

Álgebra Linear

Prof. Dr. Wagner Hugo Bonat

Estrutura e objetivos do módulo

Estrutura do módulo

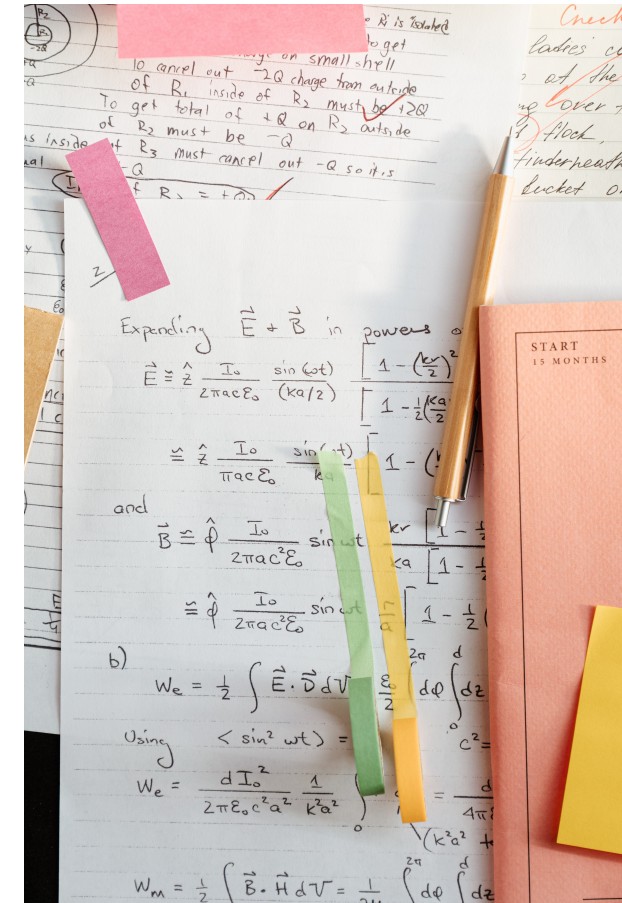
- ▶ Exemplo motivacional.
- ▶ Operações com vetores e matrizes.
- ▶ Métodos para a solução de sistemas de equações lineares.
 - ▶ Métodos diretos.
 - ▶ Métodos iterativos/indiretos.
- ▶ Decomposições matriciais e obtenção da inversa.
- ▶ Estratégias para melhorar a *performance*.
- ▶ Integrando o **R** e bibliotecas **C++** para álgebra linear.
- ▶ Exemplos com scripts **R** e **C++**.



<https://www.pexels.com/photo/top-view-of-people-at-the-meeting-3184287/>

Objetivos do módulo

- ▶ Apresentar algumas implementações computacionais **R** relacionadas a álgebra linear.
- ▶ Realizar operações com vetores e matrizes em **R** e **C++**.
- ▶ Resolver sistemas de equações lineares em **R** e **C++**.
- ▶ Realizar decomposições matriciais e obter a inversa.
- ▶ Discutir algumas estratégias para melhorar a *performance* computacional.
 - ▶ Matrizes esparsas.
 - ▶ Integrando o **R** com a **Armadillo**.
- ▶ Praticar com scripts em **R** e **C++**.



Exemplo motivacional

Motivação


- ▶ Álgebra linear é um ingrediente essencial da programação científica.
 - ▶ Modelos estatísticos.
 - ▶ Análise multivariada.
 - ▶ Análise de dados espaciais.
 - ▶ Séries temporais.
 - ▶ Algoritmos de aprendizagem de máquina.
- ▶ Essencial em pesquisa operacional.
- ▶ Importante para quem vai implementar novas metodologias.
- ▶ Alguns cuidados fazem muita diferença em termos práticos.

$$\begin{bmatrix} 1 & x+1 & x^2+1 \\ 1 & y+1 & y^2+1 \\ 1 & z+1 & z^2+1 \end{bmatrix} \quad x = \sum_{i=1}^n x_i v_i = x_1 v_1 + x_2 v_2 + \dots + x_n v_n \quad v_k = y_k - \sum_{i=1}^{k-1} \frac{(v_i, y_k)}{(v_i, v_i)} v_i$$

$$\frac{\mathbf{p}^T \nabla^2 F(\mathbf{x}) \mathbf{p}}{\|\mathbf{p}\|^2} \quad F(\mathbf{x}) = F(\mathbf{x}^*) + \nabla F(\mathbf{x})^T|_{\mathbf{x}=\mathbf{x}^*} (\mathbf{x} - \mathbf{x}^*) + \frac{1}{2} (\mathbf{x} - \mathbf{x}^*)^T \nabla^2 F(\mathbf{x})^T|_{\mathbf{x}=\mathbf{x}^*} (\mathbf{x} - \mathbf{x}^*) + \dots$$

$$\nabla F(\mathbf{x}) = \left[\frac{\partial}{\partial x_1} F(\mathbf{x}) \quad \frac{\partial}{\partial x_2} F(\mathbf{x}) \quad \dots \quad \frac{\partial}{\partial x_n} F(\mathbf{x}) \right]^T \quad \begin{bmatrix} \mathbf{p}_1^T \\ \mathbf{p}_2^T \\ \vdots \\ \mathbf{p}_Q^T \end{bmatrix}$$

