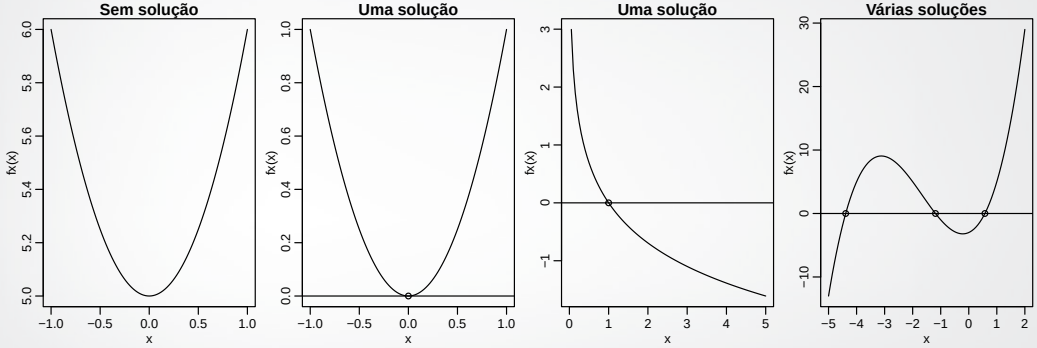
Métodos Numéricos

Prof. Wagner Hugo Bonat

Sistemas de equações não-lineares

Equações não-lineares

- Equações precisam ser resolvidas frequentemente em todas as áreas da ciência.
- Equação de uma variável: $f(x) = 0$.
- A **solução** ou **raiz** é um valor numérico de x que satisfaz a equação.



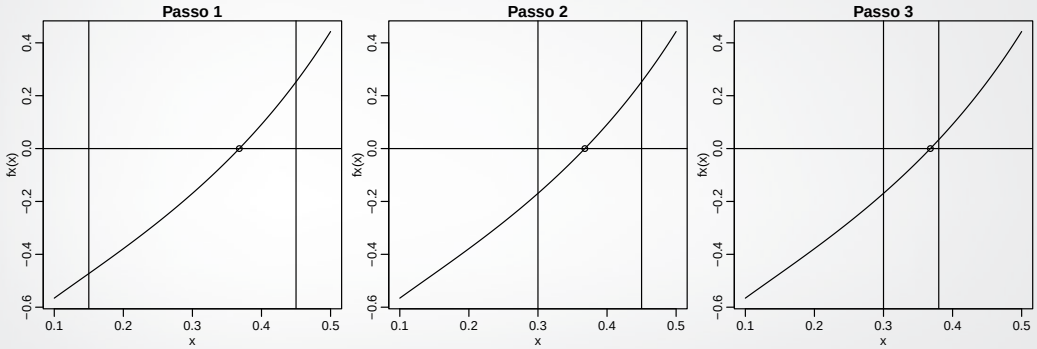
- Solução de uma equação do tipo $f(x) = 0$ é o ponto onde $f(x)$ cruza ou toca o eixo x.

Solução de equações não lineares

- Em muitas situações é impossível determinar a **raiz** analiticamente.
- Exemplo trivial $3x + 8 = 0 \rightarrow x = -\frac{8}{3}$.
- Exemplo não-trivial $8 - 4.5(x - \sin(x)) = 0 \rightarrow x = ?$
- Solução numérica de $f(x) = 0$ é um valor de x que satisfaz à equação de forma aproximada.
- Métodos numéricos para resolver equações são divididos em dois grupos:
  1. Métodos de confinamento;
  2. Métodos abertos.

