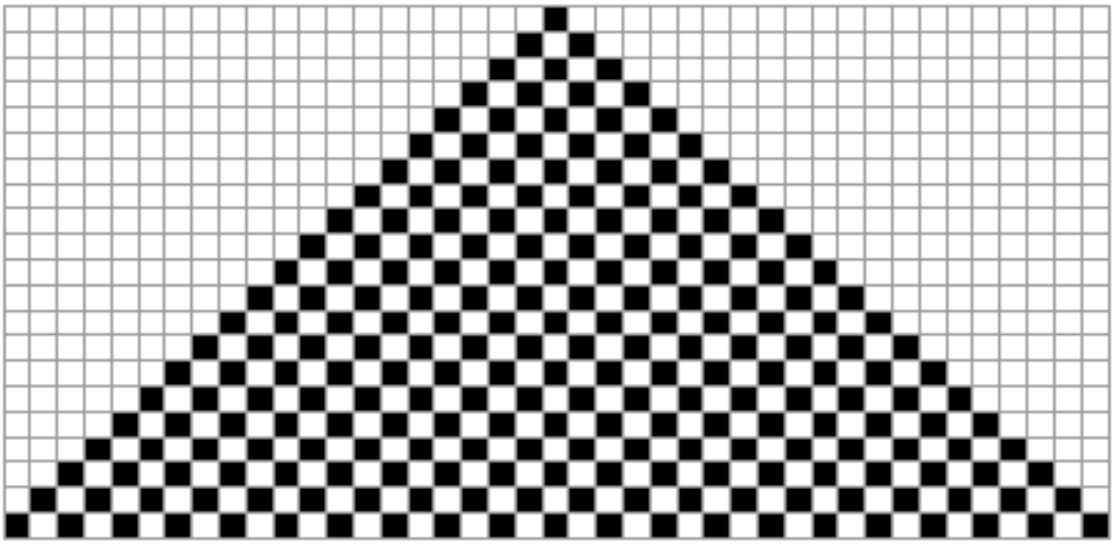
Def: Um autômatos celular é definido por uma 5-tupla de elementos

$A = (R, S, S_0, V, F)$ 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)

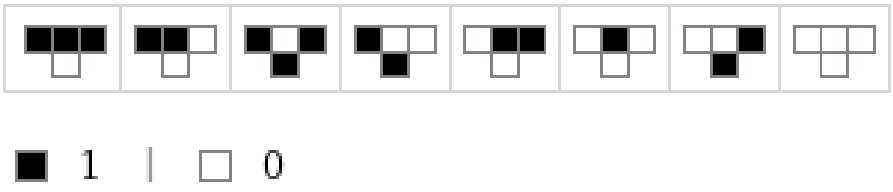


A questão é: como evoluir uma configuração de forma a construir esse padrão? Que regras são aplicadas para definir quais células vivem ou morrem na próxima geração?

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 sumarizado 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.

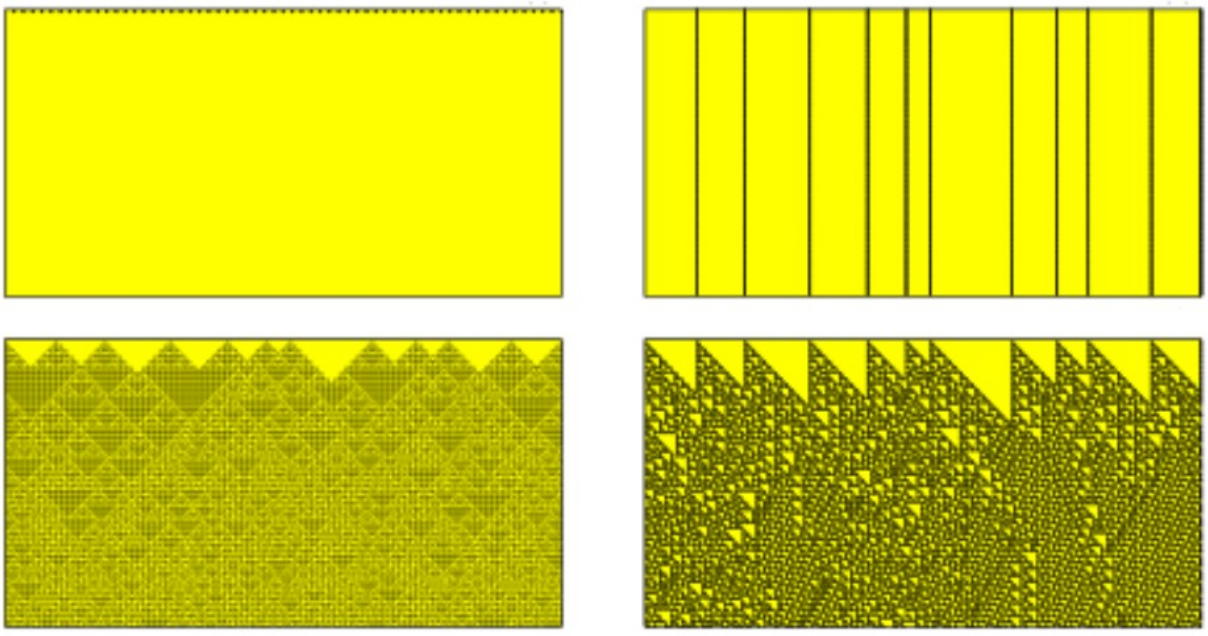
Note que não existem mais combinações possíveis de 0's e 1's usando apenas 3 bits, pois conseguimos contar em binário de 0 a 7, o que resulta em 8 possibilidades. Essa regra tem um nome: é a regra 50, pois o número binário correspondente a última coluna da função de transição vale 00110010, que em binário 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 = 500 até a geração 200.

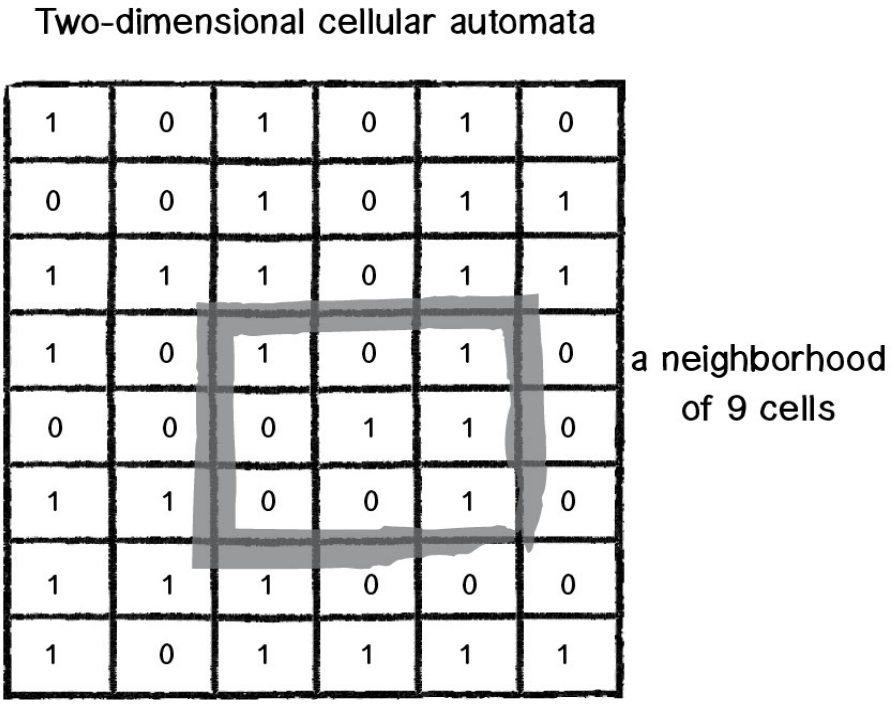
```
using Images, ImageView


# Função que converte um inteiro para binário
# Gera a tabela da regra que desejamos aplicar
function converte_binario(n)

     # Retorna um vetor com os bits de n
     L = parse.(Int8, split(bitstring(UInt8(n)), ""))

     return L

end
```

```
""" Função que aplica a regra R para simular autômato celular elementar
R é a regra: inteiro de 0 a 255
MAX é o número máximo de gerações (500, 1000, ...)
tamanho é o número de células por geração"""
function automato_celular_elementar(R, MAX, tamanho)
      # Inicializa a geração atual e a nova geração
      g = Int8.(zeros(tamanho))
      ng = Int8.(zeros(tamanho))
      # Gera o código para a regra R
      codigo = Int8.(converte_binario(R))
      println(codigo)
      # Matriz para armazenar toda a evolução
      matriz_evolucao = Int8.(zeros(MAX, tamanho))
      # Inicializa com uma única célula viva no centro
      g[div(tamanho, 2)] = 1

      # Loop principal
      for i in 1:MAX
            matriz_evolucao[i, :] = g

            # Percorre células da geração atual
            for j in 2:tamanho-1

                  # Arruma janela na primeira célula
                  if j == 1
                        previo = tamanho
                  else
                        previo = j-1
                  end
                  # Arruma janela na última célula
                  if j == tamanho
                        prox = (j+1)%tamanho
                  else
                        prox = j+1
                  end

                  if g[previo] == 0 && g[j] == 0 && g[prox] == 0
                        ng[j] = codigo[8]
                  elseif g[previo] == 0 && g[j] == 0 && g[prox] == 1
                        ng[j] = codigo[7]
                  elseif g[previo] == 0 && g[j] == 1 && g[prox] == 0
                        ng[j] = codigo[6]
                  elseif g[previo] == 0 && g[j] == 1 && g[prox] == 1
                        ng[j] = codigo[5]
                  elseif g[previo] == 1 && g[j] == 0 && g[prox] == 0
                        ng[j] = codigo[4]
                  elseif g[previo] == 1 && g[j] == 0 && g[prox] == 1
                        ng[j] = codigo[3]
                  elseif g[previo] == 1 && g[j] == 1 && g[prox] == 0
                        ng[j] = codigo[2]
                  elseif g[previo] == 1 && g[j] == 1 && g[prox] == 1
                        ng[j] = codigo[1]
                  end
            end

            g = copy(ng)

      end

      imshow(matriz_evolucao)

end
```

"The loneliest people are the kindest.
The saddest people smile the brightest.
The most damaged people are the wisest.
All because they do not wish to see anyone else suffer the way they do."
(Author unknown)

Conway’s game of life

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 de 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.



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 co 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 Julia que, dada uma configuração inicial, simule o jogo da Vida num tabuleiro de dimensões 100 x 100 por 200 gerações.

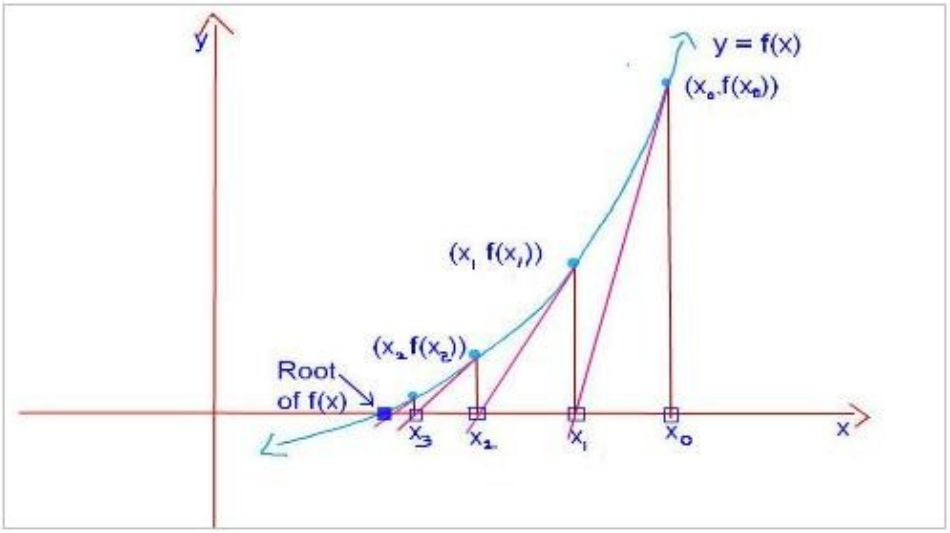
```
using Images

function inicializa_tabuleiro!(tabuleiro, padrao, i, j)

      dimensoes = size(padrao)
      linhas = dimensoes[1]
      colunas = dimensoes[2]

      tabuleiro[i:(i+linhas-1), j:(j+colunas-1)] = padrao

end

# Função que gera uma animação (GIF)
# Há dois modos de salvar animação
# grafico, como um gráfico de mapa de calor
# imagem, como uma imagem em tons de cinza
function animacao(matriz_evolucao, modo)

      if modo == "grafico"
            # Gerando animação (usa função heatmap do pacote Plots)
            # Salva como um gráfico de mapa de calor
            anim = @animate for i = 1:size(matriz_evolucao)[3]
            mat = matriz_evolucao[:,:,i]
            heatmap(mat, clim=(0,1), legend=false)
            end
            gif(anim, "animacao.gif", fps = 10)
      eles # modo == “imagem”
            # Salva como uma imagem em tons de cinza
            save("./animacao.gif", Gray.(matriz_evolucao); fps=10)
      end

end
```

```
"""Simula o Game of Life em uma grade n x n durante MAX gerações, em que
n a dimensão do tabuleiro
MAX é o número máximo de gerações (500, 1000, ...)
"""
function game_of_life(n, MAX)
     # Inicializa a geração atual e a nova geração
     g = Int8.(zeros(n, n))
     ng = Int8.(zeros(n, n))

     # Matriz para armazenar toda a evolução
     matriz_evolucao = Int8.(zeros(n, n, MAX))

     # Alguns padrões iniciais interessantes
     glider = [1 0 0; 0 1 1; 1 1 0]
     unbounded = [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]
     r_pentomino = [0 1 1; 1 1 0; 0 1 0]
     diehard = [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 = [0 1 0 0 0 0 0; 0 0 0 1 0 0 0; 1 1 0 0 1 1 1]

     # Inicializa a geração com um dos padrões acima
     meio = div(n, 2)
     inicializa_tabuleiro!(g, r_pentomino, meio, meio)

     # Loop principal
     for k in 1:MAX

          matriz_evolucao[:, :, k] = g

          # Percorre as células da geração atual (ignora as bordas)
          for i in 2:n-1
               for j in 2:n-1

                    vivos = sum(g[i-1:i+1, j-1:j+1]) - g[i, j]

                    if g[i, j] == 1
                         if vivos == 2 || vivos == 3
                              ng[i, j] = 1
                         else
                              ng[i, j] = 0
                         end
                    else
                         if vivos == 3
                              ng[i, j] = 1
                         end
                    end

               end
          end

          g = copy(ng)

     end

     return matriz_evolucao
end
```

"Fear has a large shadow, but he himself is small."
(Ruth Gendler)

Aula 4: 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 x0 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:

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

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:

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

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:

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

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) $f(x) = x - 2sen(x)$

```
using Plots

# Funções cujo zero será determinado
f1(x) = x^3 + x - 1
df1(x) = 3*x^2 + 1

f2(x) = x - cos(x)
df2(x) = 1 + sin(x)

f3(x) = exp(x) - 2*cos(x)
df3(x) = exp(x) + 2*sin(x)

# Essa função admite vários zeros (depende da inicialização)
f4(x) = x - 2*sin(x)
df4(x) = 1 - 2*cos(x)
```

```
# Função que implementa o método de Newton
# x: chute inicial e epson: tolerância do erro de aproximação
function newton(x, epson, f, df)

    while true

        x_n = x - f(x)/df(x)
        erro = abs(x_n - x)
        println("x = $x_n ********** Erro = $erro")
        x = x_n

        if erro < epson
            break
        end

    end

    # Plota gráfico da função
    eixo_x = LinRange(-2, 2, 200)
    eixo_y = f.(eixo_x)
    plot(eixo_x, eixo_y, legend=false)

end
```

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)

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

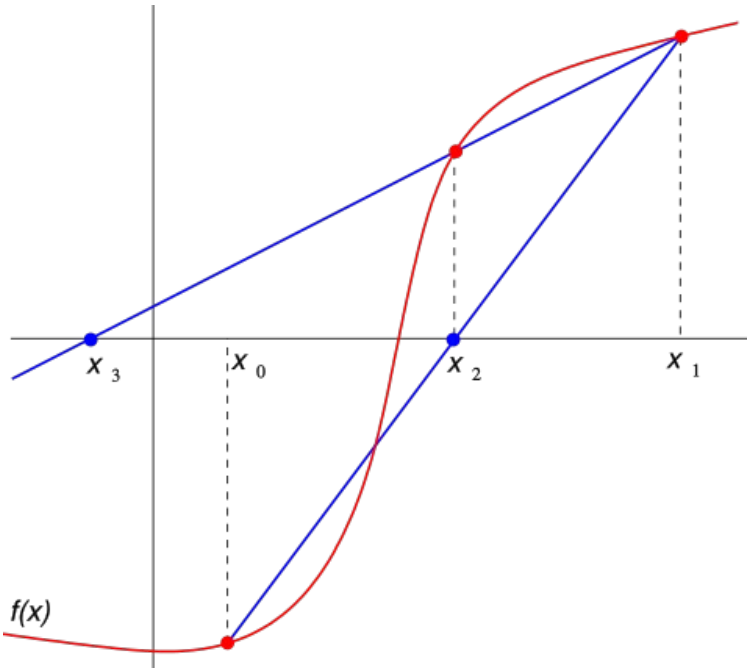
c) Iterações circulares (chute inicial gera loop infinito)

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

What is mathematics? It is only a systematic effort of solving puzzles posed by nature.
— Shakuntala Devi

Aula 5: O 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:

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

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:

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

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 $n \to \infty$. 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:

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

cuja solução positiva é dada 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.

```
using Plots

# Funções cujo zero será determinado
f1(x) = x^3 + x - 1

f2(x) = x - cos(x)

f3(x) = exp(x) - 2*cos(x)

# Essa função admite vários zeros (depende da inicialização)
f4(x) = x - 2*sin(x)
```

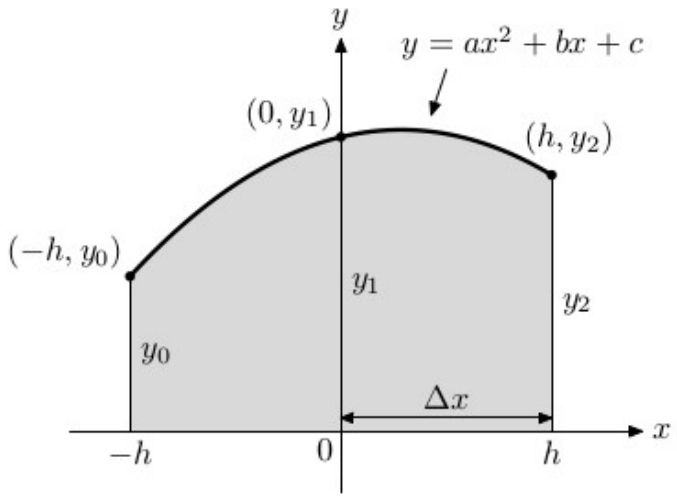
```
# Função que implementa o método da secante
# x0, x1 chutes iniciais e epson é a tolerância do erro de aproximação
# Não precisa do cálculo anÍlítico da derivada da função
function secante(x0, x1, epson, f)

    while true

        x_n = x1 - f(x1)*(x1 - x0)/(f(x1) - f(x0))
        erro = abs(x0 - x1)
        println("x = $x_n ********** Erro = $erro")
        x0, x1 = x1, x_n

        if erro < epson
            break
        end
    end

    # Plota gráfico da função
    eixo_x = LinRange(-2, 2, 200)
    eixo_y = f.(eixo_x)
    plot(eixo_x, eixo_y, legend=false)

end
```

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.