LINEAR ALGEBRA



$$W^{new} = (1 - \gamma) W^{old} + \alpha t_q p_q^T$$

$$W^{new} = W^{old} + \alpha (t_q - a_q) p_q^T$$

$$W^{new} = W^{old} + \alpha a_q p_q^T$$

$$\begin{bmatrix} \frac{\partial}{\partial x_1^2} F(\mathbf{x}) & \frac{\partial}{\partial x_1 \partial x_2} F(\mathbf{x}) & \dots & \frac{\partial}{\partial x_1 \partial x_n} F(\mathbf{x}) \\ \frac{\partial}{\partial x_2 \partial x_1} F(\mathbf{x}) & \frac{\partial}{\partial x_2^2} F(\mathbf{x}) & \dots & \frac{\partial}{\partial x_2 \partial x_n} F(\mathbf{x}) \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial}{\partial x_n \partial x_1} F(\mathbf{x}) & \frac{\partial}{\partial x_n \partial x_2} F(\mathbf{x}) & \dots & \frac{\partial}{\partial x_n^2} F(\mathbf{x}) \end{bmatrix}$$

Regressão linear múltipla: Especificação

- Modelo para cada observação

$$\begin{aligned}y_1 &= \beta_0 + \beta_1 x_{11} + \beta_2 x_{12} + \dots \beta_p x_{1p} + \epsilon_1 \\y_2 &= \beta_0 + \beta_1 x_{21} + \beta_2 x_{22} + \dots \beta_p x_{2p} + \epsilon_2 \\&\vdots \\y_n &= \beta_0 + \beta_1 x_{n1} + \beta_2 x_{n2} + \dots \beta_p x_{np} + \epsilon_n\end{aligned}$$

- Notação matricial

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}_{n \times 1} = \begin{bmatrix} 1 & x_{11} & \dots & x_{1p} \\ 1 & x_{21} & \dots & x_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n1} & \dots & x_{np} \end{bmatrix}_{n \times p} \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_p \end{bmatrix}_{p \times 1} + \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \vdots \\ \epsilon_n \end{bmatrix}_{n \times 1}$$

- Notação mais compacta

$$\underset{n \times 1}{\mathbf{y}} = \underset{n \times p}{\mathbf{X}} \underset{p \times 1}{\boldsymbol{\beta}} + \underset{n \times 1}{\boldsymbol{\epsilon}}.$$

Regressão linear múltipla: Estimação (treinamento)

- Objetivo: encontrar o vetor $\hat{\beta}$, tal que

$$SQ(\beta) = (\mathbf{y} - \mathbf{X}\beta)^\top (\mathbf{y} - \mathbf{X}\beta),$$

seja a menor possível.

1. Passo 1: encontrar o vetor gradiente. Derivando em β , temos

$$\begin{aligned}\frac{\partial SQ(\beta)}{\partial \beta} &= \frac{\partial}{\partial \beta} (\mathbf{y} - \mathbf{X}\beta)^\top (\mathbf{y} - \mathbf{X}\beta) \\ &= \frac{\partial}{\partial \beta} ((\mathbf{y} - \mathbf{X}\beta)^\top) (\mathbf{y} - \mathbf{X}\beta) + (\mathbf{y} - \mathbf{X}\beta)^\top \frac{\partial}{\partial \beta} (\mathbf{y} - \mathbf{X}\beta) \\ &= -\mathbf{X}^\top (\mathbf{y} - \mathbf{X}\beta) + (\mathbf{y} - \mathbf{X}\beta)^\top (-\mathbf{X}) \\ &= -2\mathbf{X}^\top (\mathbf{y} - \mathbf{X}\beta).\end{aligned}$$

2. Passo 2: resolver o sistema de equações lineares

Regressão linear múltipla: Exemplo

- ▶ Conjunto de dados **Boston** disponível no pacote **MASS**.
- ▶ Cinco primeiras **covariáveis** disponíveis:
 - ▶ crim: taxa de crimes per capita.
 - ▶ zn: proporção de terrenos residenciais zoneados para lotes com mais de 25.000 pés quadrados.
 - ▶ indus: proporção de acres de negócios não varejistas por cidade.
 - ▶ chas: variável dummy de Charles River (1 se a área limita o rio; 0 caso contrário).
 - ▶ nox: concentração de óxido de nitrogênio (parte por 10 milhões).
- ▶ **Variável resposta:** **medv** valor mediano das casas ocupadas em \$1000.

- ▶ Carregando a base de dados

```
require(MASS)
```

```
## Carregando pacotes exigidos: MASS
```

```
data(Boston)  
head(Boston[, c(1:5,14)])
```

##		crim	zn	indus	chas	nox	medv
##	1	0.00632	18	2.31	0	0.538	24.0
##	2	0.02731	0	7.07	0	0.469	21.6
##	3	0.02729	0	7.07	0	0.469	34.7
##	4	0.03237	0	2.18	0	0.458	33.4
##	5	0.06905	0	2.18	0	0.458	36.2
##	6	0.02985	0	2.18	0	0.458	28.7

Regressão linear múltipla: Implementação computacional

- ▶ Matriz de delineamento.

```
X <- model.matrix(~ crim + zn + indus +  
                  chas + nox, data = Boston)
```

```
head(X)
```

```
##      (Intercept)      crim zn indus chas  nox  
## 1              1 0.00632 18  2.31    0 0.538  
## 2              1 0.02731  0  7.07    0 0.469  
## 3              1 0.02729  0  7.07    0 0.469  
## 4              1 0.03237  0  2.18    0 0.458  
## 5              1 0.06905  0  2.18    0 0.458  
## 6              1 0.02985  0  2.18    0 0.458
```