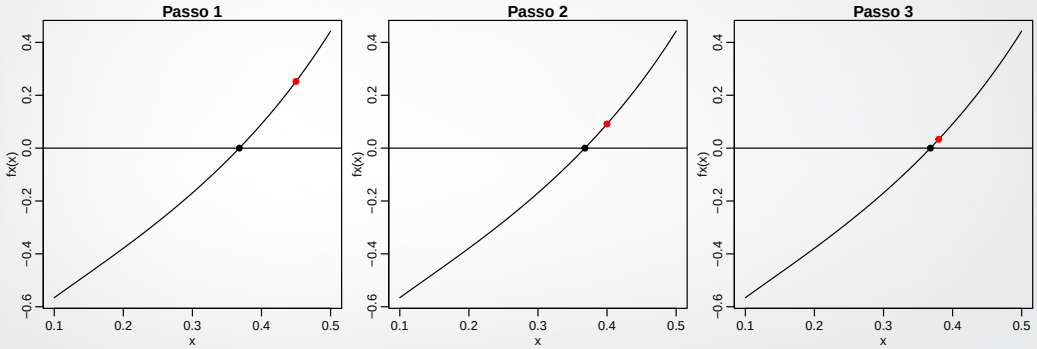
Métodos de confinamento

- Identifica-se um intervalo que possui a solução.
- Usando um esquema numérico, o tamanho do intervalo é reduzido sucessivamente até uma precisão desejada.



Métodos abertos

- Assume-se uma estimativa inicial.
- Tentativa inicial deve ser próxima a solução.
- Usando um esquema numérico a solução é melhorada.
- O processo para quando a precisão desejada é atingida.



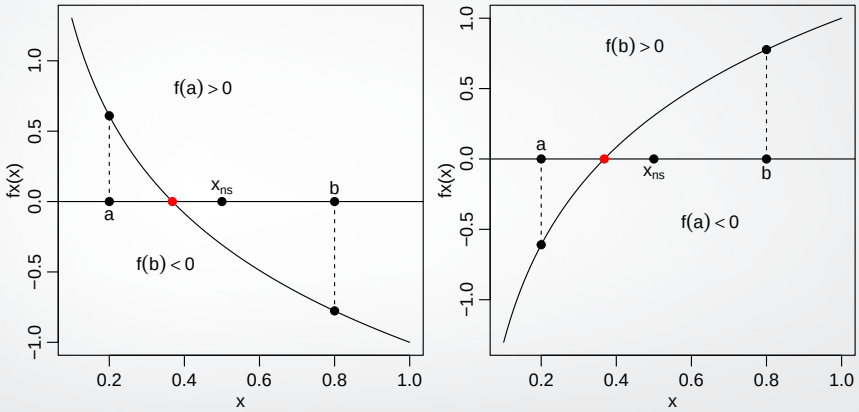
Erros em soluções numéricas

- Critério para determinar se uma solução é suficientemente precisa.
- Seja x_{ts} a solução verdadeira e x_{ns} uma solução numérica.
- Quatro medidas podem ser consideradas para avaliar o erro:
  1. Erro real $x_{ts} - x_{ns}$.
  2. Tolerância em $f(x)$
  $$|f(x_{ts}) - f(x_{ns})| = |0 - \epsilon| = |\epsilon|.$$
  3. Tolerância no tamanho do intervalo de busca:
  $$\left|\frac{b-a}{2}\right|.$$
  4. Erro relativo estimado:
  $$\left|\frac{x_{ns}^{n} - x_{ns}^{n-1}}{x_{ns}^{n-1}}\right|.$$

Métodos de confinamento

Método da bisseção

- Método de confinamento.
- Sabe-se que dentro de um intervalo $[a,b]$, $f(x)$ é contínua e possui uma solução.
- Neste caso $f(x)$ tem sinais opostos nos pontos finais do intervalo.



Algoritmo: método da bisseção

- Encontre $[a,b]$, tal que $f(a)f(b) < 0$.
- Calcule a primeira estimativa $x_{ns}^{(1)}$ usando $x_{ns}^{(1)} = \frac{a+b}{2}$.
- Determine se a solução exata está entre a e $x_{ns}^{(1)}$ ou entre $x_{ns}^{(1)}$ e b. Isso é feito verificando o sinal do produto $f(a)f(x_{ns}^{(1)})$:
  1. Se $f(a)f(x_{ns}^{(1)}) < 0$, a solução está entre a e $x_{ns}^{(1)}$.
  2. Se $f(a)f(x_{ns}^{(1)}) > 0$, a solução está entre $x_{ns}^{(1)}$ e b.
- Selecione o subintervalo que contém a solução e volte ao passo 2.
- Repita os passos 2 a 4 até que a tolerância especificada seja satisfeita.