Mathematics is the music of reason.
— James Joseph Sylvester

Aula 6: O método 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:

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

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:

$$\begin{aligned} y_0 &= ah^2 - bh + c \\ y_1 &= c \\ y_2 &= ah^2 + bh + c \end{aligned}$$

Note porém que a seguinte igualdade é válida:

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

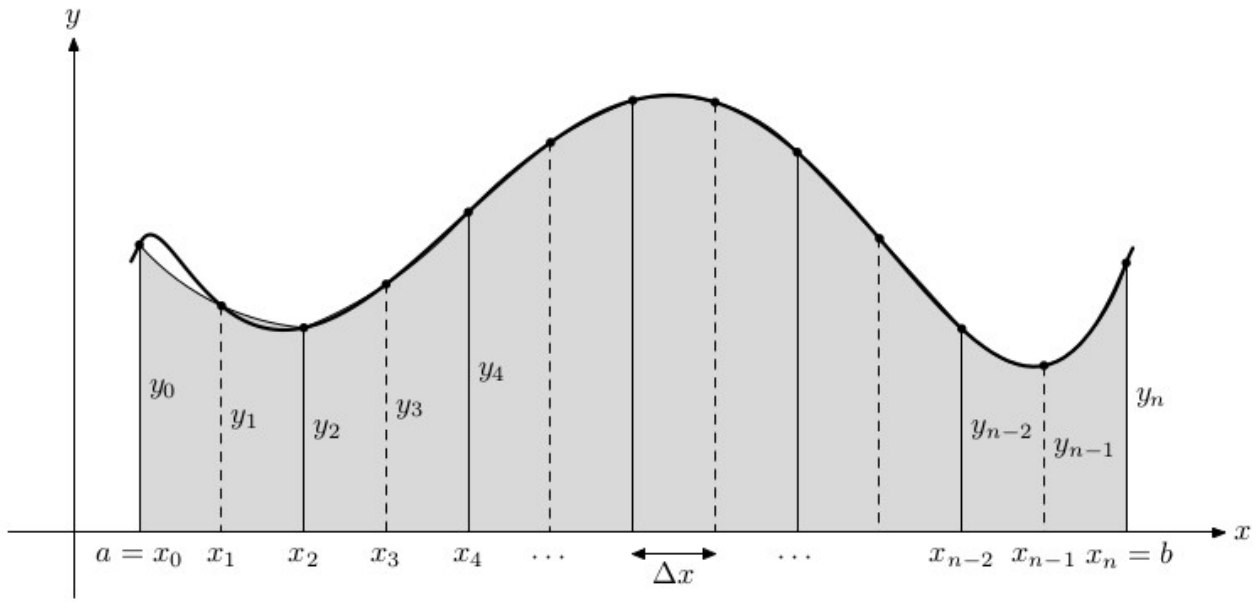
Assim, a área sob a parábola pode ser reescrita como:

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

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):

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

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



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

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

Simplificando a expressão anterior, chega-se a:

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

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:

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

Veremos a seguir uma função em Julia que implementa a regra de Simpson.

```
using Plots

# Funções cujo zero será determinado
f1(x) = sqrt(1 + x^3)

f2(x) = 1/sqrt(1 + x^4)

f3(x) = (1/sqrt(2*π))*exp(-0.5*x^2)     # Normal(0, 1)

# Função que implementa a regra de Simpson
function simpson(f, a, b, n)

    h = (b - a)/n
    soma_pares, soma_impares = 0, 0

    for i in 2:2:n
        k = a + i*h
        soma_pares += f(k)
    end

    for i in 1:2:n
        k = a + i*h
        soma_impares += f(k)
    end
```

```
    area = (h/3)*(f(a) + 4*soma_impares + 2*soma_pares + f(b))
    println("Área sob a curva = $area")

    # Plota gráfico da função
    eixo_x = LinRange(0, 2, 200)
    eixo_y = f.(eixo_x)
    plot(eixo_x, eixo_y, legend=false)

end
```

Nature is written in mathematical language.
— Galileo Galilei

Aula 7: Sistemas lineares – o método da eliminação de Gauss

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:

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

onde $x_1, x_2, \ldots, x_n$ são variáveis, $a_{11}, a_{12}, a_{13,} \ldots, a_{mn}$ são coeficientes do sistema (conhecidos) e $b_1, b_2, \ldots, b_m$ 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 $x_1, x_2, \ldots, x_n$ 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:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 &= b_1 \qquad (L_1) \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 &= b_2 \qquad (L_2) \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 &= b_3 \qquad (L_3) \end{aligned}$$

A matriz aumentada do sistema é:

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

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

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

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

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

obtém-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

$$x_3 = b_3^{(2)} / a_{33}^{(2)} \ , \ a_{33}^{(2)} \neq 0$$
$$x_2 = (b_2^{(2)} - (a_{23}^{(2)} x_3)) / a_{22}^{(2)}$$
$$x_1 = (b_1^{(2)} - (a_{12}^{(2)} x_2) - a_{13}^{(2)} x_3) / a_{11}^{(2)}$$

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, $a_{11}^{(1)}, a_{22}^{(2)}, a_{33}^{(3)}, \ldots, a_{nn}^{(n)} \neq 0$

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

```
# Função que define e imprime sistema na tela
function imprime_sistema()
    # Define sistema linear
    A = [10.0 -1.0 2.0 0.0;
        -1.0 11.0 -1.0 3.0;
         2.0 -1.0 10.0 -1.0;
         0.0 3.0 -1.0 8.0]
    b = [6.0, 25.0, -11.0, 15.0]

    if size(A)[1] != size(A)[2]
        println("Sistema inválido...")
        return -1
    end

    n = length(b)
    for i in 1:n
        linha = []
        for j in 1:n
            push!(linha, "$(A[i, j])*x$j")
        end
        println(join(linha, " + "), " = ", b[i])
    end

    return (A, b)

end

# Função que implementa o método da eliminação de Gauss
function eliminacao_gauss(A, b)
    n = length(b)
    x = zeros(n)

    # Primeira etapa: transformar sistema na forma triangular
    for i in 1:n-1
        for j in i+1:n
            m = A[j, i]/A[i, i]
            for k in i:n
                A[j, k] = A[j, k] - m*A[i, k]
            end
            b[j] = b[j] - m*b[i]
        end
    end

    # Segunda etapa: resolver sistema triangular de equações
    # Percorre colunas de trás para frente
    for j in n:-1:1
        x[j] = b[j]/A[j, j]
        # Percorre coluna de baixo para cima
        for i in j-1:-1:1
            b[i] = b[i] - A[i, j]*x[j]
        end
    end

    return x

end
```

Aula 8: Sistemas lineares – métodos iterativos

Existem métodos analíticos para resolver um sistema linear obtendo suaa 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 $n\,x\,n$, $\vec{x}$ 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

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

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

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

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 Julia.

```
# Função que define e imprime sistema na tela
function imprime_sistema()
    # Define sistema linear
    A = [10.0 -1.0 2.0 0.0; -1.0 11.0 -1.0 3.0; 2.0 -1.0 10.0 -1.0; 0.0
           3.0 -1.0 8.0]
    b = [6.0, 25.0, -11.0, 15.0]

    if size(A)[1] != size(A)[2]
        println("Sistema inválido...")
        return -1
    end

    n = length(b)
    for i in 1:n
        linha = []
        for j in 1:n
            push!(linha, "$(A[i, j])*x$j")
        end
        println(join(linha, " + "), " = ", b[i])
    end

    return (A, b)
end
```

```
# Função que implementa o método de Jacobi
function jacobi(A, b, MAX)

    n = length(b)
    x = zeros(n)

    for i in 1:MAX
        println("Solução atual = ", x)
        D = Diagonal(A)
        R = A - D
        x_n = inv(D)*(b - R*x)
        if norm(x_n - x) < 10^(-8)
            break
        end
        x = x_n
    end

    return x

end
```

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:

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

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.

```
# Função que implementa o método de Gauss-Seidel
function gauss_seidel(A, b, MAX)

    n = length(b)
    x = zeros(n)

    for i in 1:MAX
        println("Solução atual = ", x)
        L = tril(A)
        U = A - L
        x_n = inv(L)*(b - U*x)
        if norm(x_n - x) < 10^(-8)
            break
        end
        x = x_n
    end

    return x

end
```

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:

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

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

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

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

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

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 $\lim_{k\to\infty} H^k = 0$.

Decomposição em autovalores e autovetores (Eigendecomposition)

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

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

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

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

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

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

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

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

o limite acima é satisfeito somente se $|\lambda_i| < 1, i = 1,2,...n$. 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

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

$|\lambda_1|=1.23321\times10^{-5}$
$|\lambda_2|=1.23321\times10^{-5}$ Ok, iteração converge!
$|\lambda_3|=1.23322\times10^{-5}$

b) no método de Gauss-Seidel

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

$|\lambda_1|=0$
$|\lambda_2|=2$ (falhou)
$|\lambda_3|=2$ (falhou) - não haverá convergência

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 uma função em Julia que gere matrizes de Hilbert H de ordem n x 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.

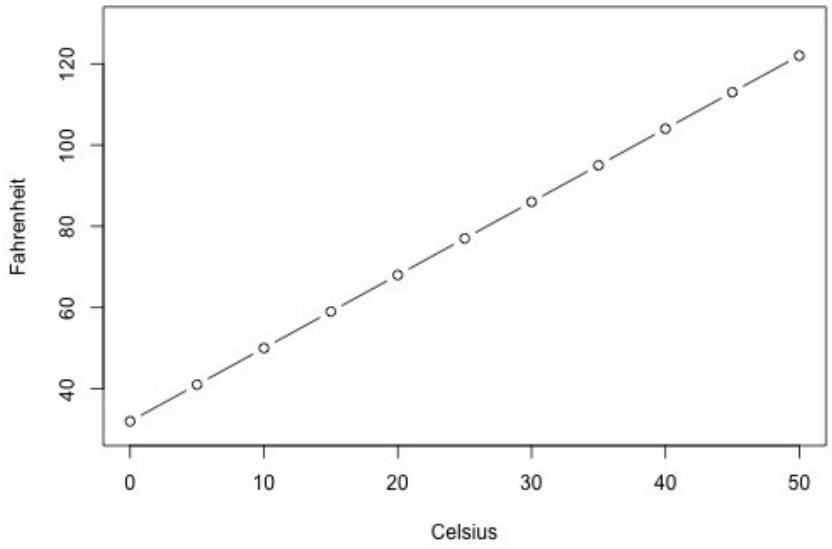
If people do not believe that mathematics is simple, it is only because they do not realize how complicated life is.
— John von Neumann, American-hungarian mathematician

Aula 9: 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.



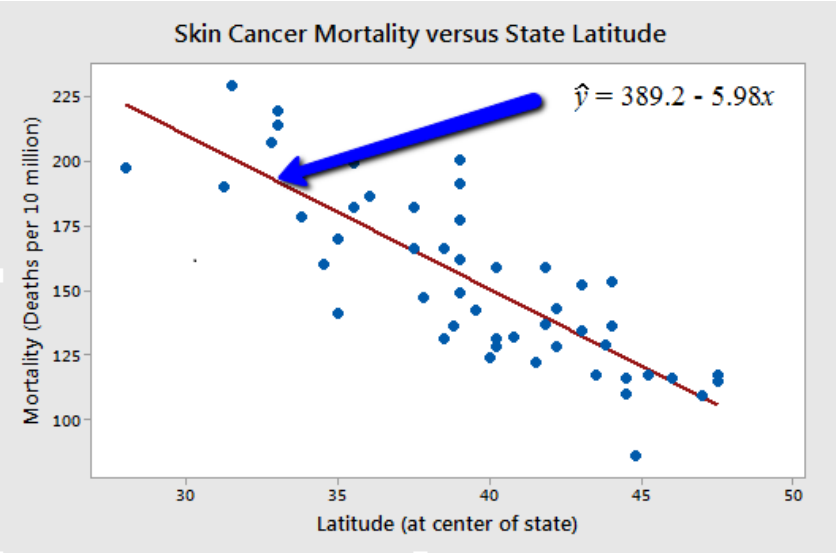
Note que os pontos (x, y) observados caem diretamente sobre uma linha reta. Isso ocorre porque o relacionamento entre graus Celsius e Farenheit é dada por:

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

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 varáveis: 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 exibe 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 $i=1,..,n$, 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 predições 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:

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

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:

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

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:

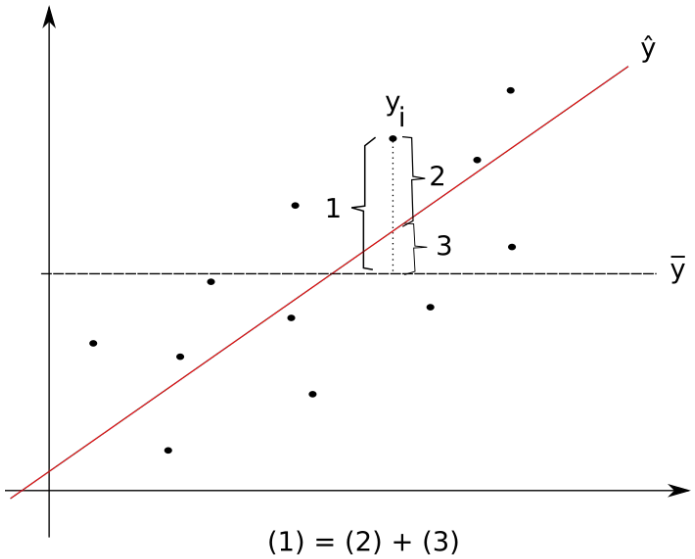
$$\beta = \frac{\overline{xy} - \bar{x}\,\bar{y}}{\overline{x^2} - \bar{x}^2} \qquad \text{(inclinação da reta)}$$

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.



(1) = (2) + (3)

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

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:

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

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

$ESS = \sum_i (\hat{y}_i - \bar{y})^2$ (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:

$$\hat{y}_i = \alpha + \beta x_i$$
$$\bar{y} = \alpha + \beta \bar{x}$$

Então,

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

Também sabemos que:

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

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

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

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

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

Agrupando em um somatório único:

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

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

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

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

$$\sum_{i=1}^{n} x_i^2 - \overline{x}\sum_{i=1}^{n} x_i - \overline{x}\sum_{i=1}^{n} x_i + n\,\overline{x}^2 = \sum_{i=1}^{n} x_i^2 - \overline{x}\sum_{i=1}^{n} x_i - \overline{x}\sum_{i=1}^{n} x_i + \sum_{i}^{n} \overline{x}^2$$

Agrupando em um somatório único:

$$\sum_{i=1}^{n} \left(x_i^2 - \overline{x}\,x_i - x_i \overline{x} + \overline{x}^2 \right) = \sum_{i=1}^{n} \left[x_i (x_i - \overline{x}) - \overline{x}(x_i - \overline{x}) \right] = \sum_{i=1}^{n} (x_i - \overline{x})(x_i - \overline{x}) = \sum_{i=1}^{n} (x_i - \overline{x})^2$$

Portanto, o valor de β é dado por:

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

Voltando a equação (*), temos:

$$\sum_{i} (y_i - \hat{y}_i)(\hat{y}_i - \overline{y}) = \beta \sum_{i} (x_i - \overline{x})(y_i - \hat{y}_i)$$

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

$$\beta \sum_{i} (x_i - \overline{x})[\, y_i - \overline{y} - \beta(x_i - \overline{x})] = \beta \left[\sum_{i} (x_i - \overline{x})(y_i - \overline{y}) - \beta \sum_{i} (x_i - \overline{x})^2 \right]$$

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

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

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

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

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

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

Observações:

1. Partindo de:

$$\sum_i (x_i-\overline{x})(y_i-\overline{y})=\sum_i x_i y_i-\overline{y}\sum_i x_i-\overline{x}\sum_i y_i+n\overline{x}\,\overline{y}$$

e dividindo numerador e denominador por n, chega-se a:

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

2. Partindo de:

$$\sum_i (x_i-\overline{x})(x_i-\overline{x})=\sum_i x_i^2-\overline{x}\sum_i x_i-\overline{x}\sum_i x_i+n\overline{x}^2$$

e dividindo numerador e denominador por n, chega-se a:

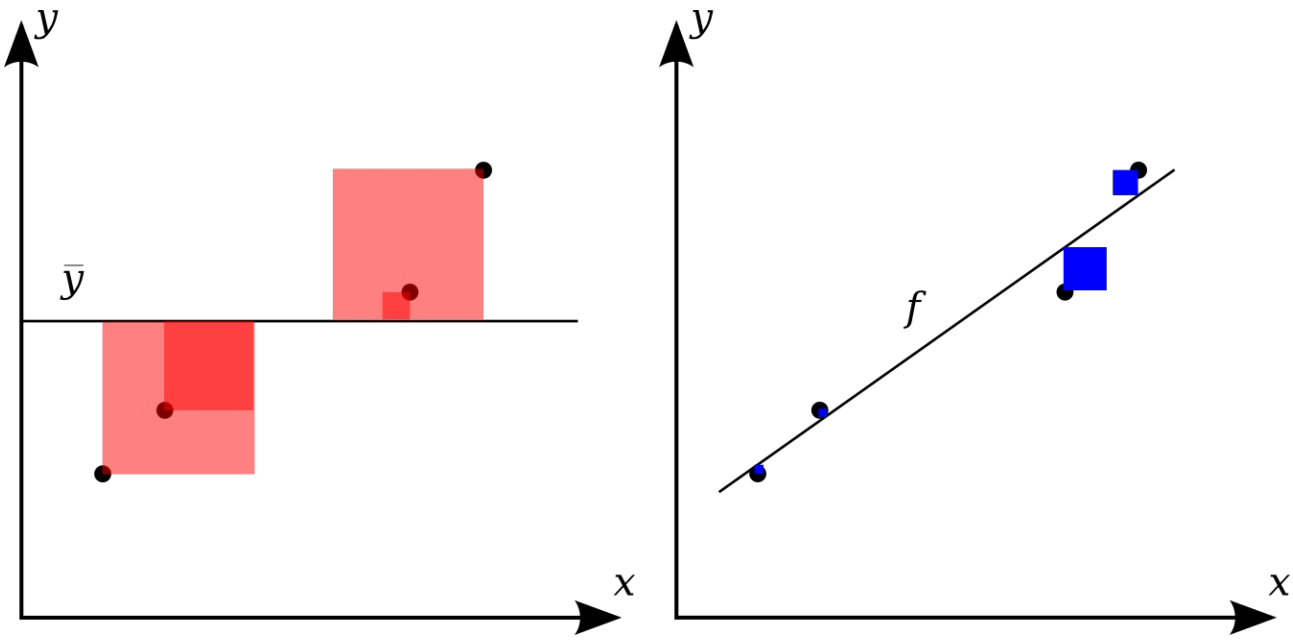
$$\overline{x^2}-\overline{x}^2-\overline{x}^2+\overline{x}^2=\overline{x^2}-\overline{x}^2$$

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

Portanto, chegamos em:

$$r^2=\frac{[\overline{xy}-\overline{x}\,\overline{y}]^2}{(\overline{x^2}-\overline{x}^2)(\overline{y^2}-\overline{y}^2)}$$ (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.