- ▶ Variável resposta

```
y <- Boston$medv
```

- ▶ Estimadores de mínimos quadrados:

$$\hat{\beta} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}.$$

- ▶ Computacionalmente: versão ingênua

```
round(solve(t(X)%*%X)%*%t(X)%*%y, 2)
```

```
##              [,1]  
## (Intercept) 29.49  
## crim       -0.22  
## zn          0.06  
## indus      -0.38  
## chas        7.03  
## nox        -5.42
```

Regressão linear múltipla: Implementação computacional

- Computacionalmente: versão menos ingênua

```
round(solve(t(X)%*%X, t(X)%*%y), 2)
```

```
##           [,1]
## (Intercept) 29.49
## crim        -0.22
## zn           0.06
## indus       -0.38
## chas         7.03
## nox         -5.42
```

- Versão "pouco mais esperta"

```
round(solve(crossprod(X), crossprod(X, y)), 2)
```

```
##           [,1]
## (Intercept) 29.49
## crim        -0.22
## zn           0.06
## indus       -0.38
## chas         7.03
## nox         -5.42
```

- Função nativa do R

```
t(round(coef(lm(medv ~ crim + zn + indus +
               chas + nox, data = Boston)), 2))
```

```
##      (Intercept)  crim   zn indus chas   nox
## [1,]      29.49 -0.22  0.06 -0.38  7.03 -5.42
```

Tempo computacional

```
library(rbenchmark)
formu <- medv ~ crim + zn + indus + chas + nox
saida <- benchmark("lm" = {
  b <- lm(formu, data = Boston)$coef
},
  "Versao ingenua" = {
    X <- model.matrix(formu, data = Boston)
    y <- Boston$medv
    b <- solve(t(X) %*% X) %*% t(X) %*% y
  },
  "linear system" = {
    X <- model.matrix(formu, data = Boston)
    y <- Boston$medv
    b <- solve(t(X) %*% X, t(X) %*% y)
  },
  "cross prod" = {
    X <- model.matrix(formu, data = Boston)
    y <- Boston$medv
    b <- solve(crossprod(X), crossprod(X, y))
  },
  replications = 1000,
  columns = c("test", "elapsed",
              "relative", "user.self"))
```

saida

##	test	elapsed	relative	user.self
## 4	cross prod	0.408	1.000	0.409
## 3	linear system	0.433	1.061	0.432
## 1	lm	0.768	1.882	0.765
## 2	Versao ingenua	0.465	1.140	0.463

Discussão

- ▶ Representação matricial é conveniente matematica e computacionalmente.
- ▶ Permite generalizar a metodologia de forma simples.
- ▶ Exige implementação computacional para operações com vetores e matrizes.
 - ▶ Soma e subtração de vetores.
 - ▶ Multiplicação de vetor por matriz.
 - ▶ Multiplicação entre matrizes.
 - ▶ Transposta de uma matriz.
 - ▶ Solução de sistemas de equações lineares.
 - ▶ Obtenção da inversa de uma matriz.
 - ▶ Diversas outras operações serão necessárias.
- ▶ Pode se tornar computacionalmente caro e até mesmo inviável.
- ▶ Diversos algoritmos estão disponível em **R**.
- ▶ Pode ser necessário usar bibliotecas **C++** para acelerar os cálculos.

Vetores e Matrizes

Operações com vetores em R

- Todas as operações com vetores são trivialmente definidas em R.

```
# Definindo vetores
```

```
x <- c(4,5,6)
```

```
y <- c(1,2,3)
```

```
# Soma
```

```
x + y
```

```
## [1] 5 7 9
```

```
# Subtração
```

```
x-y
```

```
## [1] 3 3 3
```

- Assumindo que os vetores são compatíveis.

```
# Multiplicação por escalar
```

```
alpha = 10
```

```
alpha*x
```

```
## [1] 40 50 60
```

```
alpha*y
```

```
## [1] 10 20 30
```

```
# Produto interno
```

```
x%*%y
```

```
##           [,1]  
## [1,]      32
```

Cuidado com a lei da reciclagem!!

- O **R** usa a lei da reciclagem o que pode trazer resultados inesperado.

```
# Definindo vetores de tamanhos diferentes
x <- c(4,5,6,5,6)
y <- c(1,2,3)
# Note que o 1 e 2 de y foram reciclados
# Soma
x + y

## Warning in x + y: comprimento do objeto maior não
## objeto menor

## [1] 5 7 9 6 8
```

- Cuidado com o operador `*` Multiplicação matricial é feita usando o operador especial.

```
x <- c(4,5,6)
y <- c(1,2,3)
x*y # Não é o produto escalar

## [1] 4 10 18

x%*%y # Produto escalar

##      [,1]
## [1,]    32
```


Matrizes em R

- Iniciando matrizes por colunas em R.

```
# Definindo duas matrizes
A <- matrix(c(2,5,6,-1,3,1),
            ncol = 2, nrow = 3)
B <- matrix(c(1,-5,3,3,2,7),
            ncol = 2, nrow = 3)
```

A

```
##      [,1] [,2]
## [1,]    2  -1
## [2,]    5   3
## [3,]    6   1
```

B

```
##      [,1] [,2]
## [1,]    1   3
## [2,]   -5   2
## [3,]    3   7
```

- Iniciando matrizes por linhas em R.

```
# Definindo duas matrizes
C <- matrix(c(2,5,6,-1,3,1),
            ncol = 2, nrow = 3, byrow = T)
D <- matrix(c(1,-5,3,3,2,7),
            ncol = 2, nrow = 3, byrow = T)
```