Implementação R: método da bisseção

```
bissecao <- function(fx, a, b, tol = 1e-04, max_iter = 100) {
  fa <- fx(a);  fb <- fx(b);  if(fa*fb > 0) stop("Solução não está no intervalo")
  solucao <- c(); sol <- (a + b)/2; solucao[1] <- sol;
  limites <- matrix(NA, ncol = 2, nrow = max_iter)
  for(i in 1:max_iter) {
    test <- fx(a)*fx(sol)
    if(test < 0) {
      solucao[i+1] <- (a + sol)/2
      b = sol }
    if(test > 0) {
      solucao[i+1] <- (b + sol)/2
      a = sol }
    if( abs( (b-a)/2) < tol) break
    sol = solucao[i+1]
    limites[i,] <- c(a,b) }
  out <- list("Tentativas" = solucao, "Limites" = limites, "Raiz" = solucao[i+1])
  return(out)}
```

Exemplo

- Encontre as raízes de

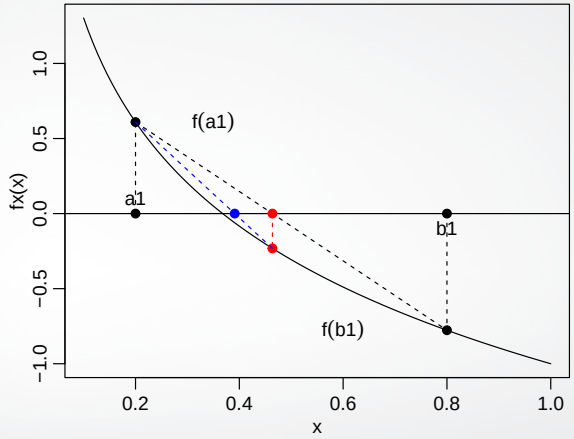
$$D(\theta) = 2n\left[\log\left(\frac{\hat{\theta}}{\theta}\right) + \bar{y}(\theta - \hat{\theta})\right] \leq 3.84.$$

```
ftheta <- function(theta){ ## Implementando a função
  dd <- 2*length(y)*(log(theta.hat/theta) + mean(y)*(theta - theta.hat))
  return(dd - 3.84)
}
set.seed(123) ## Resolvendo numericamente
y <- rexp(20, rate = 1)
theta.hat <- 1/mean(y)
Ic_min <- bissecao(fx = ftheta, a = 0, b = theta.hat)
Ic_max <- bissecao(fx = ftheta, a = theta.hat, b = 3)
c(Ic_min$Raiz, Ic_max$Raiz) ## Solução aproximada
```

```
## [1] 0.7684579 1.8545557
```

Método regula falsi

- Sabe-se que dentro de um intervalo $[a,b]$, $f(x)$ é contínua e possui uma solução.



Algoritmo: método regula falsi

- Escolha os pontos a e b entre os quais existe uma solução.
- Calcule a primeira estimativa: $x^{(i)} = \frac{af(b) - bf(a)}{f(b) - f(a)}$.
- Determine se a solução está entre a e x^i, ou entre $x^{(i)}$ e b.
  1. Se $f(a)f(x^{(i)}) < 0$, a solução está entre a e $x^{(i)}$.
  2. Se $f(a)f(x^{(i)}) > 0$, a solução está entre $x^{(i)}$ e b.
- Selecione o subintervalo que contém a solução como o novo intervalo $[a,b]$ e volte ao passo 2.
- Repita passos 2 a 4 até convergência.