$r^2 = 1 - \frac{RSS}{TSS}$ (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. Com um auxílio de um computador, responda aos questionamentos a seguir:

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

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

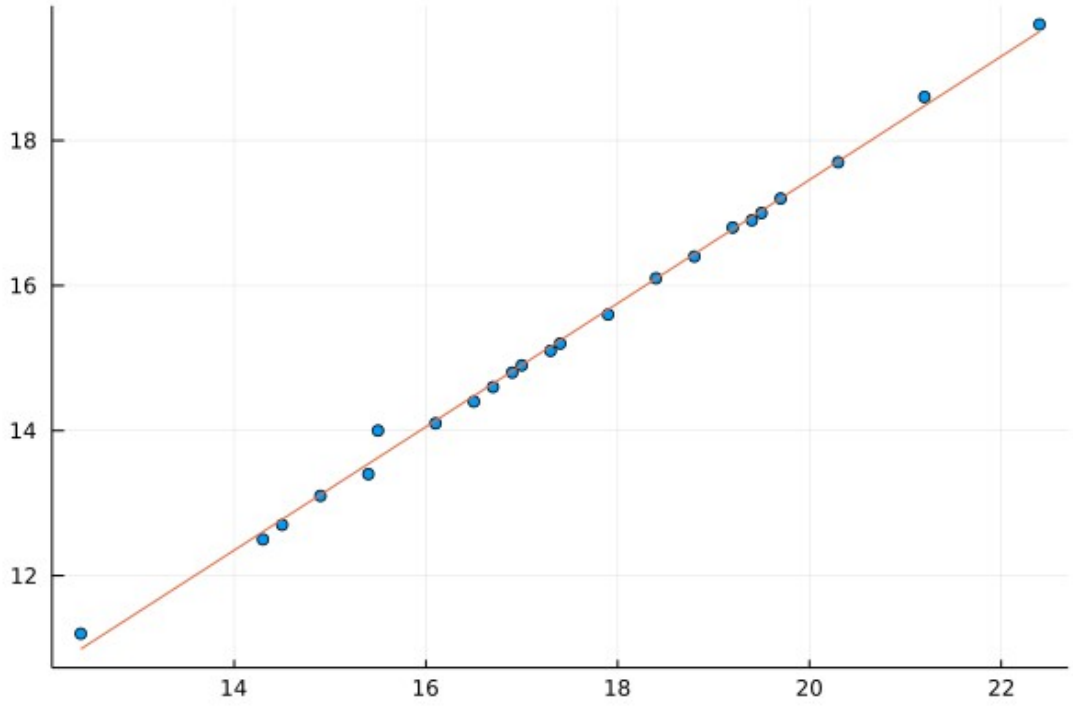
c) Compute o coeficiente de correlação de Pearson r

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

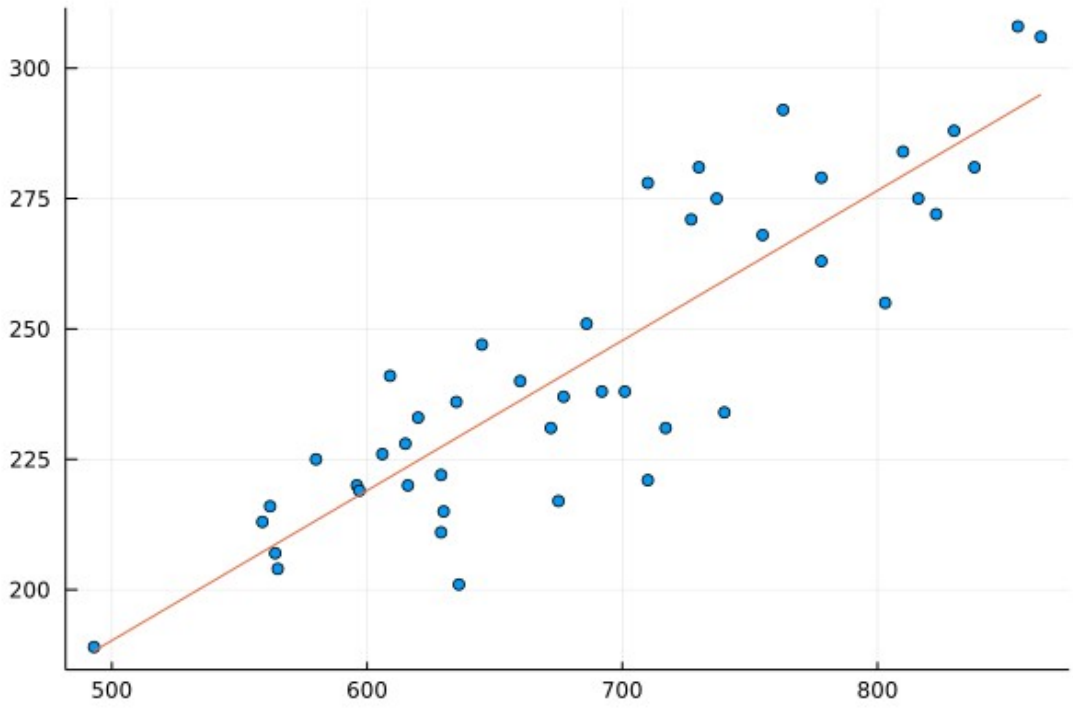
Veremos a seguir uma função em Julia que implementa a regressão linear simples.

```
using Statistics
using DelimitedFiles
using Plots

# Função que realiza a leitura de dados a partir de um arquivo texto
# Pode ser necessário modificar essa função em caso de outros conjuntos
de dados
function leitura_dados(arquivoX, arquivoY)

    X = readdlm(arquivoX, '\n', '\r')
    Y = readdlm(arquivoY, '\n', '\r')

    return (X, Y)

end

# Função que implementa a regressão linear simples
function regressao_linear()

    # Trocar os nomes dos arquivos para usar outros conjuntos de dados
    X, Y = leitura_dados("./slr01/slr01l1.txt", "./slr01/slr01l2.txt")
    n = length(X)

    # Calcula as médias
    X_bar = mean(X)
    Y_bar = mean(Y)

    # Calcula as médias de X ao quadrado
    X2_bar = (1/n)*(sum(X.*X))
    Y2_bar = (1/n)*(sum(Y.*Y))
    XY_bar = (1/n)*(sum(X.*Y))

    # Estima a inclinação da reta de mínimos quadrados
    beta = (XY_bar - X_bar*Y_bar)/(X2_bar - X_bar^2)

    # Estima o intercepto
    alpha = Y_bar - beta*X_bar

    # Estima coeficiente de correlação de Pearson
    r = (XY_bar-X_bar*Y_bar)/sqrt((X2_bar-X_bar^2)*(Y2_bar-Y_bar^2))

    println("Reta de mínimos quadrados: Y = $beta X + $alpha")
    println("R² = $(r^2)")

    # Plota gráfico
    scatter(X, Y, legend=false)

    eixo_x = LinRange(minimum(X), maximum(X), 100)
    eixo_y = beta*eixo_x .+ alpha
    plot!(eixo_x, eixo_y)

end
```

Algumas das plotagens obtidas pela função acima encontram-se a seguir.



$R^2 = 0.997$



$R^2 = 0.776$

Life is a math equation. In order to gain the most, you have to know how to convert negatives into positives.
— Anonymous

Aula 10: Regressão quadrática

Da mesma forma que podemos ajustar uma reta de mínimos quadrados em um conjunto de pontos no plano, é possível generalizar essa ideia para uma parábola: trata-se da regressão quadrática. Lembrando que uma parábola é definida pela equação quadrática:

$$y = ax^2 + bx + c \quad \text{com} \quad a \neq 0$$

Sendo assim, desejamos minimizar o erro quadrático médio:

$$Q(a,b,c) = \sum_{i=1}^{n} \epsilon_i^2 = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 = \sum_{i=1}^{n} (y_i - a x_i^2 - b x_i - c)^2$$

Derivando em relação a cada um dos parâmetros e igualando a zero, temos o seguinte sistema de equações:

$$\begin{bmatrix} \sum x_i^4 & \sum x_i^3 & \sum x_i^2 \\ \sum x_i^3 & \sum x_i^2 & \sum x_i \\ \sum x_i^2 & \sum x_i & n \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} \sum x_i^2 y_i \\ \sum x_i y_i \\ \sum y_i \end{bmatrix}$$

O coeficiente de determinação R^2 pode ser calculado por:

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

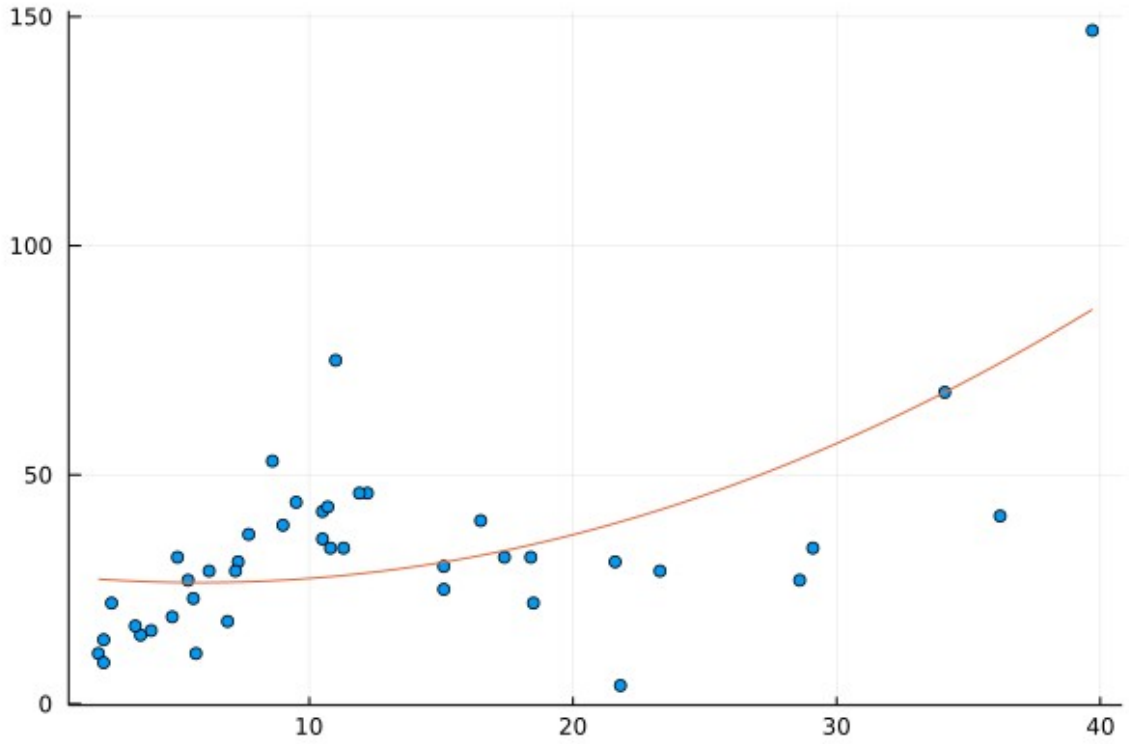
onde $RSS = \sum_{i=1}^{n} (y_i - a x_i^2 - b x_i - c)^2$ e $TSS = \sum_{i=1}^{n} (y_i - \overline{y})^2$

A seguir apresentamos uma função em Julia para realizar a regressão quadrática.

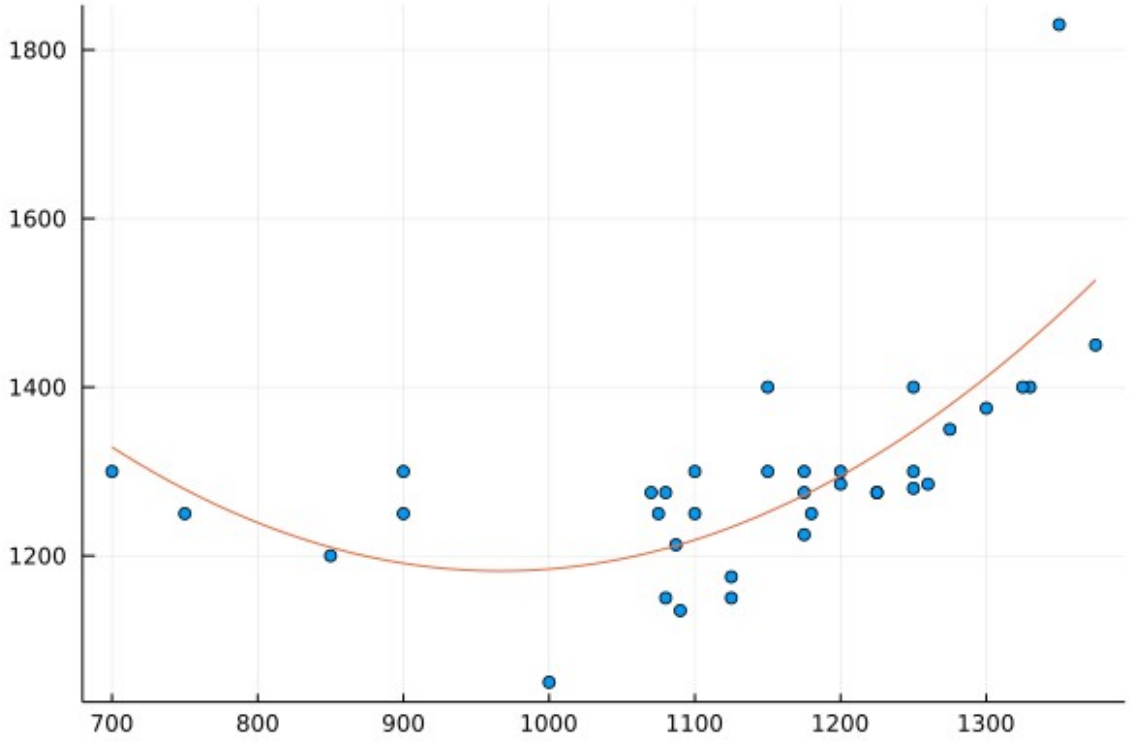
```
using Statistics
using DelimitedFiles
using Plots

# Função que realiza a leitura de dados a partir de um arquivo
function leitura_dados(arquivoX, arquivoY)

    X = readdlm(arquivoX, '\n', '\r')
    Y = readdlm(arquivoY, '\n', '\r')

    return (X, Y)

end
```

```
# Função que implementa a regressão linear simples
function regressao_quadratica()

    # Trocar os nomes dos arquivos para usar outros datasets
    X, Y = leitura_dados("./slr05/slr05l1.txt", "./slr05/slr05l2.txt")
    n = length(X)

    # Calcula as médias
    X_bar = mean(X)
    Y_bar = mean(Y)
    X2_bar = (1/n)*(sum(X.^2))
    X3_bar = (1/n)*(sum(X.^3))
    X4_bar = (1/n)*(sum(X.^4))

    # Calcula as médias cruzadas
    XY_bar = (1/n)*(sum(X.*Y))
    X2Y_bar = (1/n)*(sum((X.^2).*Y))

    # Monta matriz dos dados
    A = [X4_bar X3_bar X2_bar; X3_bar X2_bar X_bar; X2_bar X_bar 1]
    # Cria vetor de dados
    b = [X2Y_bar, XY_bar, Y_bar]

    # Computa vetor de parâmetros
    beta = A\b

    # Estima coeficiente de determinação
    Y_est = beta[1]*X.^2 .+ beta[2]*X .+ beta[3]

    RSS = sum((Y - Y_est).^2)
    TSS = sum((Y .- Y_bar).^2)
    R2 = 1 - RSS/TSS

    println("Parábola: Y = $(beta[1]) X² + $(beta[2]) X + $(beta[3])")
    println("R² = $(R2)")

    # Plota gráfico
    scatter(X, Y, legend=false)
    eixo_x = LinRange(minimum(X), maximum(X), 100)
    eixo_y = beta[1]*eixo_x.^2 .+ beta[2]*eixo_x .+ beta[3]

    plot!(eixo_x, eixo_y)

    savefig("Regressao_Quadratica.png")

end
```

Algumas das plotagens obtidas pela função acima encontram-se a seguir.



$R^2 = 0.358$



$R^2 = 0.529$

Just because we can’t find a solution, it doesn’t mean there isn’t one.
— Andrew Wiles, English mathematician

Aula 11: Regressão polinomial

Lembre-se que na regressão quadrática temos:

$$y_i = \beta_0 + \beta_1 x + \beta_2 x^2 + \epsilon$$

É possível generalizar o modelo de ordem 2 para uma ordem arbitrária n. Nesse caso, temos uma regressão polinomial. O modelo geral da regressão polinomial de ordem m é definido por:

$$y_i = \beta_0 + \beta_1 x + \beta_2 x^2 + \beta_3 x^3 + ... + \beta_m x^m + \epsilon$$

para $i = 1, 2, ..., n$. Esse modelo pode ser expresso de forma mais compacta através de uma notação matricial:

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} 1 & x_1 & x_1^2 & \dots & x_1^m \\ 1 & x_2 & x_2^2 & \dots & x_2^m \\ 1 & x_3 & x_3^2 & \dots & x_3^m \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \dots & x_n^m \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \vdots \\ \beta_m \end{bmatrix} + \begin{bmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \varepsilon_3 \\ \vdots \\ \varepsilon_n \end{bmatrix},$$

o que pode ser reescrito como:

$$\vec{y} = X\vec{\beta} + \vec{\epsilon}$$

Da mesma forma que na regressão linear, o vetor de parâmetros do modelo pode ser estimado por mínimos quadrados. Em termos matemáticos, desejamos encontrar o vetor de parâmetros β que minimiza a soma dos resíduos ao quadrado, ou seja:

$$J(\vec{\beta}) = \sum_{i=1}^{n} \varepsilon_i^2 = \vec{\varepsilon}^T \vec{\varepsilon} = (\vec{y} - \hat{\vec{y}})^T (\vec{y} - \hat{\vec{y}}) = (\vec{y} - X\vec{\beta})^T (\vec{y} - X\vec{\beta})$$

Assim, a condição para minimizar o funcional $J(\vec{\beta})$ é dada por:

$$\frac{\partial J(\vec{\beta})}{\partial \beta} = X^T (\vec{y} - X\vec{\beta}) = 0$$

Resolvendo a equação temos:

$$X^T \vec{y} - X^T X \vec{\beta} = 0$$

$$(X^T X)\vec{\beta} = X^T \vec{y}$$
$$\hat{\vec{\beta}} = (X^T X)^{-1} X^T \vec{y} = X^+ \vec{y}$$

com $X^+ = (X^T X)^{-1} X^T$ sendo a matriz pseudo-inversa de X (*generalized inverse*).

Da mesma forma que nos casos anteriores, o coeficiente de determinação R^2 pode ser calculado por:

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

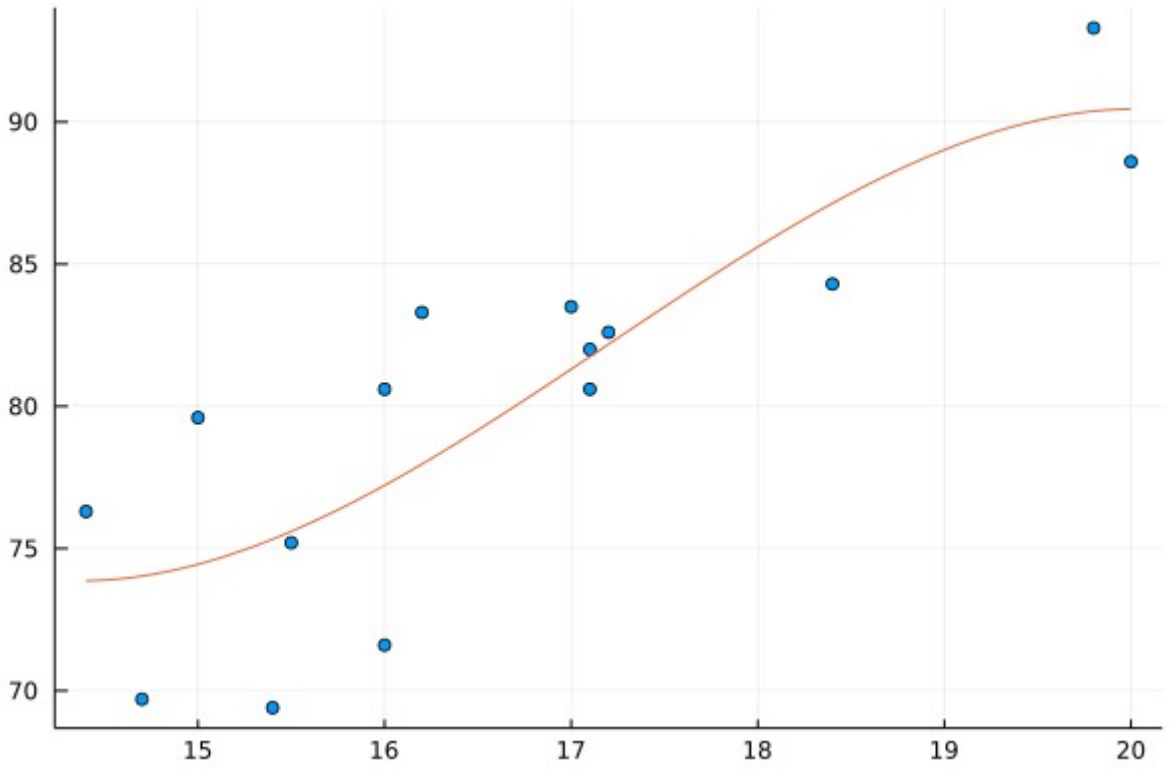
onde $RSS = \sum_{i=1}^{n} (y_i - \hat{y})^2$ e $TSS = \sum_{i=1}^{n} (y_i - \overline{y})^2$

A seguir apresentamos uma função em Julia para realizar a regressão polinomial de ordem 3.