C

```
##      [,1] [,2]
## [1,]    2   5
## [2,]    6  -1
## [3,]    3   1
```

D

```
##      [,1] [,2]
## [1,]    1  -5
## [2,]    3   3
## [3,]    2   7
```

Operações com matrizes em R

- Dimensão e soma de matrizes compatíveis.

```
# Tamanho  
dim(A)
```

```
## [1] 3 2
```

```
# Soma  
A + B
```

```
##      [,1] [,2]  
## [1,]    3    2  
## [2,]    0    5  
## [3,]    9    8
```

- Subtração entre matrizes e multiplicação por escalar.

```
# Subtração  
A - B
```

```
##      [,1] [,2]  
## [1,]    1  -4  
## [2,]   10    1  
## [3,]    3  -6
```

```
# Multiplicação por escalar  
alpha = 10  
alpha*A
```

```
##      [,1] [,2]  
## [1,]   20 -10  
## [2,]   50  30  
## [3,]   60  10
```

Operações com matrizes em R

- Multiplicação: matrizes compatíveis.

```
A <- matrix(c(2,8,6,-1,3,7),3,2)
B <- matrix(c(4,-5,9,2,1,4,-3,6),2,4)
A%*%B
```

```
##      [,1] [,2] [,3] [,4]
## [1,]   13   16  -2  -12
## [2,]   17   78  20  -6
## [3,]  -11   68  34   24
```

- Multiplicação: matrizes não compatíveis

```
# B%*%A Matrices não compatíveis
A <- matrix(c(2,8,6,-1,3,7),3,2)
B <- matrix(c(4,-5,9,2,1,4,-3,6),2,4)
B%*%A
```

```
## Error in B %*% A: argumentos não compatíveis
```

Determinante, traço e inversa

► Determinante

```
# Matriz quadrada
A <- matrix(c(1,0.8,0.8,1),2,2)
# Determinante de A
det(A)
```

```
## [1] 0.36
```

► Traço

```
sum(diag(A))
```

```
## [1] 2
```

► Inversa de uma matriz.

```
# Inversa de A
inv_A <- solve(A)
inv_A
```

```
##           [,1] [,2]
## [1,]  2.777778 -2.222222
## [2,] -2.222222  2.777778
```

```
A%%inv_A # Matriz identidade
```

```
##           [,1] [,2]
## [1,]  1.000000e+00  0
## [2,] -4.440892e-16  1
```

Propriedades envolvendo operações com matrizes

► Sendo **A**, **B**, **C** e **D** compatíveis temos,

1. $\mathbf{A} + \mathbf{B} = \mathbf{B} + \mathbf{A}$.
2. $(\mathbf{A} + \mathbf{B}) + \mathbf{C} = \mathbf{A} + (\mathbf{B} + \mathbf{C})$.
3. $\alpha(\mathbf{A} + \mathbf{B}) = \alpha\mathbf{A} + \alpha\mathbf{B}$.
4. $(\alpha + \beta)\mathbf{A} = \alpha\mathbf{A} + \beta\mathbf{A}$.
5. $\alpha(\mathbf{AB}) = (\alpha\mathbf{A})\mathbf{B} = \mathbf{A}(\alpha\mathbf{B})$.
6. $\mathbf{A}(\mathbf{B} \pm \mathbf{C}) = \mathbf{AB} \pm \mathbf{AC}$.
7. $(\mathbf{A} \pm \mathbf{B})\mathbf{C} = \mathbf{AC} \pm \mathbf{BC}$.
8. $(\mathbf{A} - \mathbf{B})(\mathbf{C} - \mathbf{D}) = \mathbf{AC} - \mathbf{BC} - \mathbf{AD} + \mathbf{BD}$.

► Propriedades envolvendo transposta, inversa e multiplicação

1. Se **A** é $n \times m$ e **B** é $m \times n$, então

$$(\mathbf{AB})^\top = \mathbf{B}^\top \mathbf{A}^\top.$$

2. Se **A**, **B** e **C** são compatíveis

$$(\mathbf{ABC})^\top = \mathbf{C}^\top \mathbf{B}^\top \mathbf{A}^\top.$$

3. Se **A** é não singular, então \mathbf{A}^\top é não singular e sua inversa é dada por

$$(\mathbf{A}^\top)^{-1} = (\mathbf{A}^{-1})^\top.$$

4. Se **A** e **B** são matrizes não singulares de mesmo tamanho, então o produto **AB** é não singular e

$$(\mathbf{AB})^{-1} = \mathbf{B}^{-1} \mathbf{A}^{-1}.$$

Sistemas de equações lineares

Sistemas de equações

- ▶ Sistema com duas equações:

$$f_1(x_1, x_2) = 0$$

$$f_2(x_1, x_2) = 0.$$

- ▶ Solução consiste em encontrar \hat{x}_1 e \hat{x}_2 que satisfaça o sistema.

- ▶ Sistema com n equações

$$f_1(x_1, \dots, x_n) = 0$$

$$\vdots$$

$$f_n(x_1, \dots, x_n) = 0.$$

- ▶ Genericamente, tem-se

$$\mathbf{f}(\mathbf{x}) = \mathbf{0}.$$

- ▶ Cada equação é linear na incógnita.
- ▶ Solução analítica em geral é possível.
- ▶ Exemplo:

$$7x_1 + 3x_2 = 45$$

$$4x_1 + 5x_2 = 29.$$

- ▶ Solução analítica: $x_1 = 6$ e $x_2 = 1$.
- ▶ Resolver no quadro (tedioso!!).
- ▶ Três possíveis casos:
 1. Uma única solução (sistema não singular).
 2. Infinitas soluções (sistema singular).
 3. Nenhuma solução (sistema impossível).