Implementação R: método regula falsi

```
regula_falsi <- function(fx, a, b, tol = 1e-04, max_iter = 100) {
  fa <- fx(a);  fb <- fx(b); if(fa*fb > 0) stop("Solução não está no intervalo")
  solucao <- c() ; sol <- (a*fx(b) - b*fx(a))/(fx(b) - fx(a))
  solucao[1] <- sol;  limites <- matrix(NA, ncol = 2, nrow = max_iter)
  for(i in 1:max_iter) {
    test <- fx(a)*fx(sol)
    if(test < 0) {
      b = sol
      solucao[i+1] <- (a*fx(b) - b*fx(a))/(fx(b) - fx(a)) }
    if(test > 0) {
      a = sol
      solucao[i+1] <- sol <- (a*fx(b) - b*fx(a))/(fx(b) - fx(a)) }
    if( abs(solucao[i+1] - solucao[i]) < tol) break
    sol = solucao[i+1]
    limites[i,] <- c(a,b) }
  out <- list("Tentativas" = solucao, "Limites" = limites, "Raiz" = sol)
  return(out)}
```

Aplicação: regula-falsi

- Encontre as raízes de

$$D(\theta) = 2n \left[\log\left(\frac{\hat{\theta}}{\theta}\right) + \bar{y}(\theta - \hat{\theta}) \right] \leq 3.84.$$

```
## Resolvendo numericamente
Ic_min <- regula_falsi(fx = ftheta, a = 0.1, b = theta.hat)
Ic_max <- regula_falsi(fx = ftheta, a = theta.hat, b = 3)
## Solução aproximada
c(Ic_min$Raiz, Ic_max$Raiz)
```

```
## [1] 0.7688934 1.8545456
```

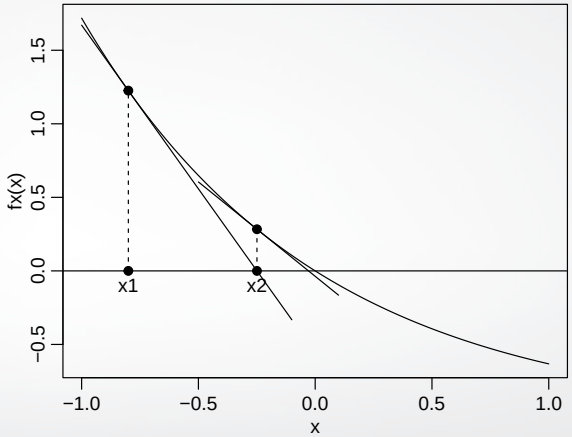
Comentários: métodos de confinamento

- Sempre convergem para uma resposta, desde que uma raiz esteja no intervalo.
- Podem falhar quando a função é tangente ao eixo x, não cruzando em $f(x) = 0$.
- Convergência é lenta em comparação com outros métodos.
- São difíceis de generalizar para sistemas de equações não-lineares.

Métodos abertos

Método de Newton

- Função deve ser contínua e diferenciável.
- Função deve possuir uma solução perto do ponto inicial.



Algoritmo: método de Newton

- Escolha um ponto x_1 como inicial.
- Para $i = 1,2,...$ até que o erro seja menor que um valor especificado, calcule

$$x^{(i+1)} = x^{(i)} - \frac{f(x^{(i)})}{f'(x^{(i)})}.$$

- Implementação computacional

```
newton <- function(fx, f_prime, x1, tol = 1e-04, max_iter = 10) {
  solucao <- c()
  solucao[1] <- x1
  for(i in 1:max_iter) {
    solucao[i+1] = solucao[i] - fx(solucao[i])/f_prime(solucao[i])
    if( abs(solucao[i+1] - solucao[i]) < tol) break
  }
  return(solucao)
}
```

Aplicação: método de Newton

- Encontre as raízes de

$$D(\theta) = 2n \left[\log\left(\frac{\hat{\theta}}{\theta}\right) + \bar{y}(\theta - \hat{\theta}) \right] \leq 3.84.$$