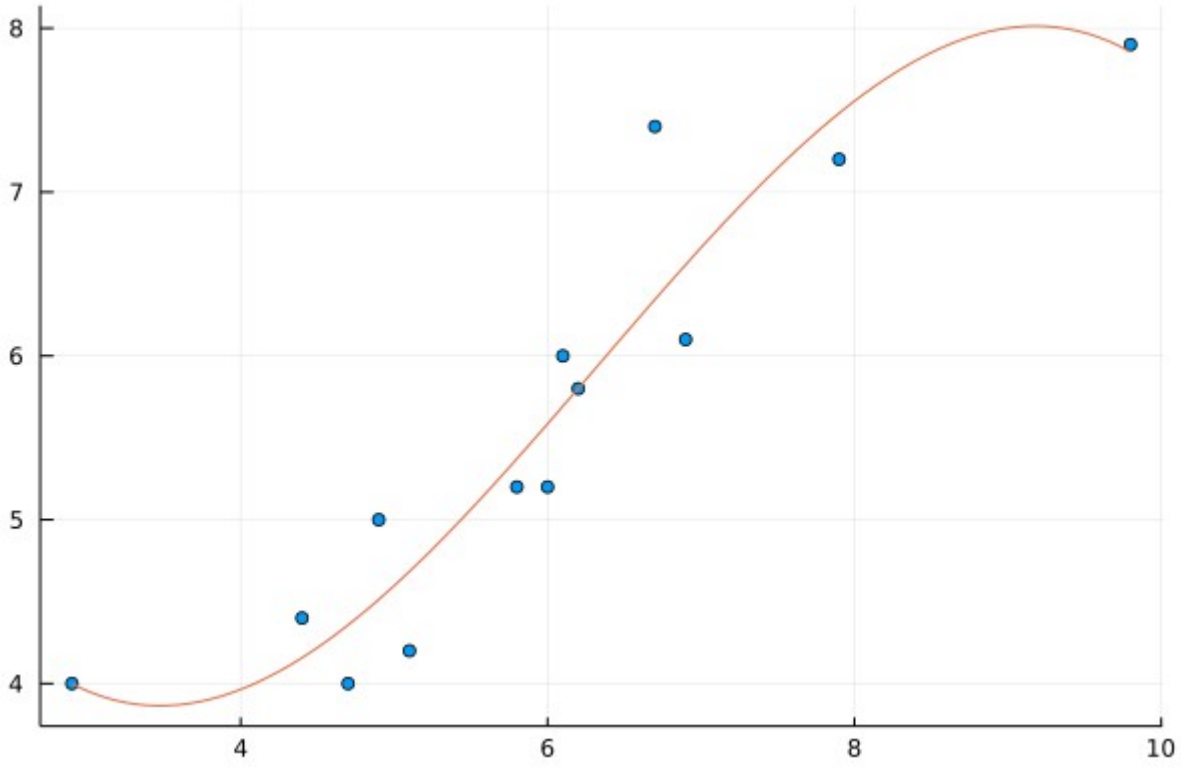
```
using Statistics
using DelimitedFiles
using Plots
using LinearAlgebra

# Função que realiza a leitura de dados a partir de um arquivo texto
function leitura_dados(arquivoX, arquivoY)
    X = readdlm(arquivoX, '\n', '\r')
    Y = readdlm(arquivoY, '\n', '\r')

    return (X, Y)

end

# Função que implementa a regressão linear simples
function regressao_polinomial()

    # Trocar os nomes dos arquivos para usar outros conjuntos de dados
    X, Y = leitura_dados("./slr02/slr02l1.txt", "./slr02/slr02l2.txt")
    n = length(X)

    # Monta matriz de dados
    dados = hcat(ones(n), X, X.^2, X.^3)

    # Estima vetor de parâmetros
    beta = pinv(dados)*Y

    # Calcula as estimstivas
    Y_est = dados*beta

    # # Calcula a média
    Y_bar = mean(Y)

    # Estima coeficiente de determinação
    RSS = sum((Y - Y_est).^2)
    TSS = sum((Y .- Y_bar).^2)
    R2 = 1 - RSS/TSS

    println("Parâmetros do modelo: ", beta)
    println("R² = $(R2)")
```

```
    # Plota gráfico
    scatter(X, Y, legend=false)
    eixo_x=LinRange(minimum(X), maximum(X), 100)
    eixo_y=beta[4]*eixo_x.^3.+beta[3]*eixo_x.^2.+beta[2]*eixo_x.+beta[1]

    plot!(eixo_x, eixo_y)

    savefig("Regressao_Polinomial.png")

end
```

Algumas das plotagens obtidas pela função acima encontram-se a seguir.



$R^2 = 0.707$



$R^2 = 0.887$

Mathematics is a place where you can do things which you can’t do in the real world.
— Marcus du Sautoy

Aula 12: Interpolação polinomial

Muitas vezes desejamos estimar o valor de uma função $y=f(x)$ em um determinado ponto x baseado em alguns valores conhecidos da função $f(x_0), f(x_1), \ldots, f(x_n)$ em um conjunto de n+1 pontos pré-determinados $a=x_0 \leq x_1 \leq \ldots \leq x_n=b$. Esse processo é chamado de interpolação se o ponto de interesse x pertence ao intervalo [a, b]. Uma maneira de realizar a interpolação é aproximar a função desconhecida por um polinômio de grau n. Neste caso, dizemos que a interpolação é polinomial:

$$f(x) \approx P_n(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_2 x^2 + a_1 x + a_0 = \sum_{j=1}^{n} a_j x^j$$

onde os n+1 coeficientes a_j podem ser obtidos a partir dos n+1 pontos conhecidos. Uma vez que o polinômio $P_n(x)$ é definido, qualquer operação matemática aplicada a função, como diferenciação, integração, cálculos de zeros e pontos de ótimo (máximos e mínimos) pode ser realizada de maneira aproximada em $P_n(x)$.

Em termos práticos, isso significa que temos n+1 equações que devem ser satisfeitas simultaneamente:

$$P_n(x_i) = \sum_{j=0}^{a_j} x_i^j = f(x_i) = y_i \qquad \text{para i = 0, 1, …, n}$$

Sendo assim, os coeficientes $a_0, a_1, \ldots, a_n$ podem ser encontrados resolvendo o seguinte sistema de equações

$$\begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ 1 & x_2 & x_2^2 & \cdots & x_2^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^n \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} = \mathbf{Va} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \mathbf{y}$$

onde V é uma matriz de Vandermonde. Através da resolução desse sistema linear, temos os coeficientes do nosso polinômio de grau n que podemos utilizar para computar o valor da função em qualquer ponto x intermediário de [a, b]. Note que aqui, as funções $x^0, x^1, x^2, \ldots, x^n$ são as funções de base polinomiais que são utilizadas para expressar o universo de todos os polinômios de grau menor ou igual a n. Pode-se mostrar que se os n+1 pontos $x_0, x_1, \ldots, x_n$ são distintos, a matriz V tem posto completo, e logo sua inversa existe e é bem definida, de modo que a solução do sistema linear acima é única, e portanto, o polinômio $P_n(x)$ também será único.

A seguir apresentamos uma função em Julia que implementa a interpolação polinomial utilizando a técnica descrita anteriormente.

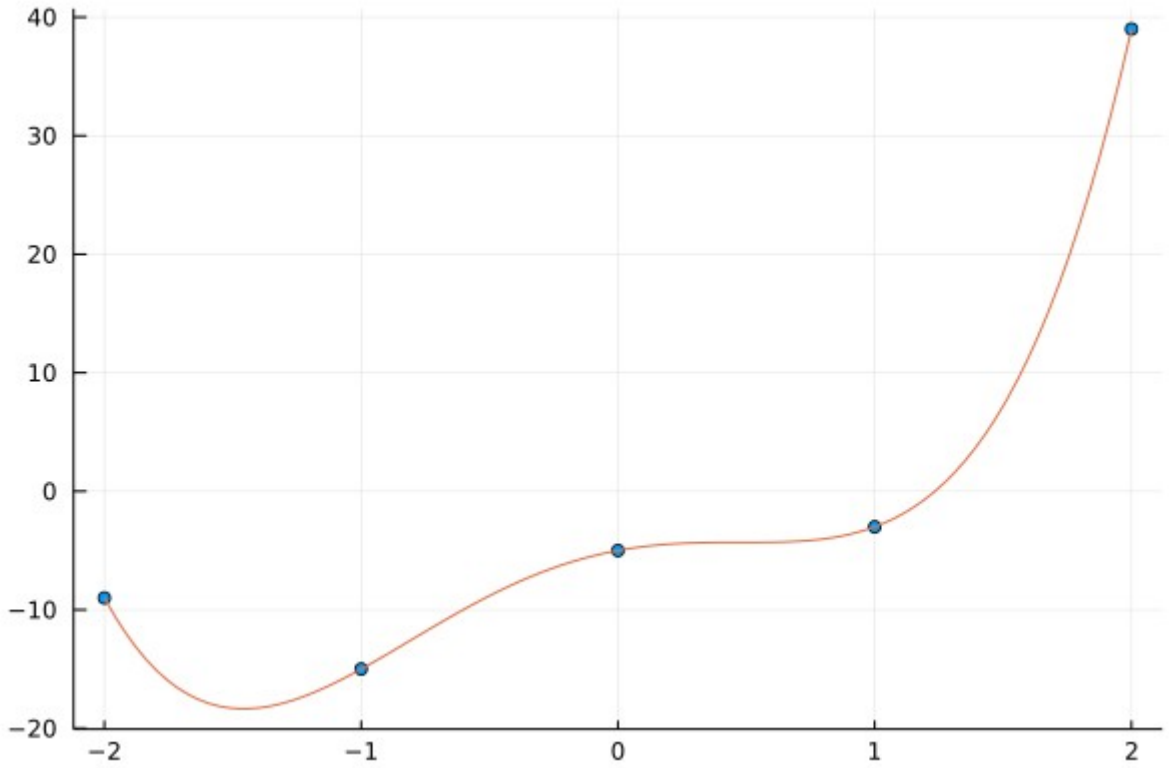
```
# Função que define e imprime sistema na tela
function define_pontos()

    # Define os pontos x e os valores de y = f(x)
    x = [-2.0, -1.0, 0.0, 1.0, 2.0]
    y = [-9.0, -15.0, -5.0, -3.0, 39.0]

    return (x, y)

end

# Função que implementa o método da matriz de Vandermonde
function interpolacao_polinomial(x, y)

    if length(x) != length(y)
        println("Dados inválidos...")
        return -1
    else
        n = length(x)
        V = hcat(ones(n), x, x.^2, x.^3, x.^4)
        # Resolve sistema linear
        a = V\y
        # Imprime coeficientes do polinômio
        println("Coeficientes do polinômio: ", a)
        # Cria eixo x com 100 pontos entre -2 e 2
        eixo_x = LinRange(minimum(x), maximum(x), 100)
        eixo_y = a[1] .+ a[2]*eixo_x .+ a[3]*eixo_x.^2 .+
                  a[4]*eixo_x.^3 + a[5]*eixo_x.^4
        scatter(x, y, legend=false)
        plot!(eixo_x, eixo_y)
    end

end
```

Em termos práticos esse método no entanto possui algumas limitações conhecidas:

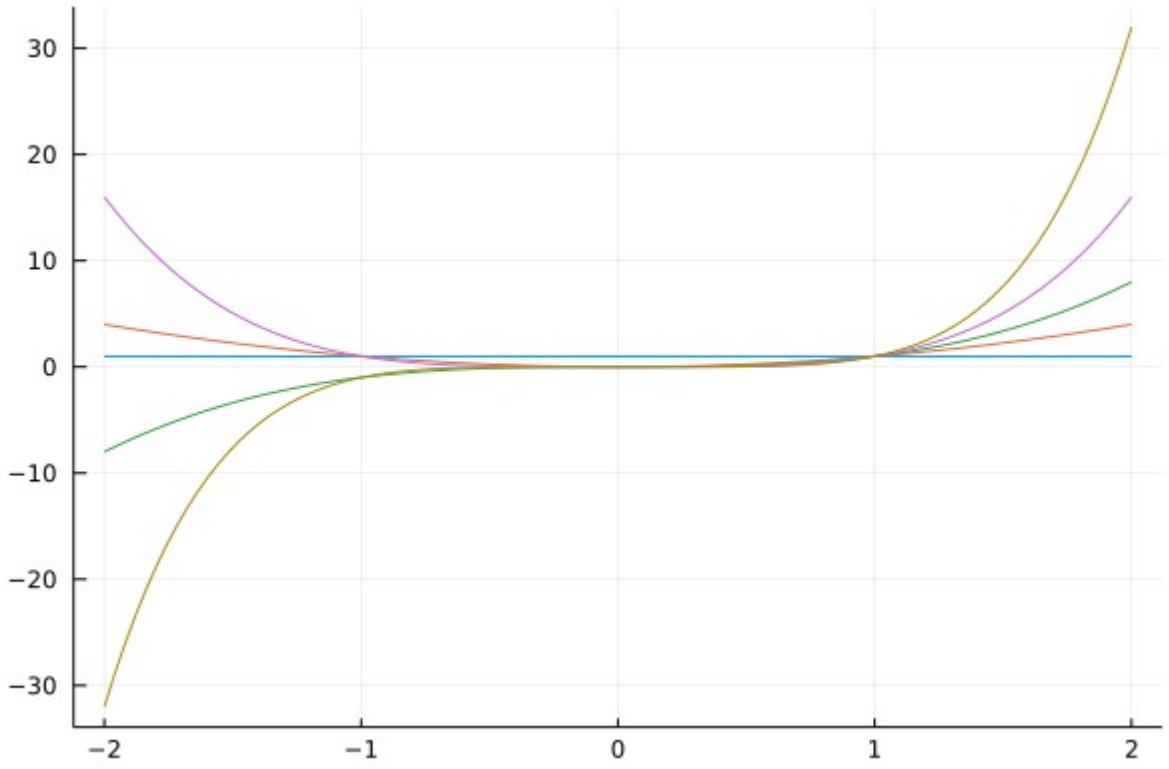
i) O elevado custo computacional para encontrar a matriz inversa de V é da ordem de $O(n^3)$ e pode ser tornar inviável se o número de pontos cresce muito.

ii) Pode-se mostrar que a matriz V torna-se mal condicionada quando n cresce de maneira arbitraria. Isso significa que pequenas perturbações nos pontos e nos valores da função nesses pontos levam a polinômios totalmente diferentes.

Por essa razão veremos duas alternativas para o método anterior. Antes, a figura a seguir ilustra o resultado obtido pela função interpolacao_polinomial(x, y) mostrada acima.



A figura a seguir plota as funções de base (monômios em x) de ordem 1 até ordem 5, ou seja, $x, x^2, \ldots, x^5$. Um dos problemas com os monômios é que todos eles são praticamente constantes ao redor do zero. Por essa razão podemos ter problemas em reconstruir funções nessas regiões.



Funções de Lagrange

O esquema utilizado até esse momento utiliza como funções de base os monômios $x^0, x^1, \ldots, x^n$. O método a seguir baseia-se em trocar as funções de base: ao invés de utilizar monômios, utiliza-se os polinômios de Lagrange como funções de base, Dados n pontos distintos, o i-ésimo polinômio de Lagrange, denotado por $L_i(x)$, é dado por:

$$L_i(x) = \prod_{j=1, j\neq i}^{n} \frac{x - x_j}{x_i - x_j} \quad (*)$$

Uma característica fundamental destes polinômios é que para um conjunto de n pontos, todos possuem grau n-1 e satisfazem:

$$L_i(x_k) = \begin{cases} 1, se\, k = i \\ 0, se\, k \neq i \end{cases}$$

o que faz com que o sistema linear associado as n equações torne-se:

$$\begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ \vdots & & & \vdots & \\ 0 & 0 & 0 & \cdots & 1 \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_n \end{pmatrix}$$

ou seja, o polinômio interpolador é dado por:

$P_{n-1}(x) = \sum_{i=1}^{n} f_i L_i(x)$ (combinação linear das funções de Lagrange ponderadas pelos valores da função)

onde f_i denota o valor da função f no ponto x_i . Veremos a segui um exemplo prático:

Sejam os pontos a seguir:

x_i	-2	-1	0	1	2
$y_i = f(x_i)$	-9	-15	-5	-3	39

Desejamos encontrar o polinômio interpolador com funções de Lagrange. Para isso, devemos computar as funções de base $L_i(x)$ para i = 1, 2, …, n. De acordo com a equação (*), temos:

$$L_1(x) = \frac{(x+1)x(x-1)(x-2)}{(-2-(-1))(-2-0)(-2-1)(-2-2)} = \frac{x(x+1)(x-1)(x-2)}{24}$$

Note que a ideia é excluir o termo i do numerador (por isso (x + 2) não entra!) e do denominador (senão teríamos o fator (-2 - (-2)) = 0 no denominador, o que tornaria a fração indeterminada. Seguindo a mesma lógica, as demais funções de Lagrange para esse caso são dadas por:

$$L_2(x)=\frac{(x+2)x(x-1)(x-2)}{(-1-(-2))(-1-0)(-1-1)(-1-2)}=\frac{(x+2)x(x-1)(x-2)}{-6}$$

$$L_3(x)=\frac{(x+2)(x+1)(x-1)(x-2)}{(0-(-2))(0-(-1))(0-1)(0-2)}=\frac{(x-1)(x-2)(x+1)(x+2)}{4}$$

$$L_4(x)=\frac{(x+2)(x+1)x(x-2)}{(1-(-2))(1-(-1))(1-0)(1-2)}=\frac{(x+1)(x+2)x(x-2)}{-6}$$

$$L_5(x)=\frac{(x+2)(x+1)x(x-1)}{(2-(-2))(2-(-1))(2-0)(2-1)}=\frac{(x+1)(x+2)x(x-1)}{24}$$

Portanto, o polinômio interpolador é dado por:

$$P_{n-1}(x)=-9\frac{(x-2)(x-1)x(x+1)}{24}-15\frac{(x-2)(x-1)x(x+2)}{-6}-5\frac{(x-1)(x-2)(x+1)(x+2)}{4}$$
$$-3\frac{(x-2)x(x+1)(x+2)}{-6}+39\frac{(x-1)x(x+1)(x+2)}{24}$$

o que após simplificações se torna:

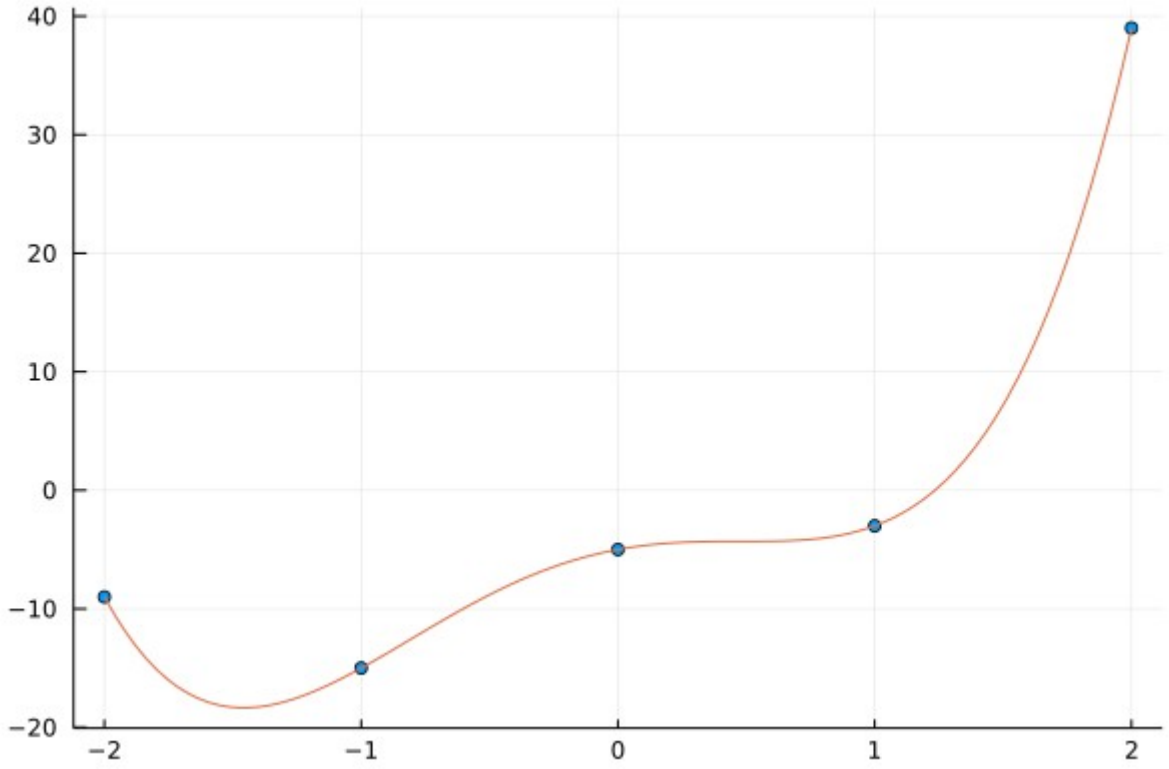
$$P_{n-1}(x)=-3\frac{(x-2)(x-1)x(x+1)}{8}+5\frac{(x-2)(x-1)x(x+2)}{2}-5\frac{(x-1)(x-2)(x+1)(x+2)}{4}$$
$$+\frac{(x-2)x(x+1)(x+2)}{2}+13\frac{(x-1)x(x+1)(x+2)}{8}$$

A seguir apresentamos uma função em Julia para plotar o gráfico do polinômio interpolador.