Solução de sistemas de equações

- ▶ Diversos algoritmos estão disponíveis.
- ▶ Divisão geral entre métodos diretos e iterativos.
- ▶ Métodos diretos: Usam operações de linhas para transformar o sistema original em um sistema simples de resolver: triangular inferior, superior ou diagonal.
- ▶ Métodos tradicionais são o método de eliminação de Gauss e suas variações.
- ▶ Para uma descrição e implementação computacional [veja aqui](#).
- ▶ Método de eliminação de Gauss é muito usado para obter a decomposição LU e então resolver o sistema linear.
- ▶ Função `solve()` do **R** usa a decomposição LU.
- ▶ Nos métodos iterativos, as equações são colocadas em uma forma explícita onde cada incógnita é escrita em termos das demais.
- ▶ Dado um valor inicial para as incógnitas estas serão atualizadas até a convergência.
- ▶ Para uma descrição e implementação computacional [veja aqui](#).

Método de eliminação de Gauss

- Método de eliminação de Gauss consiste em manipular o sistema original usando operações de linha até obter um sistema triangular superior.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{23} & a_{33} & a_{34} \\ a_{41} & a_{24} & a_{34} & a_{44} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix} \rightarrow \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ 0 & a'_{22} & a'_{23} & a'_{24} \\ 0 & 0 & a'_{33} & a'_{34} \\ 0 & 0 & 0 & a'_{44} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} b_1 \\ b'_2 \\ b'_3 \\ b'_4 \end{bmatrix}$$

- Usar eliminação regressiva no novo sistema para obter a solução.
- Substituição regressiva:

$$x_n = \frac{b_n}{a_{nn}} \quad x_i = \frac{b_i - \sum_{j=i+1}^{j=n} a_{ij}x_j}{a_{ii}}, \quad i = n-1, n-2, \dots, 1.$$

Implementação: Eliminação de Gauss sem pivotação

- ▶ Passo 1: Obtendo uma matriz triangular superior.

```
gauss <- function(A, b) {  
  # Sistema aumentado  
  Ae <- cbind(A, b)  
  n_row <- nrow(Ae)  
  n_col <- ncol(Ae)  
  # Matriz para receber os resultados  
  SOL <- matrix(NA, n_row, n_col)  
  # Pivotação  
  #Ae <- Ae[order(Ae[,1], decreasing = TRUE),]  
  SOL[1,] <- Ae[1,]  
  pivo <- matrix(0, n_col, n_row)  
  for(j in 1:c(n_row-1)) {  
    for(i in c(j+1):c(n_row)) {  
      pivo[i,j] <- Ae[i,j]/SOL[j,j]  
      SOL[i,] <- Ae[i,] - pivo[i,j]*SOL[j,]  
      Ae[i,] <- SOL[i,]  
    }  
  }  
  return(SOL)  
}
```

- ▶ Passo 2: Substituição regressiva.

```
sub_reg <- function(SOL) {  
  n_row <- nrow(SOL)  
  n_col <- ncol(SOL)  
  A <- SOL[1:n_row, 1:n_row]  
  b <- SOL[, n_col]  
  n <- length(b)  
  x <- c()  
  x[n] <- b[n]/A[n,n]  
  for(i in (n-1):1) {  
    x[i] <- (b[i] - sum(A[i,c(i+1):n]*x[c(i+1)])) / A[i,i]  
  }  
  return(x)  
}
```

Aplicação: Eliminação de Gauss sem pivotação

- Resolva o sistema:

```
A <- matrix(c(3,2,5,2,4,3,6,3,4),3,3)
b <- c(24,23,33)
cbind(A, b)
```

```
##           b
## [1,] 3 2 6 24
## [2,] 2 4 3 23
## [3,] 5 3 4 33
```

```
## Passo1: Triangulação
```

```
S <- gauss(A, b)
S
```

```
##           [,1]           [,2]           [,3]           [,4]
## [1,]      3  2.000000e+00      6.000      24.000
## [2,]      0  2.666667e+00     -1.000       7.000
## [3,]      0 -5.551115e-17     -6.125     -6.125
```

```
# Passo 2: Substituição regressiva
```

```
sol = sub_reg(SOL = S)
sol
```

```
## [1] 4 3 1
```

```
# Verificando a solução
```

```
A%%sol
```

```
##           [,1]
## [1,]      24
## [2,]      23
## [3,]      33
```

Métodos iterativos

- Nos métodos iterativos, as equações são colocadas em uma forma explícita onde cada incógnita é escrita em termos das demais, i.e.

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 &= b_1 & x_1 &= [b_1 - (a_{12}x_2 + a_{13}x_3)]/a_{11} \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 &= b_2 & \rightarrow x_2 &= [b_2 - (a_{21}x_1 + a_{23}x_3)]/a_{22}. \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 &= b_3 & x_3 &= [b_3 - (a_{31}x_1 + a_{32}x_2)]/a_{33} \end{aligned}$$

- Dado um valor inicial para as incógnitas estas serão atualizadas até a convergência.
- Atualização: Método de Jacobi

$$x_i = \frac{1}{a_{ii}} \left[b_i - \left(\sum_{j=1; j \neq i}^{j=n} a_{ij}x_j \right) \right] \quad i = 1, \dots, n.$$