- Derivada

$$D'(\theta) = 2n(\bar{y} - 1/\theta).$$

```
## Derivada da função a ser resolvida
fprime <- function(theta){2*length(y)*(mean(y) - 1/theta)}
## Solução numerica
Ic_min <- newton(fx = ftheta, f_prime = fprime, x1 = 0.1)
Ic_max <- newton(fx = ftheta, f_prime = fprime, x1 = 2)
c(Ic_min[length(Ic_min)], Ic_max[length(Ic_max)])
```

```
## [1] 0.7684495 1.8545775
```

Método gradiente descendente

- Método do gradiente descendente em geral é usado para otimizar uma função.
- Suponha que desejamos minimizar $F(x)$ cuja derivada é $f(x)$.
- Sabemos que um ponto critico será obtido em $f(x) = 0$.
- Note que $f(x)$ é o gradiente de $F(x)$, assim aponta na direção de máximo.
- Assim, podemos caminhar na direção contrária seguindo o gradiente, i.e.

$$x^{(i+1)} = x^{(i)} - \alpha f(x^{(i)}).$$

- α é um parâmetro de *tuning* usado para controlar o tamanho do passo.
- A escolha do α é fundamental para atingir convergência.
- Busca em gride pode ser uma opção razoável.

Algoritmo: método gradiente descendente

- Escolha um ponto x_1 como inicial.
- Para $i = 1,2,...$ até que o erro seja menor que um valor especificado, calcule

$$x^{(i+1)} = x^{(i)} - \alpha f(x^{(i)}).$$

- Implementação computacional

```
grad_des <- function(fx, x1, alpha, max_iter = 100, tol = 1e-04) {
    sol <- c()
    sol[1] <- x1
    for(i in 1:max_iter) {
      sol[i+1] <- sol[i] - alpha*fx(sol[i])
      if(abs(fx(sol[i+1])) < tol) break
    }
    return(sol)
}
```

Aplicação: método gradiente descendente

- Encontre as raízes de

$$D(\theta) = 2n \left[\log\left(\frac{\hat{\theta}}{\theta}\right) + \bar{y}(\theta - \hat{\theta}) \right] \leq 3.84.$$

```
## Solução numerica
Ic_min <- grad_des(fx = ftheta, alpha = -0.02, x1 = 0.1)
Ic_max <- grad_des(fx = ftheta, alpha = 0.01, x1 = 4)
c(Ic_min[length(Ic_min)], Ic_max[length(Ic_max)])
```

```
## [1] 0.7684546 1.8545880
```

Sistemas de equações não-lineares

Sistemas de equações

- Sistema com duas equações:

$$\begin{aligned} f_1(x_1,x_2) &= 0 \\ f_2(x_1,x_2) &= 0. \end{aligned}$$

- A solução numérica consiste em encontrar $\hat{x}_1$ e $\hat{x}_2$ que satisfaça o sistema de equações.

Sistemas de equações

- A ideia é facilmente estendida para um sistema com n equações

$$\begin{aligned} f_1(x_1, \dots, x_n) &= 0 \\ &\vdots \\ f_n(x_1, \dots, x_n) &= 0. \end{aligned}$$

- Genericamente, tem-se

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

Método de Newton

Algoritmo: método de Newton

- Escolha um vetor x_1 como inicial.
- Para $i = 1,2,...$ até que o erro seja menor que um valor especificado, calcule

$$x^{(i+1)} = x^{(i)} - \mathrm{J}(x^{(i)})^{-1} f(x^{(i)})$$

onde

$$\mathrm{J}(x^{(i)}) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \cdots & \frac{\partial f_n}{\partial x_n} \end{bmatrix}$$

é chamado Jacobiano de f(x).