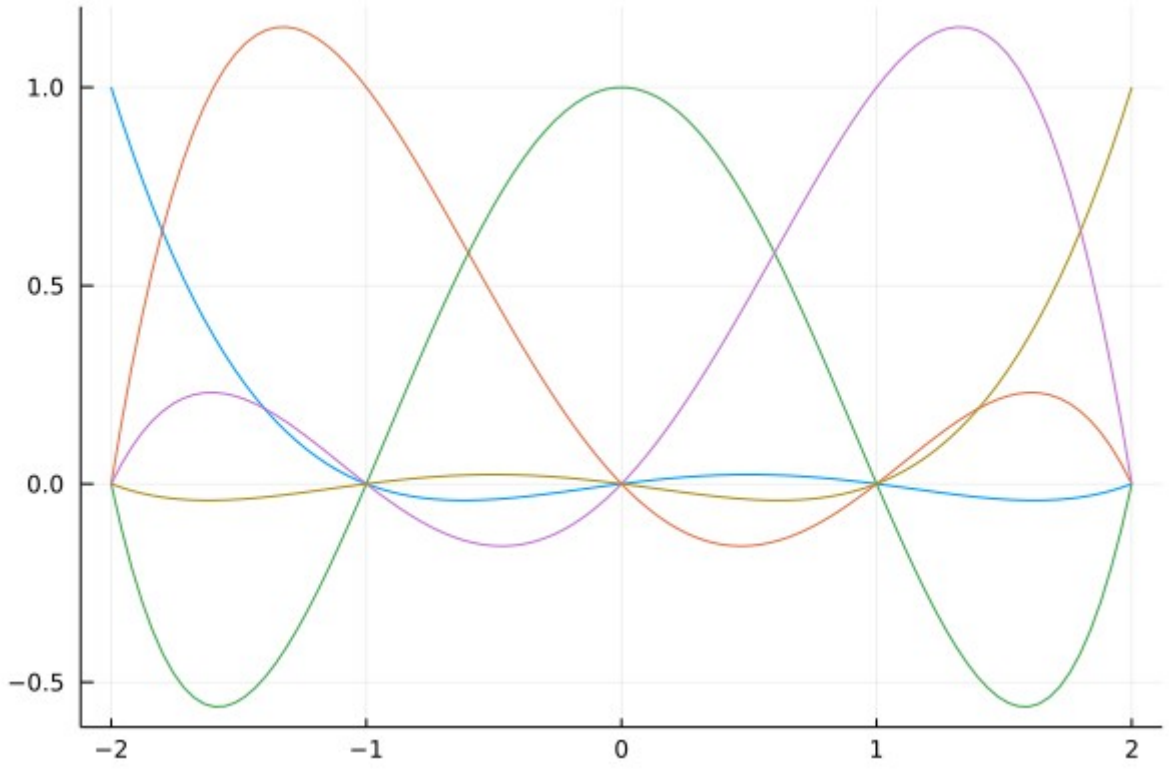
```
# Função que define e imprime sistema na tela
function define_pontos()

    # Define os pontos x e os valores de y = f(x)
    x = [-2.0, -1.0, 0.0, 1.0, 2.0]
    y = [-9.0, -15.0, -5.0, -3.0, 39.0]

    return (x, y)

end
```

```
# Função que plota o gráfico do polinômio (Lagrange)
function interpolacao_lagrange(x, y)

    if length(x) != length(y)
        println("Dados inválidos...")
        return -1
    else
        n = length(x)
        eixo_x = LinRange(minimum(x), maximum(x), 100)

        # Define polinômios de Lagrange
        L1=((eixo_x.-2).*(eixo_x.-1).*eixo_x.*(eixo_x.+1))/24
        L2=((eixo_x.-2).*(eixo_x.-1).*eixo_x.*(eixo_x.+2))/(-6)
        L3=((eixo_x.-2).*(eixo_x.-1).*(eixo_x.+1).*(eixo_x.+2))/4
        L4=((eixo_x.-2).*eixo_x.*(eixo_x.+1).*(eixo_x.+2))/(-6)
        L5=((eixo_x.-1).*eixo_x.*(eixo_x.+1).*(eixo_x.+2))/24

        eixo_y = y[1]*L1 + y[2]*L2 + y[3]*L3 + y[4]*L4 + y[5]*L5
        scatter(x, y, legend=false)
        plot!(eixo_x, eixo_y)
        savefig("Lagrange.png")
    end

end
```

Como era de se esperar, como estamos lidando com um número pequeno de pontos, o resultado da interpolação com funções de Lagrange deve ser muito próximo do método da matriz de Vandermonde. O gráfico a seguir ilustra o polinômio interpolador obtido.



A figura a seguir ilustra as funções de Lagrange, utilizadas como funções de base na reconstrução do polinômio interpolador. Note que diferentemente dos monômios, elas são muito diferentes entre si, nas vizinhanças ao redor do zero.



Funções de Newton

Os polinômios de Newton são definidos como:

$$
\begin{aligned}
&N_1(x) = 1 \\
&N_2(x) = x - x_1 \\
&N_3(x) = (x - x_1)(x - x_2) \\
&N_4(x) = (x - x_1)(x - x_2)(x - x_3) \\
&\dots \\
&N_n(x) = (x - x_1)(x - x_2)\dots(x - x_{n-1}) = \prod_{j=1}^{n-1} (x - x_j)
\end{aligned}
$$

Para todo i, o polinômio $N_i(x)$ possui grau i – 1. Além disso, uma propriedade fundamental de tais polinômios é que $N_i(x_j) = 0$ para todo j < i, o que faz com que a matriz associada ao problema de interpolação polinomial seja definida como:

$$
\begin{pmatrix}
1 & 0 & \cdots & 0 & 0 \\
1 & x_2 - x_1 & 0 & 0 & 0 \\
\vdots & \vdots & \ddots & \ddots & \vdots \\
1 & x_{n-1} - x_1 & \cdots & \prod_{j=1}^{n-2}(x_{n-1} - x_j) & 0 \\
1 & x_n - x_1 & \cdots & \prod_{j=1}^{n-2}(x_n - x_j) & \prod_{j=1}^{n-1}(x_n - x_j)
\end{pmatrix}
$$

Isso significa que o sistema linear é triangular inferior, de modo que a solução pode ser obtida de maneira eficiente (solução trivial), bastando para isso utilizar um esquema de substituição progressiva.

$$a_1 = f_1$$

$$a_2 = \frac{f_2 - a_1}{x_2 - x_1}$$

…

$$a_n = \frac{f_n - \sum_{i=1}^{n-1} a_i \prod_{j=1}^{i-1} (x_n - x_j)}{\prod_{j=1}^{i-1} (x_n - x_j)}$$

Voltemos ao exemplo anterior. Sejam os pontos a seguir:

x_i	-2	-1	0	1	2
$y_i = f(x_i)$	-9	-15	-5	-3	39

Desejamos encontrar o polinômio interpolador com funções de Newton. Como temos n = 5 pontos, temos que gerar os 5 primeiros polinômios de Newton. Neste caso, eles são:

$$N_1(x) = 1$$
$$N_2(x) = x - (-2) = (x + 2)$$
$$N_3(x) = (x + 2)(x + 1)$$
$$N_4(x) = (x + 2)(x + 1)x$$
$$N_5(x) = (x + 2)(x + 1)x(x - 1)$$

A matriz referente ao problema da interpolação polinomial neste caso fica:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & (x_2 - x_1) & 0 & 0 & 0 \\ 1 & (x_3 - x_1) & (x_3 - x_1)(x_3 - x_2) & 0 & 0 \\ 1 & (x_4 - x_1) & (x_4 - x_1)(x_4 - x_2) & (x_4 - x_1)(x_4 - x_2)(x_4 - x_3) & 0 \\ 1 & (x_5 - x_1) & (x_5 - x_1)(x_5 - x_2) & (x_5 - x_1)(x_5 - x_2)(x_5 - x_3) & (x_5 - x_1)(x_5 - x_2)(x_5 - x_3)(x_5 - x_4) \end{bmatrix}$$

Substituindo os valores temos:

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 2 & 2 & 0 & 0 \\ 1 & 3 & 6 & 6 & 0 \\ 1 & 4 & 12 & 24 & 24 \end{bmatrix}$$

Resolvendo o sistema $Ma = y$, o polinômio interpolador torna-se:

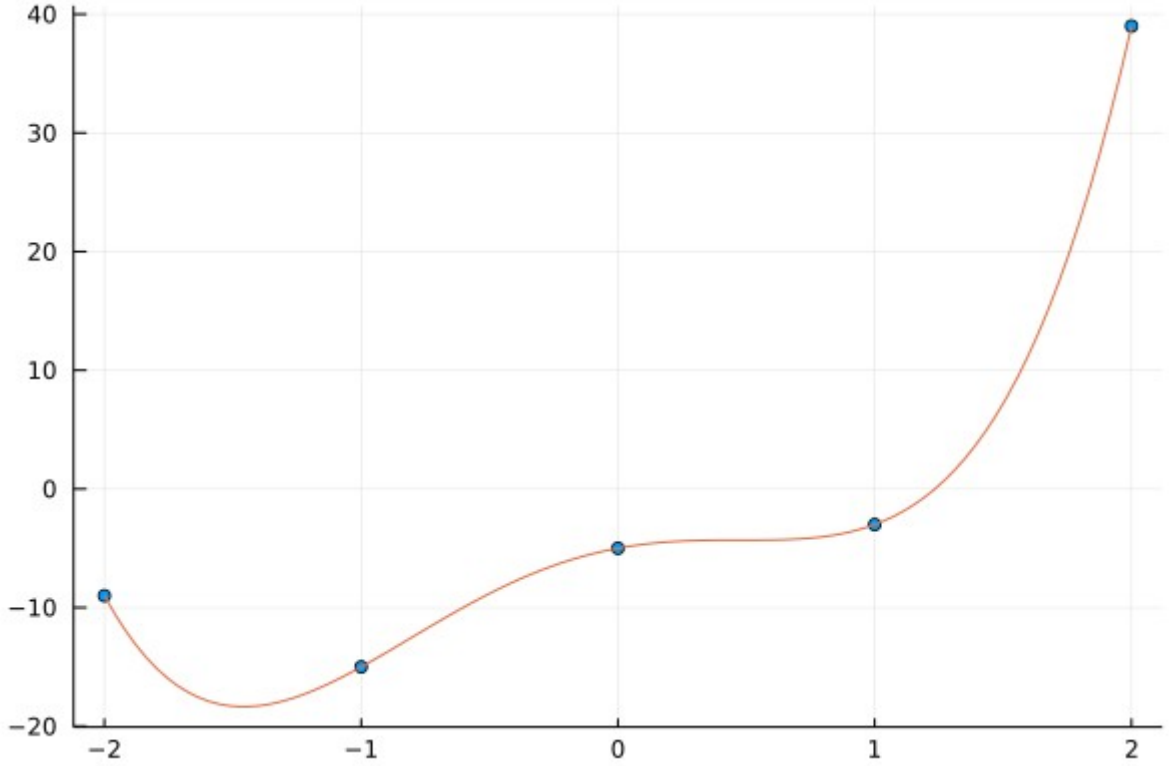
$$P_{n-1}(x) = a_1 N_1(x) + a_2 N_2(x) + a_3 N_3(x) + a_4 N_4(x) + a_5 N_5(x)$$

A seguir apresentamos uma função em Julia para construir o polinômio interpolador com as funções de Newton.

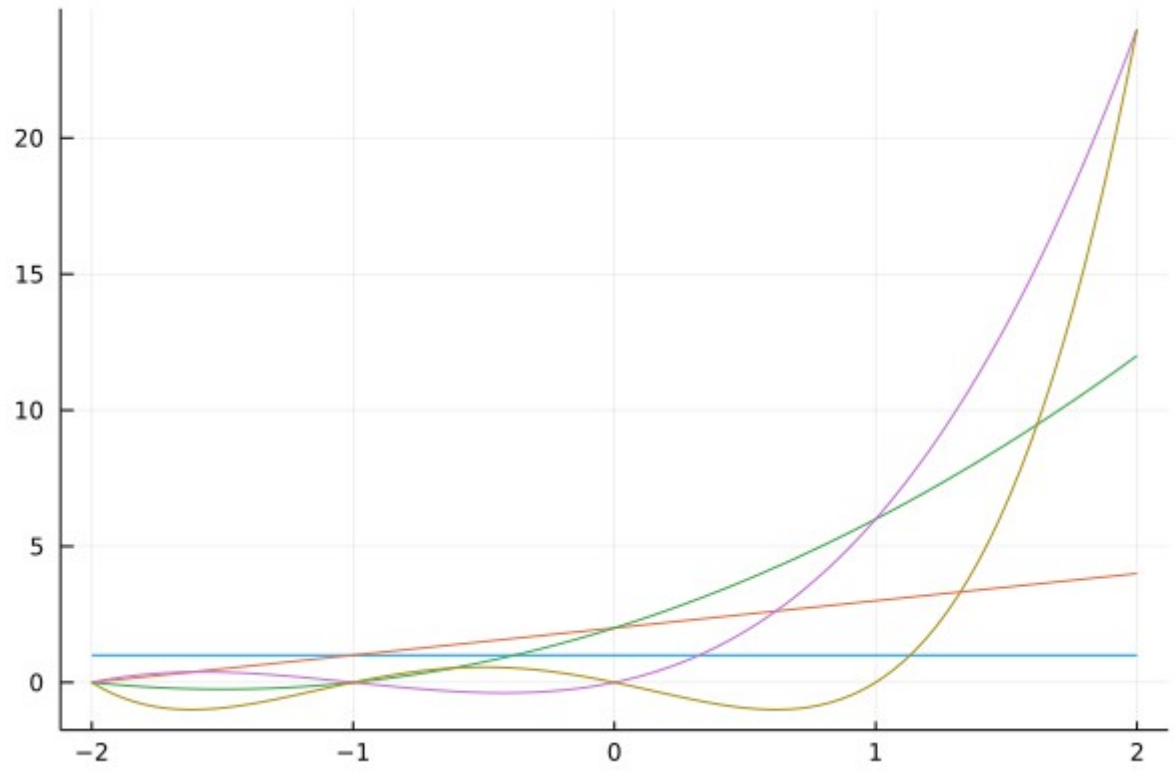
```
# Função que define e imprime sistema na tela
function define_pontos()

    # Define os pontos x e os valores de y = f(x)
    x = [-2.0, -1.0, 0.0, 1.0, 2.0]
    y = [-9.0, -15.0, -5.0, -3.0, 39.0]

    return (x, y)

end

# Função que plota o gráfico do polinômio (Newton)
function interpolacao_newton(x, y)

    if length(x) != length(y)
        println("Dados inválidos...")
        return -1
    else
        n = length(x)
        eixo_x = LinRange(minimum(x), maximum(x), 100)
        # Define polinômios de Newton
        N1 = 1
        N2 = (eixo_x .- x[1])
        N3 = (eixo_x .- x[1]).*(eixo_x .- x[2])
        N4 = (eixo_x .- x[1]).*(eixo_x .- x[2]).*(eixo_x .- x[3])
        N5 = (eixo_x .- x[1]).*(eixo_x .- x[2]).*(eixo_x .- x[3])
                                                .*(eixo_x .- x[4])

        # Define a matriz do problema de interpolação
        M = zeros(n, n)
        for i in 1:n
            for j in 1:n
                if j == 1
                    M[i, j] = 1
                elseif i >= j
                    # Calcula produtório
                    produto = 1
                    for k in 1:j-1
                        produto *= (x[i] - x[k])
                    end
                    M[i, j] = produto
                end
            end
        end
```

```
        # Resolve sistema linear
        a = M\y
        # Constrói polinômio interpolador
        eixo_y=a[1].*N1.+a[2].*N2.+a[3].*N3.+a[4].*N4.+a[5]*N5

        scatter(x, y, legend=false)
        plot!(eixo_x, eixo_y)
        savefig("Newton.png")
    end

end
```

A figura a seguir ilustra o polinômio interpolador obtido. Como era de se esperar, o resultado é muito próximo do obtido pelos métodos anteriores, uma vez que o número de pontos é pequeno. Quando n cresce e a matriz de Vandermonde torna-se mal condicionada os resultados tendem a se diferenciar.



A figura a seguir ilustra os polinômios de Newton utilizados como funções de base para o caso em questão. É possível notar que possuem baixa variação no semi eixo negativo e maior variação no semi eixo positivo.



Go down deep enough into anything and you will find mathematics.
— Dean Schlicter

Aula 13: Convolução e a filtragem linear de imagens

Em termos computacionais, a operação de convolução discreta consiste em um processo de média móvel ponderada, em que uma máscara (filtro ou kernel) desliza sobre a imagem para calcularmos a soma dos produtos entre os elementos da máscara com os elementos da imagem. Veremos a seguir alguns fundamentos matemáticos sobre essa operação.

A convolução 2D contínua

Matematicamente, a convolução de duas funções f(x,y) (representa a imagem de entrada) e h(x,y) (representa o filtro ou *Point Spread Function* – PSF, também conhecida como função de espalhamento pontual) é definida por:

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

Em termos geométricos, essa equação mede a variação do volume da intersecção das duas funções quando uma versão espelhada de h(x, y) se desloca de menos infinito a mais infinito no plano (x, y) como função do deslocamento. É uma operação mais complicada de ser resolvida analiticamente.

A convolução 2D discreta

Neste caso, ao invés de funções, trabalhamos com matrizes, que são representações discretas de imagens digitais. Supondo que h defina o filtro de dimensões $w\times w$ (mascara de tamanho 3x3, 5x5, 7x7, …) e x defina a imagem de entrada, então a imagem filtrada é dada por:

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

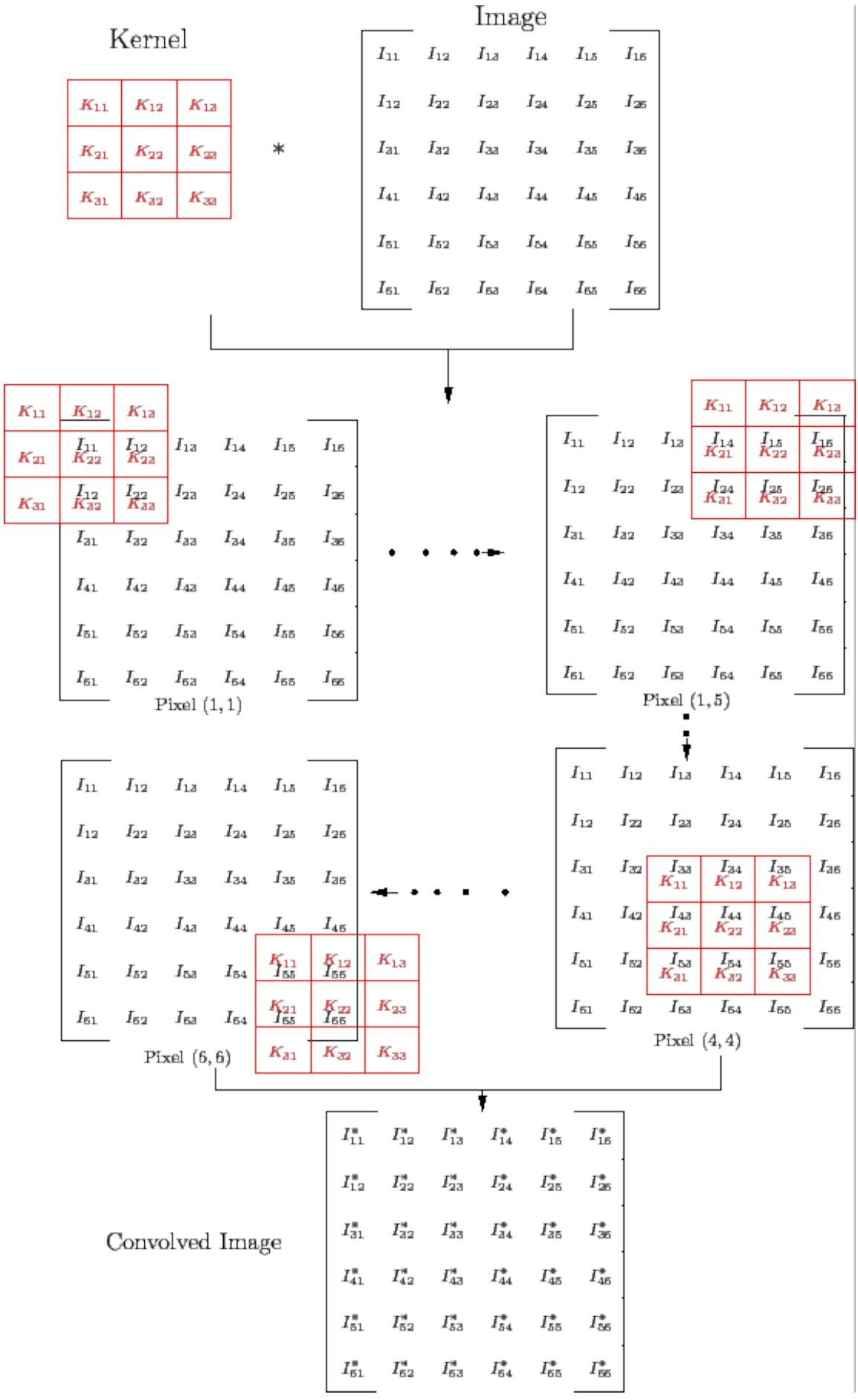
A interpretação da fórmula consiste em deslizar uma versão rebatida da máscara sobre todos os pontos da imagem (sinal bidimensional) e efetuar a operação de convolução sobre cada elemento. Custo computacional alto, principalmente se máscara for grande (loops múltiplos).

OBS: Em geral para lidar com o problema de valor de contorno (bordas), recomenda-se realizar uma de duas opções ao invés de simplesmente substituir os vazios por zeros (especialmente se as dimensões da máscara forem muito elevadas):

i) implementar um processo circular (borda superior ligada com a borda inferior e borda esquerda ligada com borda direita). Imagem é vista como um toro.

ii) espelhar as bordas da imagem (rebate os pixels das bordas)

Nas implementações em Python e Julia, iremos adotar a segunda opção, implementada pela função **pad**.


Kernel
Image
Pixel (1, 1)
Pixel (1, 5)
Pixel (6, 6)
Pixel (4, 4)
Convolved Image

Filtragem linear espacial

Filtros são ditos lineares se podem ser aplicados através da operação de convolução

Filtros Passa-baixa

São filtros de suavização, que borram a imagem atenuando as altas frequências da imagem.