Método iterativo de Jacobi

► Implementação computacional

```
jacobi <- function(A, b, inicial, max_iter = 10, tol = 1e-04) {  
  n <- length(b)  
  x_temp <- matrix(NA, ncol = n, nrow = max_iter)  
  x_temp[1,] <- inicial  
  x <- x_temp[1,]  
  for(j in 2:max_iter) { #### Equação de atualização  
    for(i in 1:n) {  
      x_temp[j,i] <- (b[i] - sum(A[i,1:n][-i]*x[-i]))/A[i,i]  
    }  
    x <- x_temp[j,]  
    if(sum(abs(x_temp[j,] - x_temp[c(j-1),])) < tol) break #### Critério de parada  
  }  
  return(list("Solucao" = x, "Iteracoes" = x_temp))  
}
```

Método iterativo de Jacobi

- Resolva o seguinte sistema de equações lineares usando o método de Jacobi.

$$\begin{aligned}9x_1 - 2x_2 + 3x_3 + 2x_4 &= 54.5 \\2x_1 + 8x_2 - 2x_3 + 3x_4 &= -14 \\-3x_1 + 2x_2 + 11x_3 - 4x_4 &= 12.5 \\-2x_1 + 3x_2 + 2x_3 - 10x_4 &= -21\end{aligned}$$

- Computacionalmente

```
A <- matrix(c(9,2,-3,-2,-2,8,2,
              3,3,-2,11,2,2,3,-4,10),4,4)
b <- c(54.5, -14, 12.5, -21)
ss <- jacobi(A = A, b = b,
            inicial = c(0,0,0,0),
            max_iter = 15)
## Solução aproximada
ss$Solucao

## [1] 4.999502 -1.999771 2.500056 -1.000174

## Solução exata
solve(A, b)

## [1] 5.0 -2.0 2.5 -1.0
```

Métodos iterativo de Jacobi e Gauss-Seidel

- ▶ Em **R** o pacote **Rlinsolve** fornece implementações eficientes dos métodos de Jacobi e Gauss-Seidel.
- ▶ **Rlinsolve** inclui suporte para matrizes esparsas via **Matrix**.
- ▶ **Rlinsolve** é implementado em **C++** usando o pacote **Rcpp**.

```
A <- matrix(c(9,2,-3,-2,-2,8,2,3,3,-2,11,
              2,2,3,-4,10),4,4)
b <- c(54.5, -14, 12.5, -21)
## pacote extra
require(Rlinsolve)
lsolve.jacobi(A, b)$x ## Método de jacobi
```

```
##           [,1]
## [1,]  5.000012
## [2,] -1.999944
## [3,]  2.499971
## [4,] -1.000049
```

```
lsolve.gs(A, b)$x ## Método de Gauss-Seidel
```

```
##           [,1]
## [1,]  4.9999663
## [2,] -2.0000532
## [3,]  2.5000136
## [4,] -0.9999759
```

Decomposições matriciais e a obtenção da inversa

Decomposição LU

- ▶ Estamos interessados em resolver sistemas do tipo

$$\mathbf{Ax} = \mathbf{b}.$$

- ▶ Suponha que temos dois sistemas

$$\mathbf{Ax} = \mathbf{b}_1, \quad \text{e} \quad \mathbf{Ax} = \mathbf{b}_2.$$

- ▶ Cálculos do primeiro não ajudam a resolver o segundo.
- ▶ IDEAL! - Operações realizadas em \mathbf{A} fossem dissociadas das operações em \mathbf{b} .

- ▶ Essa ideia leva a obtenção das chamadas **decomposições matriciais**.
- ▶ Suponha que precisamos resolver vários sistemas do tipo

$$\mathbf{Ax} = \mathbf{b}.$$

para diferentes $\mathbf{b}'s$.

- ▶ Opção 1 - Calcular a inversa \mathbf{A}^{-1} , assim a solução

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}.$$

- ▶ Cálculo da inversa é computacionalmente ineficiente.

Algoritmo: Decomposição LU

- Decomponha (fatore) a matriz **A** em um produto de duas matrizes

$$\mathbf{A} = \mathbf{LU},$$

onde **L** é triangular inferior e **U** é triangular superior.

- Baseado na decomposição o sistema tem a forma:

$$\mathbf{LUx} = \mathbf{b}. \quad (3)$$

- Defina $\mathbf{Ux} = \mathbf{y}$.
- Substituindo acima tem-se

$$\mathbf{Ly} = \mathbf{b}. \quad (4)$$

- Solução é obtida em dois passos
 1. Resolva Eq.(4) para obter **y** usando substituição progressiva.
 2. Resolva Eq.(3) para obter **x** usando substituição regressiva.
- Para obter as matrizes L e U a usa-se o método de Gauss.

Implementação computacional: Decomposição LU

- Para descrição do algoritmo consulte [Matemática para Cientistas de Dados](#).

```
my_lu <- function(A) {  
  n_row <- nrow(A)  
  n_col <- ncol(A)  
  # Matriz para receber os resultados  
  SOL <- matrix(NA, n_row, n_col)  
  SOL[1,] <- A[1,]  
  pivo <- matrix(0, n_col, n_row)  
  for(j in 1:c(n_row-1)) {  
    for(i in c(j+1):c(n_row)) {  
      pivo[i,j] <- A[i,j]/SOL[j,j]  
      SOL[i,] <- A[i,] - pivo[i,j]*SOL[j,]  
      A[i,] <- SOL[i,]  
    }  
  }  
  diag(pivo) <- 1  
  return(list("L" = pivo, "U" = SOL))  
}
```

Aplicação: Decomposição LU

- Fazendo a decomposição.

```
A <- matrix(c(3,2,5,2,4,3,6,3,4),3,3)
b <- c(24,23,33)
LU <- my_lu(A) # Decomposição
LU
```

```
## $L
##      [,1] [,2] [,3]
## [1,] 1.0000000 0.000 0
## [2,] 0.6666667 1.000 0
## [3,] 1.6666667 -0.125 1
##
## $U
##      [,1] [,2] [,3]
## [1,] 3 2.000000e+00 6.000
## [2,] 0 2.666667e+00 -1.000
## [3,] 0 -5.551115e-17 -6.125
```

- Verificando a solução.

```
LU$L %*% LU$U # Verificando a solução
```

```
##      [,1] [,2] [,3]
## [1,] 3 2 6
## [2,] 2 4 3
## [3,] 5 3 4
```

Aplicação: Decomposição LU

- Resolvendo o sistema de equações.

```
# Passo 1: Substituição progressiva
y = forwardsolve(LU$L, b)
y
```

```
## [1] 24.000  7.000 -6.125
```

```
# Passo 2: Substituição regressiva
x = backsolve(LU$U, y)
x
```

```
## [1] 4 3 1
```

```
A%*%x # Verificando a solução
```

```
##      [,1]
## [1,]    24
## [2,]    23
## [3,]    33
```

- Função `lu()` do {Matrix} fornece a decomposição LU.

```
library(Matrix)
# Calcula mas não retorna
LU_M <- lu(A)
# Captura as matrizes L U e P
LU_M <- expand(LU_M)
# Substituição progressiva.
# NOTE MATRIZ DE PERMUTAÇÃO
y <- forwardsolve(LU_M$L, LU_M$P%*%b)
# Substituição regressiva
x = backsolve(LU_M$U, y)
x
```

```
## [1] 4 3 1
```

Obtendo a inversa via decomposição LU

- ▶ O método LU é especialmente adequado para o cálculo da inversa.
- ▶ Lembre-se que a inversa de **A** é tal que

$$\mathbf{A}\mathbf{A}^{-1} = \mathbf{I}.$$

- ▶ O procedimento de cálculo da inversa é essencialmente o mesmo da solução de um sistema de equações lineares, porém com mais incógnitas.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- ▶ Três sistemas de equações diferentes, em cada sistema, uma coluna da matriz **X** é a incógnita.

Implementação: Inversa via decomposição LU

- Função para resolver o sistema usando decomposição LU.

```
solve_lu <- function(LU, b) {  
  y <- forwardsolve(LU_M$L, LU_M$P%*%b)  
  x = backsolve(LU_M$U, y)  
  return(x)  
}
```

- Resolvendo vários sistemas

```
my_solve <- function(LU, B) {  
  n_col <- ncol(B)  
  n_row <- nrow(B)  
  inv <- matrix(NA, n_col, n_row)  
  for(i in 1:n_col) {  
    inv[,i] <- solve_lu(LU, B[,i])  
  }  
  return(inv)  
}
```

Aplicação: Inversa via decomposição LU

- Calcule a inversa de

$$\mathbf{A} = \begin{bmatrix} 3 & 2 & 6 \\ 2 & 4 & 3 \\ 5 & 3 & 4 \end{bmatrix}$$

```
A <- matrix(c(3,2,5,2,4,3,6,3,4),3,3)
I <- Diagonal(3, 1)
# Decomposição LU
LU <- my_lu(A)
# Obtendo a inversa
inv_A <- my_solve(LU = LU, B = I)
inv_A
```

```
##           [,1]      [,2]      [,3]
## [1,] -0.1428571 -0.20408163  0.36734694
## [2,] -0.1428571  0.36734694 -0.06122449
## [3,]  0.2857143 -0.02040816 -0.16326531
```


Aplicação: Inversa via decomposição LU

```
# Verificando o resultado
```

```
A*%inv_A
```

```
##      [,1]      [,2]      [,3]  
## [1,]    1 6.938894e-17 0.000000e+00  
## [2,]    0 1.000000e+00 -5.551115e-17  
## [3,]    0 -2.775558e-17 1.000000e+00
```

Decomposição de matrizes

- ▶ Uma infinidade de estratégias para decompor uma matriz em outras mais simples estão disponíveis.
- ▶ As decomposições mais usadas em estatística são:
 1. Decomposição em autovalores e autovetores (`eigen()`).
 2. Decomposição em valores singulares (`svd()`).
 3. Decomposição QR (`qr()`).
 4. Decomposição de Cholesky (`chol()`).
 5. Entre outras.
- ▶ Verifique para quais tipos de matrizes cada decomposição é adequada.

Matrizes esparsas

Matrizes esparsas

- ▶ Matrizes aparecem em todos os tipos de aplicação em ciência de dados.
- ▶ Modelos estatísticos, *machine learning*, análise de texto, análise de *cluster*, etc.
- ▶ Muitas vezes as matrizes usadas têm uma grande quantidade de zeros.
- ▶ Quando uma matriz tem uma quantidade considerável de zeros, dizemos que ela é **esparsa**, caso contrário dizemos que a matriz é **densa**.
- ▶ Todas as propriedades que vimos para matrizes em geral valem para matrizes esparsas.
- ▶ O **R** tem um conjunto de métodos altamente eficiente por meio do pacote **Matrix**.
- ▶ Saber que uma matriz é esparsa é útil pois permite:
 - ▶ Planejar formas de armazenar a matriz em memória.
 - ▶ Economizar cálculos em algoritmos numéricos (multiplicação, inversa, determinante, decomposições, etc).