1) Filtro da Média

Define uma máscara (h[ ] 3x3, i.e.) que espalha cada impulso Delta de Dirac igualitariamente entre as células vizinhas. A soma dos coeficientes é igual a 1.

$$h[\,]=\begin{bmatrix}1/9 & 1/9 & 1/9\\ 1/9 & 1/9 & 1/9\\ 1/9 & 1/9 & 1/9\end{bmatrix}$$

2) Filtro Gaussiano

Seus pesos referem-se a probabilidades de uma distribuição Gaussiana. Podem ser aproximados pelo triângulo de Pascal.

n	2^n	Coeficientes (w)
1	2	1 1
2	4	1 2 1
3	8	1 3 3 1
4	16	**1 4 6 4 1**
5	32	1 5 10 10 5 1

$\vec{w}=[1\,4\,6\,4\,1]$

$$h(m,n)=\frac{1}{(2^n)^2}\vec{w}^T\vec{w}=\frac{1}{256}\begin{bmatrix}1 & 4 & 6 & 4 & 1\\ 4 & 16 & 24 & 16 & 4\\ 6 & 24 & 36 & 24 & 6\\ 4 & 16 & 24 & 16 & 4\\ 1 & 4 & 6 & 4 & 1\end{bmatrix}$$ Filtro gaussiano circularmente simétrico

Filtros Passa-altas

São filtros para detecção de bordas, atenuam as baixas frequências presentes na imagem.

→ Operadores diferenciais (baseado em derivadas)

a) <u>Gradiente</u>: Vetor que aponta para o sentido de maior crescimento da função

$$\nabla f(x,y)=\begin{bmatrix}\frac{\partial}{(\partial x)}f(x,y)\\ \frac{\partial}{(\partial y)}f(x,y)\end{bmatrix}$$ (magnitude alta significa transição abrupta/bordas)

b) <u>Divergente</u>

$\text{div}(f(x,y)) = \frac{\partial}{(\partial x)} f(x,y) + \frac{\partial}{(\partial y)} f(x,y)$ (retorna um escalar. valores altos = Bordas)

c) <u>Laplaciano</u> (soma das derivadas segundas nas direções x e y)

$$\nabla^2 f(x,y) = \frac{\partial^2}{(\partial x^2)} f(x,y) + \frac{\partial^2}{(\partial y^2)} f(x,y)$$

Diferenças finitas

$$\frac{\partial}{(\partial x)} f(x,y) \approx f(x+1,y) - f(x,y)$$

$$\frac{\partial}{(\partial y)} f(x,y) \approx f(x,y+1) - f(x,y)$$

$$\frac{\partial^2}{(\partial x^2)} f(x,y) \approx (f(x+1,y) - f(x,y)) - (f(x,y) - f(x-1,y))$$

$$\frac{\partial^2}{(\partial y^2)} f(x,y) \approx f(x,y-1) - 2f(x,y) + f(x,y+1)$$

Assim, $\nabla^2 f(x,y) \approx f(x-1,y) + f(x,y-1) - 4f(x,y) + f(x+1,y) + f(x,y+1)$

Portanto, a resposta impulsivo / filtro é dado por:

$h[\,] = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$, detecta bordas → Filtro não direcional (bordas em qualquer direção)

Muito usado pois resulta em bordas mais finas (cruzamento por zeros)

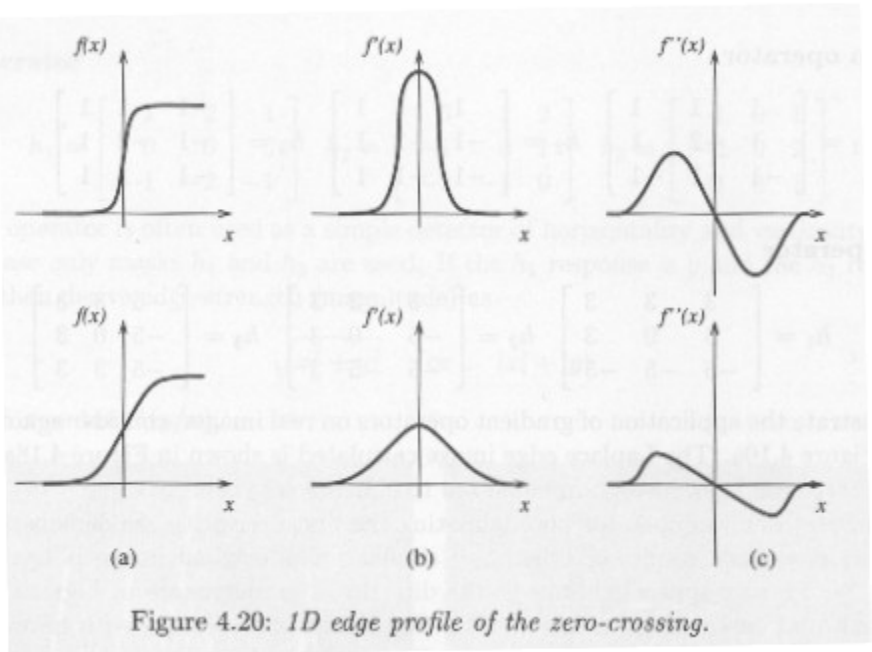


Figure 4.20: *1D edge profile of the zero-crossing.*

Outros modelos na literatura:
→ Filtros direcionais: detectam bordas horizontais e verticais
→ Sobel, Prewitt, Roberts

A seguir apresentamos o script Python filtragem_linear.py que contém a implementação de diversas funções para filtragem linear de imagens.

```
import time
import numpy as np
from skimage.io import imread
from skimage.io import imsave
from skimage.color import rgb2gray
from math import factorial
from numpy import outer


# Implementa a convolução 2D
def convolucao2D(img, h):
    # Dimensões da imagem
    M, N = img.shape
    # Dimensões do filtro
    m, n = h.shape
    # Verifica se o filtro é válido (quadrado)
    if (m == n):
        k = m // 2      # Tamanho da borda
        # Adiciona uma borda ao redor da imagem
        img = np.pad(img, ((k, k), (k, k)), mode='symmetric')
        # Dimensões da imagem expandida
        row, col = img.shape
        # Cria imagem de saída
        out = np.zeros((M, N))
        # Percorre miolo da imagem (descarta bordas)
        for i in range(k, row-k):
            for j in range(k, col-k):
                out[i-k,j-k] = np.sum(img[i-k:i+(k+1),j-k:j+(k+1)]*h)

        return out


# Função que gera um filtro da média tamanho n
def filtro_media(n):

    h = (1/(n**2))*np.ones((n, n))

    return h
```

```
# Função que gera filtro Gaussiano circularmente simétrico
def filtro_gaussiano(n):

    w = []

    for i in range(n+1):
        binomial = factorial(n)/(factorial(i)*(factorial(n-i)))
        w.append(binomial)

    # Converte de lista para array
    w = np.array(w)
    # Faz o produto externo
    h = outer(w, w)
    # Normaliza os coeficientes do filtro
    h = h/np.sum(h)

    return h


# Função que gera um filtro Laplaciano
def filtro_laplaciano(viz):

    if viz == 4:
        h = np.array([[0, 1, 0], [1, -4, 1], [0, 1, 0]])
    else:
        h = np.array([[1, 1, 1], [1, -8, 1], [1, 1, 1]])

    return h


# Função que aplica um filtro linear na imagem via convolução
# Se modo = media ou gaussiano, ordem = 3, 5, 7, 9, etc..
# Se modo = laplaciano, ordem = 4 ou 8
def filtra_imagem(nome_imagem, modo, ordem):
    # Leitura da imagem
    img = imread(nome_imagem)

    if modo == 'media' or modo == 'gaussiano':

        if modo == 'media':
            h = filtro_media(ordem)
        else:
            h = filtro_gaussiano(ordem)

        y = convolucao2D(img, h)

        imsave('Borrada.png', y.astype('uint8'))
```

```
    else:

        h = filtro_laplaciano(ordem)
        y = convolucao2D(img, h)
        y = abs(y)   # ou y = np.sqrt(y**2) - eliminar negativos
        imsave('Bordas.png', y.astype('uint8'))
```

No ambiente IPython, basta executar o script para carregar as funções na memória:

```
run filtragem_linear
```

Chamando a função e medindo o tempo de execução:

```
inicio = time.time()
filtra_imagem('Man.tiff', 'gaussiano', 8)
fim = time.time()
print('Elapsed time: %f' %(fim - inicio))
```

Imagem original

Imagem filtrada (Python)

```
Elapsed time: 7.84
```

O tempo gasto para o processamento da imagem em Python foi de aproximadamente 8 segundos para um filtro Gaussiano de ordem 8, ou seja, um kernel de tamanho 9 x 9.

A plataforma computacional utilizada para executar os experimentos é composta por um processador Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz de 64 bits, 16 Gb de memória RAM e SSD de 512 Gb com Ubuntu Linux versão 20.04.

A seguir veremos a mesma implementação na linguagem Julia. A ideia é comparar os tempos de execução entre as duas linguagens: Python e Julia.

```
using Images

# Implementa a convolução 2D
function convolucao2D(img, h)
    # Dimensões da imagem
    M, N = size(img)
    # Dimensões do filtro
    m, n = size(h)
    # Verifica se o filtro é válido (quadrado)
    if (m == n)
        k = div(m, 2)      # Tamanho da borda
        # Adiciona uma borda ao redor da imagem
        img = padarray(img, Pad(k, k))
        # Dimensões da imagem expandida
        row, col = size(img)
        # Cria imagem de saída
        new_image = zeros((M, N))
        # Percorre miolo da imagem (descarta bordas)
        for i in 1:M
            for j in 1:N
                new_image[i, j] = sum(img[i-k:i+k, j-k:j+k].*h)
            end
        end

        return new_image

    end

end

# Função para gerar o filtro da média
function filtro_media(n)

    h = (1/n^2)*ones(n, n)

    return h

end

# Função para gerar um filtro Gaussiano
function filtro_gaussiano(n)

    w = []
    for i in 0:n
        bin = factorial(n)/(factorial(i)*(factorial(n-i)))
        push!(w, bin)
    end
    h = w*w'/sum(w*w')

    return h

end
```

```
# Função que gera o filtro Laplaciano
function filtro_laplaciano(viz)

    if viz == 4
        h = [0 1 0; 1 -4 1; 0 1 0]
    else
        h = [1 1 1; 1 -8 1; 1 1 1]
    end

end

# Função que aplica um filtro linear na imagem via convolução
# Se modo = media ou gaussiano, ordem = 3, 4, 5, 6, 7, 8, etc..
# Se modo = laplaciano, ordem = 4 ou 8
function filtra_imagem(nome_imagem, modo, ordem)

    img = load(nome_imagem)

    if modo == "media" || modo == "gaussiano"

        if modo == "media"
            h = filtro_media(ordem)
        else
            h = filtro_gaussiano(ordem)
        end

        y = convolucao2D(img, h)

        save("Borrada_Julia.png", y)

    else

        if ordem == 4 || ordem == 8
            h = filtro_laplaciano(ordem)
        else
            println("Parâmetros inválidos para Laplaciano")
            return -1
        end

        y = convolucao2D(img, h)
        y = abs.(y)         # ou sqrt.(y.^2)
        y = clamp!(y, 0, 1)  # clipa valores para [0, 1]

        save("Bordas_Julia.png", y)

    end

end
```

Imagem filtrada (Julia)

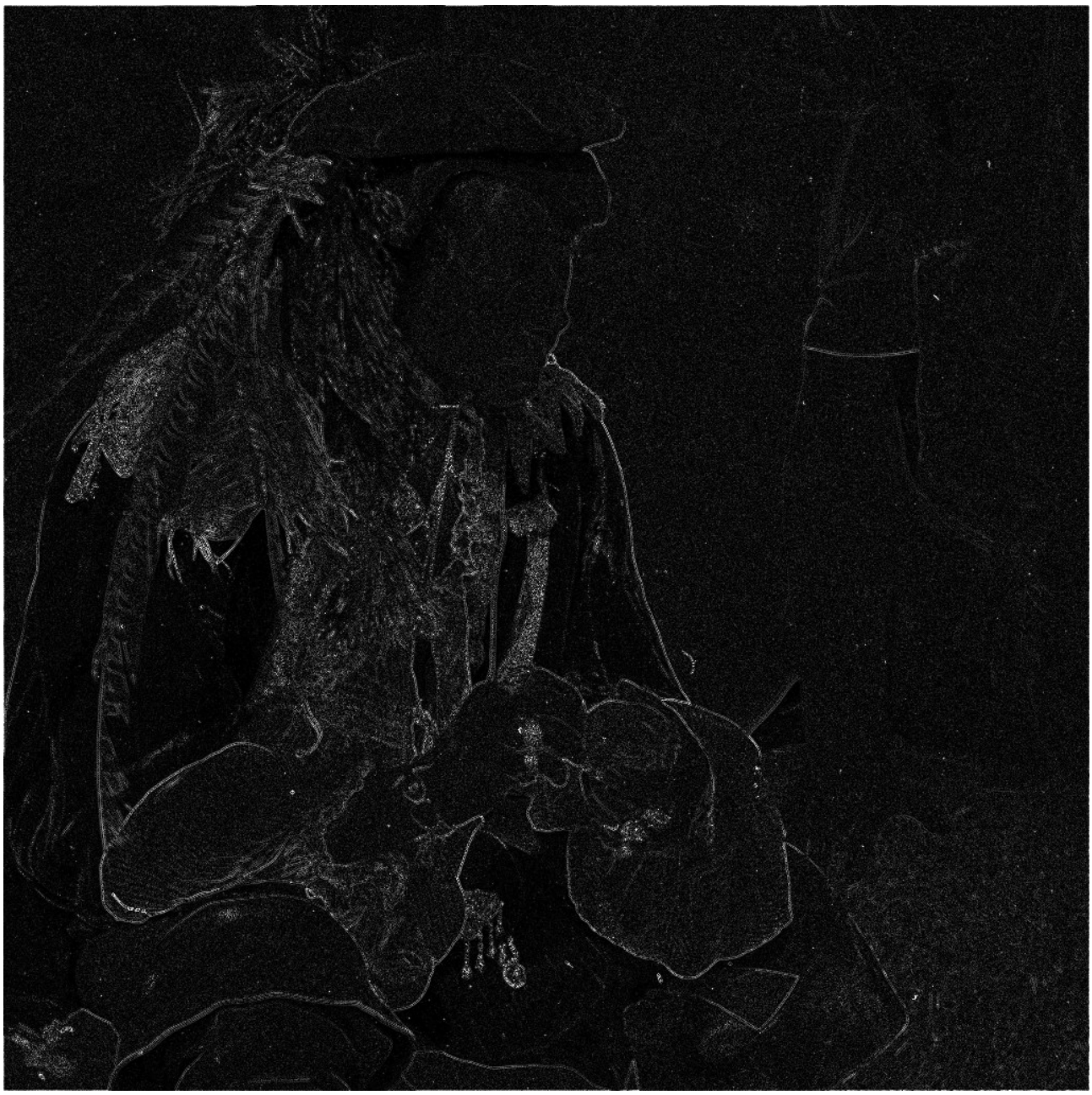
```
@time filtra_imagem("Man.tiff", "gaussiano", 8)
```

```
Elapsed time: 0.790
```

O tempo gasto para o processamento da imagem em Julia foi de menos de 1 segundo, o que em comparação com Python é cerca de 10 vezes mais rápido (9.92). Em seguida comparamos a filtragem passa-altas com o operador Laplaciano 3 x 3.

Detecção de bordas (Python)

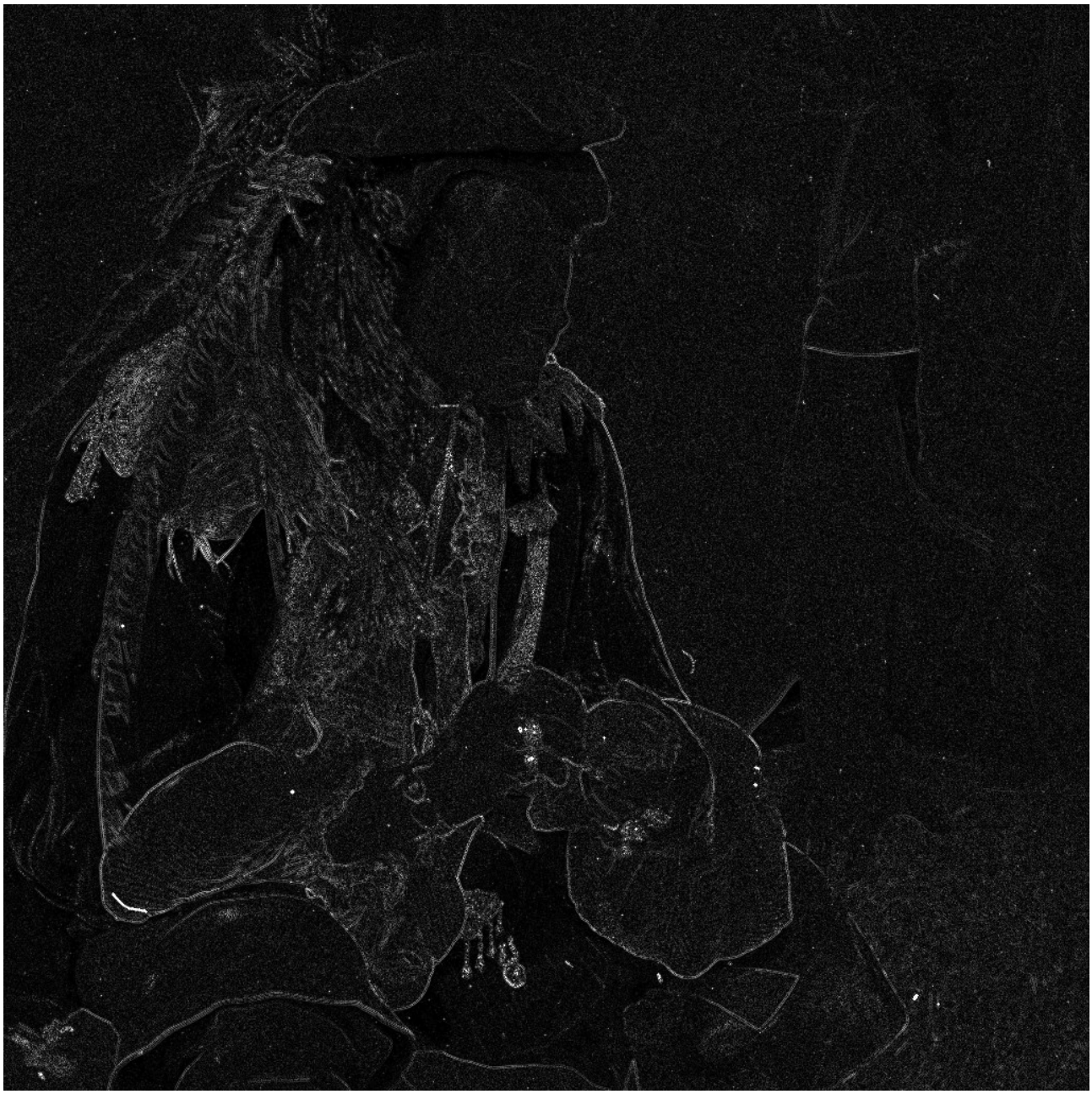
```
inicio = time.time()
filtra_imagem('Man.tiff', 'laplaciano', 4)
fim = time.time()
print('Elapsed time: %f' %(fim – inicio))
```



```
Elapsed time: 5.75
```

Detecção de bordas (Julia)

```
@time filtra_imagem("Man.tiff", "laplaciano", 4)
```



```
Elapsed time: 0.409
```

Mais uma vez, é possível verificar que o tempo gasto em Julia é bem menor, sendo cerca de meio segundo. Em termos comparativos, isso representa que o programa em Julia é aproximadamente 14 vezes mais rápido.

"If you're always trying to be normal you will never know how amazing you can be."
-- Maya Angelou

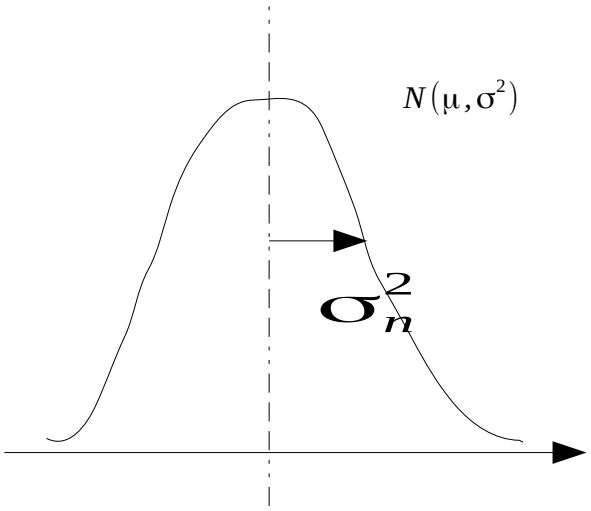
Aula 14: Filtragem de mínimos quadrados – o filtro de Wiener

A formulação matemática do filtro de Wiener é realizada em termos estatísticos, assumindo um modelo Gaussiano para o ruído.

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 → Variável Aleatória Normal (0, σ_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: $\hat{s}(n)=\alpha f(n)+\beta$ → 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:

$E[f(n)]=E[s(n)]$ ou seja $\mu_f=\mu_s$ (média do sinal observado = média do sinal original pois na média os erros positivos e negativos se cancelam)

$Var[f(n)]=Var[s(n)]+Var[\eta(n)]$ 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 , temos um sistema com duas equações:

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

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

Simplificando a equação (I), temos:

$$E[\alpha f(n)^2 + \beta f(n) - s(n) f(n)] = 0$$
$$\alpha E[f(n)^2] + \beta E[f(n)] - E[s(n) f(n)] = 0$$
$$\alpha E[f(n)^2] + \beta E[f(n)] - E[s(n)(s(n) + \eta(n))] = 0$$

Note que Var[x] = E[x²] – E²[x] o que implica em E[x²] = Var[x] + E²[x]. Assim, temos:

$$\alpha(\sigma_f^2 + \mu_f^2) + \beta \mu_f - (\sigma_s^2 + \mu_s^2) = 0$$
$$\alpha(\sigma_f^2 + \mu_f^2) + \beta \mu_f - \sigma_s^2 - \mu_s^2 = 0$$

Simplificando a equação (II), temos:

$\alpha E[f(n)] + \beta - E[s(n)] = 0$ (note que $\mu_f = \mu_s$)
$\alpha \mu_f + \beta - \mu_f = 0$

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

Substituindo a expressão de β na equação (I) nos leva a:

$\alpha(\sigma_f^2 + \mu_f^2) + (1 - \alpha) \mu_f \mu_f - \sigma_s^2 - \mu_s^2 = 0$ (note que $\mu_s^2 = \mu_f^2$)
$\alpha \sigma_f^2 + \alpha \mu_f^2 + \mu_f^2 - \alpha \mu_f^2 - \sigma_s^2 - \mu_f^2 = 0$

$$\alpha = \frac{\sigma_s^2}{\sigma_f^2} = \frac{\sigma_s^2}{(\sigma_s^2 + \sigma_n^2)}$$

Portanto o estimador $\hat{s}(n)$ pode ser escrito como:

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

o que após simplificações nos gera:

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

A ideia consiste em estimar os parâmetros μ_f e σ_f^2 diretamente do sinal observado e σ_n^2 é conhecido a priori nas simulações computacionais. Em caso de imagens reais, é preciso estudar maneiras de estimar a variância do ruído, mas esse é um problema mais avançado que foge do escopo da nossa aula.

Utilizando a média e a variância amostrais, os estimadores dos parâmetros do filtro de Wiener podem ser computados por:

$$\mu_f = \frac{1}{N}\sum_n f(n)$$
$$\sigma_f = \frac{1}{N}\sum_n (f(n)-\mu_f)^2$$
(média e variância amostrais)

Uma observação importante é que esses parâmetros não são globais, mas sim locais, pois a grande maioria das imagens reais é um sinal 2D não estacionário, ou seja, a média e a variância mudam de acordo com a região (são variantes no espaço). Assim, a estimação de tais parâmetros é realizada localmente, usando uma janela adaptativa ao redor do pixel atual, tipicamente quadrada e com dimensões ímpares, ou seja, 3 x 3, 5 x 5, 7 x 7, etc. Em termos práticos, temos que:

$f(n)$: é o ponto central da janela no pixel n da imagem ruidosa de dimensões 3 x 3, 5 x 5, ...)
μ_f é a média dos pixels da janela local
σ_f^2 é a variância dos pixels da janela local
σ_n^2 consiste numa constante referente ao nível de ruído (quanto maior, mais degradada imagem).

Filtro de Wiener Adaptativo Pontual

Lembrando que:

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

e sabendo que 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 é média local numa janela (3x3, 5x5, 7x7, etc.) na imagem observada e σ_s^2 é a variância local na janela referente a imagem livre de ruído, a pergunta que surge 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)

Nos experimentos computacionais descritos aqui, adotaremos a estratégia 1), que consiste em realizar a subtração entre a variância estimada na janela local referente a imagem observada (ruidosa) e a variância do ruído, caso essa diferença seja maior que zero (pois não existe variância negativa). No caso de obtermos um valor negativo, ele é substituído por zero.

A seguir mostramos um algoritmo para a filtragem de imagens com o método proposto.

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 extremos para ser analisados: quando o ganho do filtro tende a zero e quando o ganho do filtro tende a 1.

→ 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, borra 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, quase não filtra, pois ele entende que não há ruído, apenas sinal)

Na grande maioria dos casos, o filtro atua de modo intermediário, ou seja, de maneira a encontrar um compromisso entre borrar e preservar detalhes e bordas. Quanto mais detalhes finos na janela, menos o pixel em questão será filtrado. Note que o filtro, adapta-se aos dados, o que não ocorre na filtragem linear via convolução. Na convolução, a mesma operação é realizada em todos os pixels, sendo detalhes finos e bordas ou não. Por isso, os filtros lineares são conhecidos por suavizar demais o sinal, causando um borramento desnecessário à imagem.

A seguir é apresentado um conjunto de funções em Julia que implementa o filtro de Wiener adaptativo pontual.

```
using Images, Statistics, Distributions, Random

# Função para gerar o filtro da média (quanto maior desvio, mais ruído)
function adiciona_ruido(img, desvio)

    # Obtém as dimensões da imagem
    n, m = size(img)
    # Cria distribuição normal para ruído
    # média e desvio padrão
    d = Normal(0, desvio)
    # Gera ruído aleatório
    ruido = rand(d, n, m)
```

```
    # Cria imagem ruidosa
    ruidosa = img + ruido
    # Clipa em 0 e 255
    clamp!(ruidosa, 0, 1)
    # Salva imagem em arquivo
    save("Imagem_Ruidosa.png", ruidosa)

    return ruidosa

end

# Função que implementa o filtro de Wiener adaptativo
# img: é a imagem ruidosa
# w: é o parâmetro que contrloa o tamanho da janela (2w+1)
# var_ruido: é o desvio padrão do ruído ao quadrado
function filtro_wiener(img, w, var_ruido)
    # Dimensões da imagem
    row, col = size(img)
    # Aloca matriz para o resultado
    filtrada = zeros(row, col)
    # Adiciona uma borda ao redor da imagem
    img = padarray(img, Pad(w, w))
    # Percorre imagem
    for i in 1:row
        for j in 1:col
            janela = img[i-w:i+w, j-w:j+w]
            media = mean(janela)
            variancia = var(janela)
            # Clipa variância em zero
            var_f = variancia - var_ruido
            if var_f < 0
                var_f = 0
            end
            # Filtra pixel atual
            filtrada[i, j] = media + (var_f/variancia)*(img[i, j] - media)
        end
    end

    save("Imagem_Filtrada.png", filtrada)

    return filtrada

end

# Função que adiciona ruído e filtra imagem com Wiener
function filtra_imagem(arquivo)

    # Carrega imagem do arquivo
    img = load(arquivo)
    # Define a variância do ruído
    desvio_ruido = 0.05
    # Adiciona ruído
    ruidosa = adiciona_ruido(img, desvio_ruido)     # desvio padrão
    # Filtra imagem
    filtrada = filtro_wiener(ruidosa, 2, desvio_ruido^2)    # variância

end
```

A figura a seguir ilustra uma comparação entre a imagem Lena ruidosa e a mesma imagem após a filtragem com o método descrito. É possível notar claramente que após a filtragem, o ruído é praticamente todo removido da imagem. O comando utilizado para produzir a imagem filtrada foi:

```
filtra_imagem(“lena.png”)
```

Imagem ruidosa (0.05) Imagem filtrada (5 x 5)

Imagem ruidosa (0.1) Imagem filtrada (7 x 7)

A problem well-stated is a problem half-solved.
-- Charles Kettering

Aula 15: Filtragem não local de imagens – o filtro NLM

A filtragem não local consiste em utilizar amostras similares mas que não necessariamente pertencem a uma janela local na vizinhança espacial ao redor do pixel a ser filtrado. Dessa forma, geramos uma estimativa não local para a média, o que nos permite suavizar o ruído presente na imagem, mas sem prejudicar os detalhes finos, além de evitar o surgimento de artefatos ao longo das bordas da imagens (transições abruptas do sinal).

O filtro non-local means (NLM) tenta tirar vantagem do alto nível de redundância presente em imagens naturais, a partir do escaneamento de uma vasta porção da imagem na busca por pixels similares, utilizando o conceito de medida de similaridade ente patches. Dada uma imagem ruidosa $x=\{x_i|i\in I\}$, o valor estimado do pixel livre de ruído, denotado por $NL[x](i)$, é computado como uma média ponderada de todos os pixels da imagem:

$$NL[x](i)=\frac{\sum_{j\in I} w(i,j)x_j}{\sum_{j\in I} w(i,j)} \qquad (*)$$

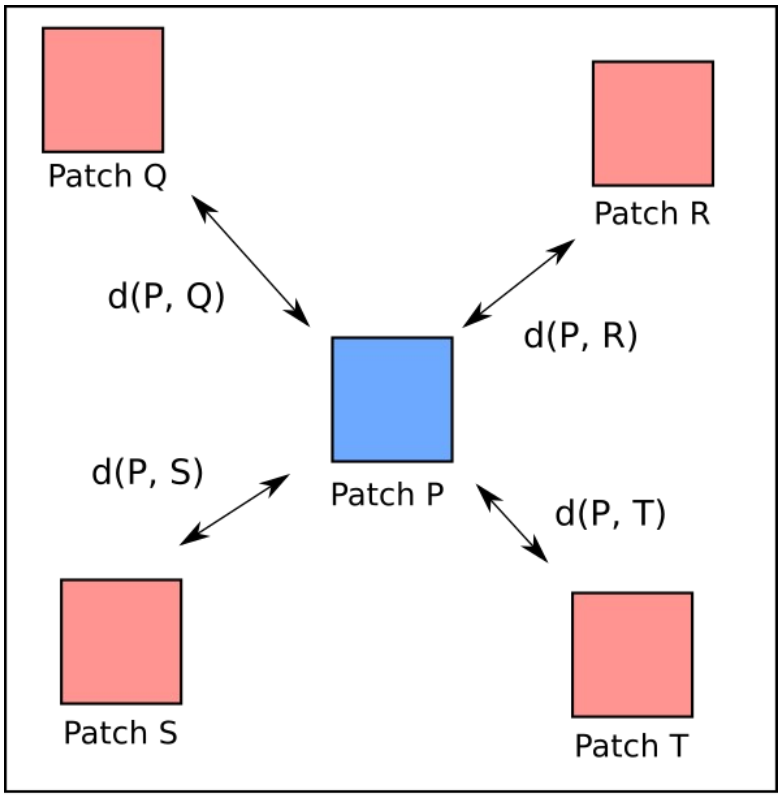
onde a família de pesos $\{w(i,j)\}$ depende da similaridade entre os pixels i e j. A similaridade entre dois pixels i e j depende da similaridade entre os vetores $x(\eta_i)$ e $x(\eta_j)$ formados pelos patches P_i e P_j (janelas locais ao redor dos pixels i e j). A ideia é que os pixels com vizinhança em tons de cinza similares produzam maiores pesos em média.

Os pesos dependem da distância Euclidiana entre os patches:

$$w(i,j)=\exp\left\{-\frac{\|x(\eta_i)-x(\eta_j)\|^2}{h^2}\right\}$$

onde h é um parâmetro que controla o decaimento da exponencial, atuando como um controle para o nível de suavização do filtro. Quanto maior h, maior o nível de suavização. Considerando uma imagem com n pixels, o algoritmo proposto tem um custo computacional da ordem de $O(n^2p)$, onde p denota o tamanho do patch. Em termos práticos, esse método seria inviável. Por essa razão, de modo a permitir um compromisso entre custo e desempenho, a estimativa do pixel livre de ruído, conforme a equação (*), o somatório em j não envolve todos os pixels da imagem. Ao invés disso, devemos considerar uma janela de busca de tamanho s (com s > p – tamanho do patch) ao redor do pixel a ser filtrado. Dessa forma, o custo computacional do algoritmo é reduzido para O(nps).

A figura a seguir ilustra o conceito do filtro NLM. Nele podemos ver que a janela de busca deve ser maior que a o patch.



Uma função em Julia que implementa o filtro NLM é apresentada a seguir. Note que os resultados obtidos por esse filtro são, em geral, melhores do que os resultados obtidos pelo filtro de Wiener adaptativo pontual.

```
using Images

# Função para gerar o filtro da média
# Bons desvios para ruído: 0.05, 0.1
function adiciona_ruido(img, desvio)

    # Obtém as dimensões da imagem
    n, m = size(img)
    # Cria distribuição normal para ruído
    # média e desvio padrão
    d = Normal(0, desvio)
    # Gera ruído aleatório
    ruido = rand(d, n, m)
    # Cria imagem ruidosa
    ruidosa = img + ruido
    # Clipa em 0 e 255
    clamp!(ruidosa, 0, 1)
    # Salva imagem em arquivo
    save("Imagem_Ruidosa.png", ruidosa)

    return ruidosa

end
```

```
# Função que implementa o filtro de Wiener adaptativo
# img: é a imagem ruidosa
# p: é o tamanho do patch (p=1->3x3, p=2->5x5, etc.)
# s: é o tamanho da janela de busca (s=5->11x11, s=7->15x15, etc.)
# h: parâmetro que controla a suavização
function filtro_nlm(img, p, s, h)
    # Dimensões da imagem
    row, col = size(img)
    # Aloca matriz para o resultado
    filtrada = zeros(row, col)
    # Adiciona uma borda ao redor da imagem (virtual)
    img = padarray(img, Pad(p, p))
    # Percorre imagem
    for i in 1:row
        for j in 1:col
            # Obtém o patch 1
            P1 = img[i-p:i+p, j-p:j+p]
            # Calcula as bordas da janela de busca
            rmin = max(i-s, 1);  # linha inicial
            rmax = min(i+s, row);  # linha final
            smin = max(j-s, 1);  # coluna inicial
            smax = min(j+s, col);  # coluna final
            # Calcula média ponderada
            NL = 0      # numerador
            Z = 0       # denominador
            # Loop para todos os pixels da janela de busca
            for r in rmin:rmax
                for s in smin:smax
                    # Obtém o patch a ser comparado
                    P2 = img[r-p:r+p, s-p:s+p]
                    # Calcula o quadrado da distância Euclidiana
                    d2 = sum((P1 - P2).*(P1 - P2))
                    # Calcula a medida de similaridade
                    sij = exp(-d2/(h^2))
                    # Atualiza Z e NL
                    Z = Z + sij
                    NL = NL + sij*img[r, s]
                end
            end
            filtrada[i, j] = NL/Z
        end
    end

    save("Imagem_Filtrada.png", filtrada)

    return filtrada

end
```

```
# Função que adiciona ruído e filtra imagem com NLM
function filtra_imagem(arquivo)

    # Carrega imagem do arquivo
    img = load(arquivo)
    # Parâmetros do NLM
    # tamanho do patch, tamanho da janela de busca e suavização
    p, s, h = 2, 5, 0.25
    # Adiciona ruído
    ruidosa = adiciona_ruido(img, 0.05)     # desvio padrão
    # Filtra imagem
    filtrada = filtro_nlm(ruidosa, p, s, h)

end
```

A figura a seguir ilustra o resultado do filtro NLM. Note como ele consegue eliminar o ruído preservando muitos detalhes finos na imagem, graças ao pouco borramento gerado.

Imagem ruidosa (0.05) Imagem filtrada (NLM)

We cannot become what we need to be by remaining what we are.
Max de Pree

Aula 16: Análise de Componentes Principais (PCA)

Antes de iniciarmos com a apresentação da técnica Análise de Componentes Principais, iremos discutir uma breve introdução aos autovalores e autovetores, ferramentas matemáticas que são muito importantes para a compreensão do método PCA.

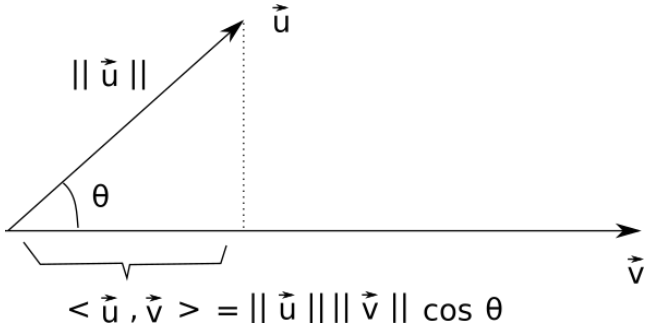
Autovalores e autovetores: uma breve revisão

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 $\vec{u}$

$\vec{u} = P\vec{v}$ (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 tendo seu tamanho modificado (sendo alongado ou comprimido)? Em termos matemáticos temos:

$\vec{u} = P\vec{v} = \lambda\vec{v}$ (*)

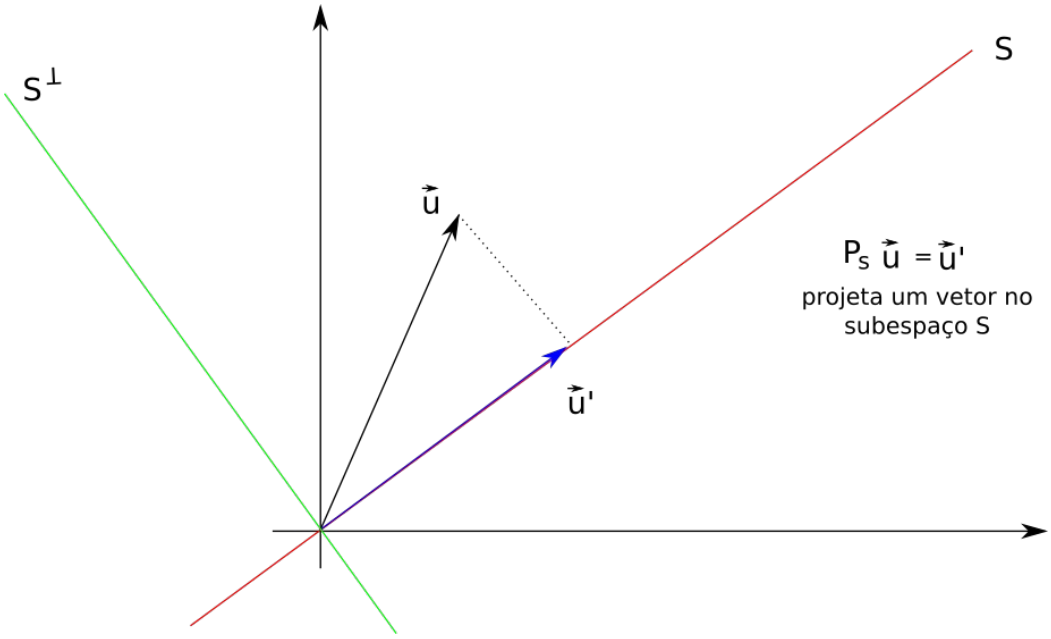
→ Todos os vetores $\vec{v}$ que satisfazem a equação (*) são chamados de autovetores de P
→ Todos os escalares λ que satisfazem a equação (*) são chamados de autovalores de P .