Pacote {Matrix}

Methods for a wide variety of functions and operators applied to objects from these classes provide efficient access to BLAS (Basic Linear Algebra Subroutines), Lapack (dense matrix), CHOLMOD including AMD and COLAMD and Csparse (sparse matrix) routines. - {Matrix} documentation

- ▶ Operações matriciais em **R** são nativamente importadas da **lapack**.
- ▶ No entanto, o **R** não provê acesso direto a todas as funções da **lapack**.
- ▶ Por exemplo, se uma matriz for simétrica, podemos usar esse fato para acelerar os cálculos.
- ▶ O **R** usa um algoritmo genérico e portanto menos eficiente neste caso.
- ▶ O pacote {Matrix} aumenta a capacidade do **R** integrando com funções mais específicas da própria **lapack** e também de outras bibliotecas.
- ▶ No caso de matrizes esparsas o {Matrix} oferece um conjunto completo de funções para operações matriciais.
- ▶ Diversos algoritmos otimizados disponíveis em diversas bibliotes.
- ▶ Biblioteca **Csparse** é específica para matrizes esparsas.
- ▶ Diversas classes estão disponíveis para matrizes esparsas e densas.
- ▶ Para ver todas as classes disponíveis:

```
getClassDef("Matrix")
```

Matrizes esparsas e densas

- Quantidade de memória utilizada.

```
library(Matrix)
m1 <- matrix(0, nrow = 1000, ncol = 1000)
m2 <- Matrix(0, nrow = 1000, ncol = 1000,
             sparse = TRUE)
object.size(m1)
```

```
## 8000216 bytes
```

```
object.size(m2)
```

```
## 9240 bytes
```

- Matrizes esparsa e densa.

```
y <- rnorm(1000)
X_esp <- Matrix(0, ncol = 100, nrow = 1000)
for(i in 1:100) {
  X_esp[,i] <- rbinom(1000, size = 1, p = 0.1)
}
X_esp <- as(X_esp, "sparseMatrix")
X_densa <- matrix(X_esp,
                  ncol = 100, nrow = 1000)
```

- Classe das matrizes

```
class(X_densa)
```

```
## [1] "matrix" "array"
```

```
class(X_densa_Matrix)
```

```
## [1] "dgeMatrix"
## attr(,"package")
## [1] "Matrix"
```

```
class(X_esp)
```

```
## [1] "dgCMatrix"
## attr(,"package")
## [1] "Matrix"
```

Comparando matrizes esparsas e densas

► Tempo computacional

```
system.time(replicate(100, solve(t(X_densa)%*%X_densa, t(X_densa)%*%y)))
```

```
##   usuário   sistema decorrido  
##      0.772     0.040      0.812
```

```
system.time(replicate(100, solve(t(X_densa_Matrix)%*%X_densa_Matrix,  
                                t(X_densa_Matrix)%*%y)))
```

```
##   usuário   sistema decorrido  
##      0.831     0.024      0.854
```

```
system.time(replicate(100, solve(t(X_esp)%*%X_esp, t(X_esp)%*%y)))
```

```
##   usuário   sistema decorrido  
##      0.195     0.012      0.207
```

Integrando o R e o Armadillo via Rcpp

Exemplo: Multiplicação de matrizes

- **R** básico matriz densa.

```
# R básico
A <- matrix(c(1,2,3,4), 2, 2)
B <- matrix(c(5,6,7,8), 2, 2)
A%*%B
```

- {Matrix} matriz esparsa.

```
# Ineficiente!!!
A_sp <- Matrix(c(1,2,3,4), 2, 2,
               sparse = TRUE)
B_sp <- Matrix(c(5,6,7,8), 2, 2,
               sparse = TRUE)
A_sp%*%B_sp
```

- Para outras operações ver diretório **C++** e script **Algebra.R**.

- **Armadillo** matriz densa e esparsa.

```
#include <RcppArmadillo.h>
using namespace Rcpp;
using namespace arma;

// [[Rcpp::depends("RcppArmadillo")]]
// [[Rcpp::export]]
arma::mat prod_mat (arma::mat A, arma::mat B)
{
    arma::mat C = A * B ;
    return(C) ;
}
```

```
#include <RcppArmadillo.h>
using namespace Rcpp;
using namespace arma;

// [[Rcpp::depends("RcppArmadillo")]]
// [[Rcpp::export]]
arma::sp_mat prod_mat_sparse (arma::sp_mat A,
                              arma::sp_mat B)
{
    arma::sp_mat C = A * B ;
    return(C) ;
}
```

Comentários

- ▶ Só vale a pena usar algoritmos para matrizes esparsas quando a matriz for realmente esparsa! Muitos zeros!
- ▶ Matrizes com formas especiais: diagonal, triangular, bloco diagonal tem classes específicas que aumentam a eficiência computacional.
- ▶ Quanto mais específico você for mais eficiente e menos genérico!
- ▶ Em ambos os casos o `R` está acessando uma biblioteca `C` ou `Fortran` para fazer as contas.
- ▶ Podemos acessar outras bibliotecas `C++` diretamente usando o `Rcpp`.
- ▶ As bibliotecas `Armadillo` e `eigen` são o estado da arte em álgebra matricial em `C++`.
- ▶ Fácil acesso via os pacotes `RcppArmadillo` e `RcppEigen`.
- ▶ `Armadillo`): Armadillo is a high-quality linear algebra library for the C++ language, aiming towards a good balance between speed and ease of use. It provides high-level syntax and functionality deliberately similar to Matlab (TM).
- ▶ `Eigen`: Eigen is a C++ template library for linear algebra: matrices, vectors, numerical solvers and related algorithms. It supports dense and sparse matrices on integer, floating point and complex numbers, decompositions of such matrices, and solutions of linear systems.