λ é o fator de escala da redução ou incremento de tamanho do vetor

Def: Dizemos que $\vec{v} \in R^n$ 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. Então, $I_{n\times n}\vec{v} = 1\vec{v}$,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:

$$A\vec{v} = \lambda \vec{v}$$

$$A\vec{v} - \lambda I \vec{v} = 0$$

$$(A - \lambda I)\vec{v} = 0$$

<u>Teorema:</u> 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:

$\frac{9}{4} - 2 \cdot \frac{3}{2}\lambda + \lambda^2 - \frac{1}{4} = 0$, que é a equação do segundo grau $\lambda^2 - 3\lambda + 2 = 0$, cujas soluções são:

$$\lambda_1 = \frac{3-1}{2} = 1$$

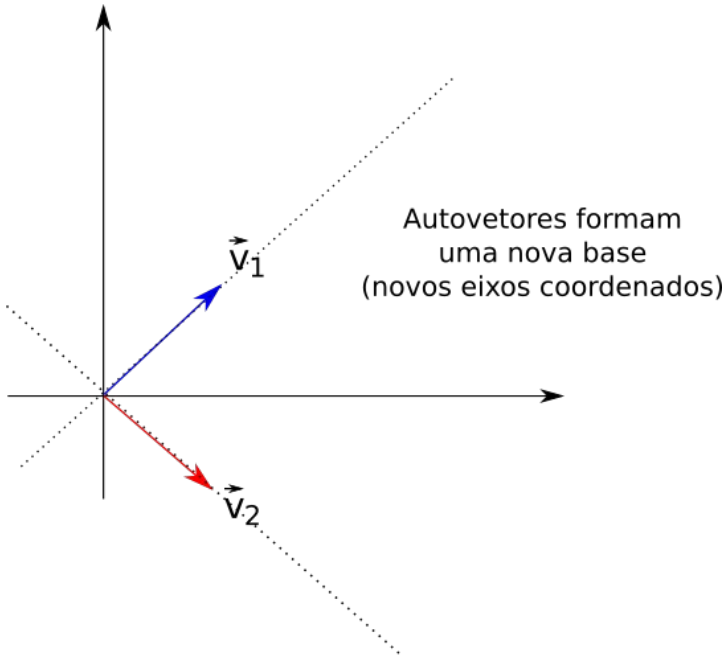
$$\lambda_2 = \frac{3+1}{2} = 2$$

Para obter o autovetor associado ao autovalor $\lambda_1 = 1$ temos:

$$\begin{bmatrix} 1/2 & -1/2 \\ -1/2 & 1/2 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \rightarrow \begin{cases} \frac{1}{2}v_1 - \frac{1}{2}v_2 = 0 \\ -\frac{1}{2}v_1 + \frac{1}{2}v_2 = 0 \end{cases} \rightarrow v_1 = v_2 \rightarrow \vec{v}^{(1)} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}$$

Para obter o autovetor associado ao autovalor $\lambda_2 = 2$ temos:

$$\begin{bmatrix} -1/2 & -1/2 \\ -1/2 & -1/2 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \rightarrow \begin{cases} -\frac{1}{2}v_1 - \frac{1}{2}v_2 = 0 \\ -\frac{1}{2}v_1 - \frac{1}{2}v_2 = 0 \end{cases} \rightarrow v_1 = -v_2 \rightarrow \vec{v}^{(2)} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} \end{bmatrix}$$



<u>Def:</u> Se P é um operador linear que possui n autovalores distintos então os autovetores de P definem uma nova base para R^n

$$Q = \begin{bmatrix} | & | & \dots & | \\ \vec{v}_1 & \vec{v}_2 & \dots & \vec{v}_n \\ | & | & \dots & | \end{bmatrix}$$ (matriz dos autovetores)

Obs: Uma matriz é dita positiva semidefinida se todos seus autovalores forem maiores ou iguais a 0

<u>Def:</u> 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.

Note que $P=\begin{bmatrix} | & - & \vec{p}_1 & - & | \\ | & - & \vec{p}_2 & - & | \\ ... & ... & ... & ... & ... \\ | & - & \vec{p}_n & - & | \end{bmatrix}$ e assim $PQ=\begin{bmatrix} \vec{p}_1\vec{v}_1 & \vec{p}_1\vec{v}_2 & ... & \vec{p}_1\vec{v}_n \\ \vec{p}_2\vec{v}_1 & \vec{p}_2\vec{v}_2 & ... & \vec{p}_2\vec{v}_n \\ ... & ... & ... & ... \\ \vec{p}_n\vec{v}_1 & \vec{p}_n\vec{v}_2 & ... & \vec{p}_n\vec{v}_n \end{bmatrix}$.

Observando PQ , podemos reescrever a matriz como $PQ=\begin{bmatrix} | & | & ... & | \\ P\vec{v}_1 & P\vec{v}_2 & ... & P\vec{v}_n \\ | & | & ... & | \end{bmatrix}$

O lado direito é dado por $Q\Lambda=\begin{bmatrix} v_{11}\lambda_1 & v_{12}\lambda_2 & ... & v_{1n}\lambda_n \\ v_{21}\lambda_1 & v_{22}\lambda_2 & ... & v_{2n}\lambda_n \\ ... & ... & ... & ... \\ v_{n1}\lambda_1 & v_{n2}\lambda_2 & ... & v_{nn}\lambda_n \end{bmatrix}=\begin{bmatrix} | & | & ... & | \\ \lambda_1\vec{v}_1 & \lambda_2\vec{v}_2 & ... & \lambda_{nP}\vec{v}_n \\ | & | & ... & | \end{bmatrix}$

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,...,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. Se P é 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 uma 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 daos encontram-se num subespaço Euclidiano do R^n
→ Método não supervisionado (não requer rótulos das classes)
→ Decorrelaciona os dados de entrada eliminando redundâncias

$$\vec{x}\in R^d=\begin{bmatrix} x_1 \\ x_2 \\ ... \\ x_d \end{bmatrix} \qquad \overset{\Rightarrow}{W_{pca}} \qquad \vec{y}\in R^k=\begin{bmatrix} y_1 \\ ... \\ y_k \end{bmatrix}, k \ll d$$

PCA pela Maximização da Variância

Seja $Z=[T^T, S^T]$ uma base ortonormal para R^d, como:

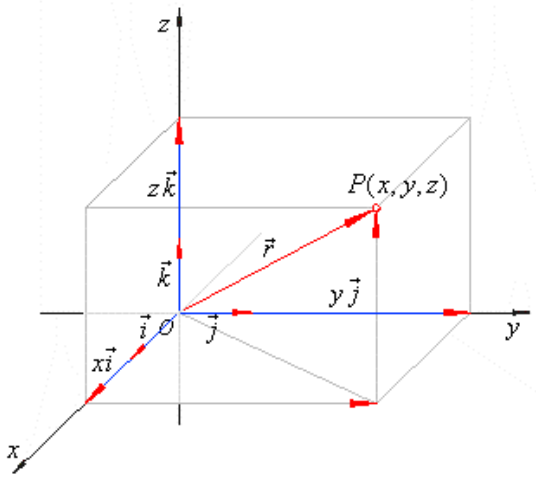
$T^T=[\vec{w}_1, \vec{w}_2, \ldots, \vec{w}_k]$ (d componentes que desejamos reter no processo de redução)
$S^T=[\vec{w}_{k+1}, \vec{w}_{k+2}, \ldots, \vec{w}_d]$ (componentes restantes que deve ser descartadas)

onde T representa o novo sistema de eixos coordenados (subespaço PCA) e S representa 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 $\vec{x} \in R^d$ como (expansão na base ortonormal):

$$\vec{x}=\sum_{j=1}^{d}(\vec{x}^T\vec{w}_j)\vec{w}_j=\sum_{j=1}^{d}c_j\vec{w}_j$$ (c_j são os coeficientes da expansão)



Assim, o novo vetor $\vec{y} \in R^k$ pode ser obtido pela transformação:

$$\vec{y}=T\vec{x} \rightarrow \vec{y}^T=\vec{x}^T T^T=\sum_{j=1}^{d}c_j\vec{w}_j^T[\vec{w}_1, \vec{w}_2, \ldots, \vec{w}_k]=[c_1, c_2, \ldots, c_k]$$

uma vez que usando a propriedade de ortonormalidade (base ortonormal):

$$\vec{w}_i^T\vec{w}_j=\begin{cases}1, & se \quad i=j\\ 0, & se \quad i\neq j\end{cases}$$

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 $E[\vec{y}^T \vec{y}] = E[y_1^2] + E[y_2^2] + ... + E[y_k^2]$ 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 $\vec{wj}$), podemos escrever (substituindo c em *):

$$J_1^{PCA}(T) = \sum_{i=1}^{k} E[\vec{w}_j^T \vec{x} \vec{x}^T \vec{w}_j] = \sum_{i=1}^{k} \vec{w}_j^T E[\vec{x}\vec{x}^T] \vec{w}_j = \sum_{j=1}^{k} \vec{w}_j^T \Sigma_x \vec{w}_j \qquad sujeito\ \ a \qquad \|\vec{w}_j\| = 1$$

em que $E[\vec{x}\vec{x}^T] = \Sigma_x$ denota a matriz de covariância dos dados observados.

O problema então consiste em resolver:

$$\underset{\vec{w}_j}{argmax} \sum_{j=1}^{k} \vec{w}_j^T \Sigma_x \vec{w}_j \qquad sujeito\ \ a \qquad \|\vec{w}_j\| = 1, \, para \ \ j = 1, 2, ..., k$$

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 ferramente 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 $f(x, y)$ 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:

$$L(x, y, \lambda_1, \lambda_2, ..., \lambda_m) = f(x, y) - \sum_{i=1}^{m} \lambda_i (g_i(x, y) - c_i)$$

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:

$$\frac{\partial}{\partial \vec{w}_j} J(T, \lambda_j) = \Sigma_x \vec{w}_j - \lambda_j w_j = 0$$

o que nos leva a:

$\Sigma_x \vec{w}_j = \lambda_j \vec{w}_j$ (problema de autovalores e autovetores)

e portanto 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:

$$\underset{\vec{w}_j}{argmax} \sum_{j=1}^{k} \vec{w}_j^T \Sigma_x \vec{w}_j = \underset{\vec{w}_j}{argmax} \sum_{j=1}^{k} \vec{w}_j^T \lambda_j \vec{w}_j = \underset{\vec{w}_j}{argmax} \sum_{j=1}^{k} \lambda_j \|\vec{w}_j\|^2 = \underset{\vec{w}_j}{argmax} \sum_{j=1}^{k} \lambda_j$$

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:

$$\Sigma_y = A^T Q \Lambda Q^T A$$

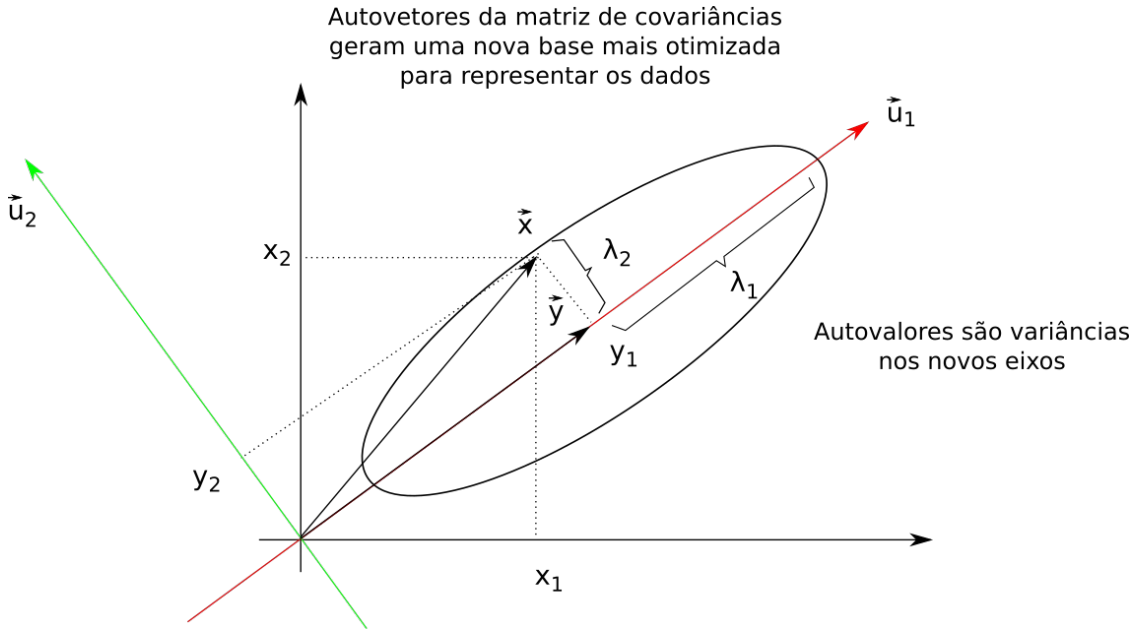
onde A é a matriz dos autovetores de Σ_x , ou seja, é igual a Q e portanto:

$$\Sigma_y = Q^T Q \Lambda Q^T Q = \Lambda = \begin{bmatrix} \lambda_1 & 0 & ... & 0 \\ 0 & \lambda_2 & ... & 0 \\ 0 & 0 & ... & 0 \\ 0 & 0 & ... & \lambda_n \end{bmatrix}$$

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



Algoritmo PCA:

1. Ler dataset $X=\{\vec{x}_1,\vec{x}_2,\vec{x}_3,\ldots,\vec{x}_n\}$ com $\vec{x}_i \in R^d$
2. Computar o vetor média μ_x a matriz de covariâncias: $\Sigma_x=\frac{1}{n}\sum_{i=1}^{n}(\vec{x}_i-\vec{\mu}_x)(\vec{x}_i-\vec{\mu}_x)^T$
3. Obter autovalores e autovetores de Σ_x
4. Selecionar os k autovetores associados aos k maiores autovalores da matriz de covariâncias, denotados por $\vec{w}_1,\vec{w}_2,\ldots,\vec{w}_k$
5. Definir a matriz de transformação $W_{PCA}=[\vec{w}_1,\vec{w}_2,\ldots,\vec{w}_k]$ ($n\,x\,k$)
6. Projetar dados na nova base: $\vec{y}=W_{PCA}^T\vec{x}_i$

As funções a seguir implementam o PCA em Julia utilizando o conjunto de dados Iris, que possui 150 amostras com 4 atributos cada, o que define uma matriz de dados de 150 linhas por 4 colunas.

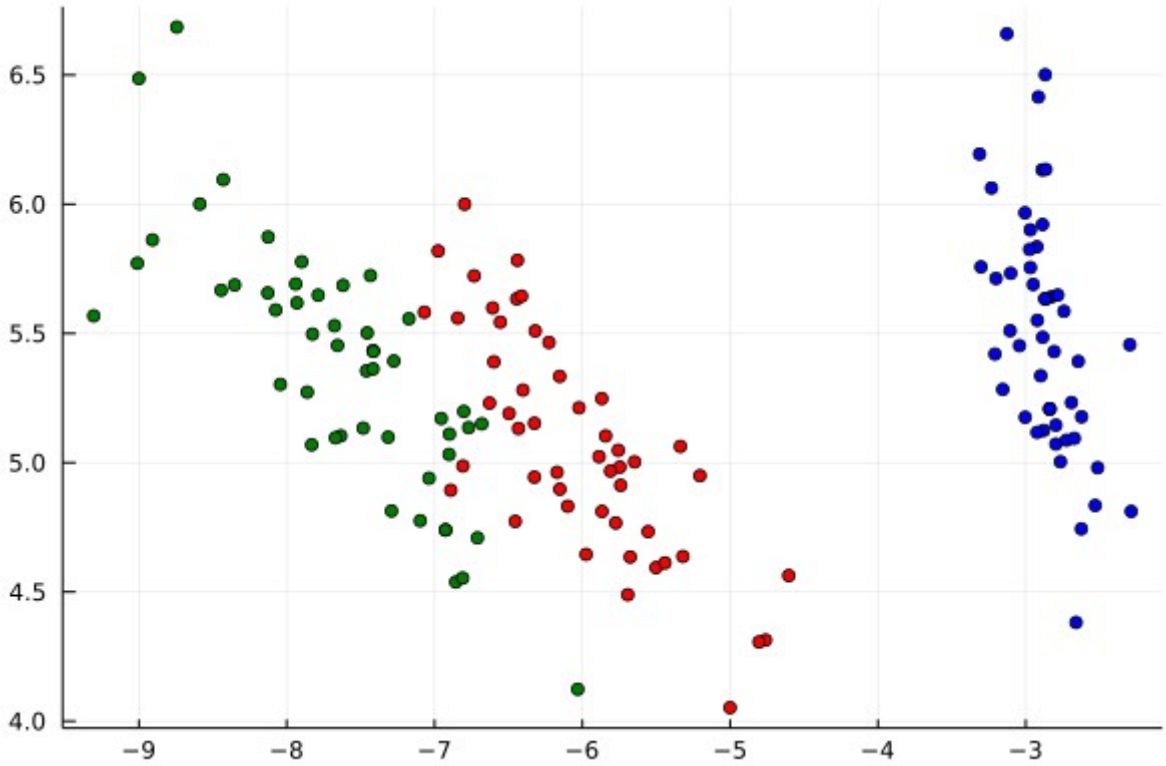
```
using DelimitedFiles, Statistics, LinearAlgebra, Plots

# Função que realiza o carregamento da base de dados Iris
function leitura_iris()
    # Carrega dataset do arquivo
    dados = readdlm("./iris/iris.data", ',')
    # Cria matriz de dados (exclui última coluna)
    X = dados[:, 1:end-1]
    # Carrega rótulos
    Y = dados[:, end]
    # Converte rótulos para números inteiros
    replace!(Y, "Iris-setosa" => 1)
    replace!(Y, "Iris-versicolor" => 2)
    replace!(Y, "Iris-virginica" => 3)

    return X
end
```

```
# Função que implementa o filtro de Wiener adaptativo
function PCA(X, d)

    # Calcula matriz de covariâncias
    mc = cov(X)
    # Computa os autovalores e autovetores
    vals = eigvals(mc)
    vecs = eigvecs(mc)
    # Pega a ordem decrescente dos autovalores
    ordem = sortperm(vals, rev=true)
    # Seleciona os d primeiros autovetores (maiores)
    maiores = vecs[:, ordem[1:d]]
    # Monta matriz de projeção
    Wpca = maiores'
    # Projeta dados
    dados_pca = Wpca*X'

    return dados_pca'

end

# Função para plotar gráfico de dispersão 2D
function plota_iris_2D(X)

    scatter(X[1:50, 1], X[1:50,2],color="blue",legend=false)
    scatter!(X[51:100,1], X[51:100, 2],color="red",legend=false)
    scatter!(X[101:150,1],X[101:150,2],color="green",legend=false)

    savefig(“Iris_PCA_2D.png”)

end
```

Para aplicar o PCA no conjunto de dados Iris e plotar um gráfico de dispersão utilizando as duas principais componentes, devemos incluir o arquivo fonte no REPL e chamar as funções na seguinte ordem:

```
include(“./pca.jl”)

X = leitura_iris()
dados_pca = PCA(X, 2)
plota_iris_2D(dados_pca)
```

A figura a seguir ilustra um gráfico de dispersão com as duas principais componentes do conjunto de dados Iris.



The greatest obstacle to discovery is not ignorance - it is the illusion of knowledge.
Daniel J. Boorstin

Bibliografia

Levada, A. L. M. Introdução à Computação em Julia: Problemas e Aplicações, Maio/2021. Disponível em: http://dx.doi.org/10.13140/RG.2.2.15777.53605/1

Lauwens, B.; Downey, A.B. Think Julia: How to Think Like a Computer Scientist, Editora O'Reilly, 2019. Disponível em https://benlauwens.github.io/ThinkJulia.jl/latest/book.html

Página oficial da linguagem Julia na internet sobre recursos de aprendizagem. Contém tutoriais, cursos online, vídeo aulas e exercícios. Disponível em: https://julialang.org/learning/

Wolfram, S. A New Kind of Science, Wolfram Research, 2002.

Castro, M. L. A.; Castro, R. O. Autômatos celulares: implementações de Von Neumann, Conway e Wolfram, Revista de Ciências Exatas e Tecnologia, vol. 3, n. 3, 2008. Disponível em: http://www.esalq.usp.br/lepse/imgs/conteudo_thumb/Aut-matos-Celulares-Implementa--es-de-Von-Neumann--Conway-e-Wolfram.pdf

A Regra 110 e suas propriedades. Disponível em: https://en.wikipedia.org/wiki/Rule_110,

Cook, M. Universality in elementary cellular automata, Complex systems, vol. 15, 2004, pp. 1-40. Disponível en http://www.complex-systems.com/pdf/15-1-1.pdf

May, R. M. "Simple mathematical models with very complicated dynamics". *Nature*. **261** (5560): 459–467, 1976. Available at: http://abel.harvard.edu/archive/118r_spring_05/docs/may.pdf

Epperson, J. F. An Introduction to Numerical Methods and Analysis, Wiley, 2nd edition, 2013.

Introduction to numerical methods for solving systems of linear equations. Disponível em: https://en.wikibooks.org/wiki/Introduction_to_Numerical_Methods/System_of_Linear_Equations

Claudio Hirofume Asano, Eduardo Colli, Cálculo Numérico – Fundamentos e Aplicações, IME/USP, 2009. Disponível em: https://homepages.dcc.ufmg.br/~assuncao/an/LivroNumerico.pdf

John O. Rawlings, Sastry G. Pantula, David A. Dickey. Applied Regression Analysis: a Research Tool, 2nd edition, Springer, 1998.

Montgomery, D. C.; Peck, E. A.; Vining, G. G. Introduction to Linear Regression Analysis, 5th Edition, Wiley, 2012.

Gonzalez, R.C.; Woods, R.E. Digital Image Processing, Pearson, 3rd edition, 2007.

J. S. Lee, Digital Image Enhancement and Noise Filtering by Use of Local Statistics, IEEE Trans. on Pattern Analysis and Machine Intelligence, vol. PAMI-2, no. 2, pp. 165-168, March 1980.

Antoni Buades, Bartomeu Coll, Jean-Michel Morel. A non-local algorithm for image denoising, Computer Vision and Pattern Recognition (CVPR), pp. 60-65, 2005.

Jolliffe, I. T. Principal Component Analysis, 2nd edition, Springer Series in Statistics, 2002.

Jackson, J.E. A User's Guide to Principal Components, Wiley, 1991.

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 visitar a página do LinkeIn:

“As coisas que você faz para si mesmo vão com você, mas as coisas que você faz para os outros permanecem como seu legado.”
(Kalu Ndukwe Kalu)