Implementação: método de Newton

- Implementação computacional

```
newton <- function(fx, jacobian, x1, tol = 1e-04, max_iter = 10) {
  solucao <- matrix(NA, ncol = length(x1), nrow = max_iter)
  solucao[1,] <- x1
  for(i in 1:max_iter) {
    J <- jacobian(solucao[i,])
    grad <- fx(solucao[i,])
    solucao[i+1,] = solucao[i,] - solve(J, grad)
    if( sum(abs(solucao[i+1,] - solucao[i,])) < tol) break
  }
  return(solucao)
}
```

Aplicação: método de Newton

- Resolva

$$\begin{aligned} f_1(x_1, x_2) &= x_2 - \frac{1}{2}(\exp^{x_1/2} + \exp^{-x/2}) = 0 \\ f_2(x_1, x_2) &= 9x_1^2 + 25x_2^2 - 225 = 0. \end{aligned}$$

- Precisamos obter o Jacobiano, assim tem-se

$$\mathrm{J}(x) = \begin{bmatrix} -\frac{1}{2}(\frac{\exp^{x1/2}}{2} - \frac{\exp^{-x1/2}}{2}) & 1 \\ 18x_1 & 50x_2 \end{bmatrix}.$$

Aplicação: método de Newton

- Resolvendo sistemas não-lineares.

```
## Sistema a ser resolvido
fx <- function(x){c(x[2] - 0.5*(exp(x[1]/2) + exp(-x[1]/2)),
                    9*x[1]^2 + 25*x[2]^2 - 225 )}
## Jacobiano
Jacobian <- function(x) {
  jac <- matrix(NA,2,2)
  jac[1,1] <- -0.5*(exp(x[1]/2)/2 - exp(-x[1]/2)/2)
  jac[1,2] <- 1
  jac[2,1] <- 18*x[1]
  jac[2,2] <- 50*x[2]
  return(jac)
}
```

Aplicação: método de Newton

- Resolvendo sistemas não-lineares.

```
## Resolvendo
sol <- newton(fx = fx, jacobian = Jacobian, x1 = c(1,1))
tail(sol,4) ## Solução
```

```
##           [,1]     [,2]
##  [7,] 3.031159 2.385865
##  [8,] 3.031155 2.385866
##  [9,]       NA       NA
## [10,]       NA       NA
```

```
fx(sol[8,]) ## OK
```

```
## [1] -3.125056e-12  9.907808e-11
```

Comentários: método de Newton

- Método de Newton irá convergir tipicamente se três condições forem satisfeitas:
  1. As funções $f_1, f_2, \ldots, f_n$ e suas derivadas forem contínuas e limitadas na vizinhança da solução.
  2. O Jacobiano deve ser diferente de zero na vizinhança da solução.
  3. A estimativa inicial de solução deve estar suficientemente próxima da solução exata.
- Derivadas parciais (elementos da matriz Jacobiana) devem ser determinados. Isso pode ser feito analitica ou numericamente.
- Cada passo do algoritmo envolve a inversão de uma matriz.

Método gradiente descendente

Método gradiente descendente

- Escolha um vetor x_1 como inicial.
- Para $i = 1,2,...$ até que o erro seja menor que um valor especificado, calcule

$$x^{(i+1)} = x^{(i)} - \alpha f(x^{(i)}).$$

- Implementação computacional

```
grad_des <- function(fx, x1, alpha, max_iter = 100, tol = 1e-04) {
solucao <- matrix(NA, ncol = length(x1), nrow = max_iter)
solucao[1,] <- x1
for(i in 1:c(max_iter-1)) {
  solucao[i+1,] <- solucao[i,] - alpha*fx(solucao[i,])
  #print(c(i, solucao[i+1,]))
  if( sum(abs(solucao[i+1,] - solucao[i,])) <= tol) break
  }
return(solucao)
}
```

Aplicação: método gradiente descendente

- Resolva

$$
\begin{aligned}
f_1(x_1, x_2) &= -2\sum_{i=1}^{10}(y_i - x_1 - x_2 z_i) \\
f_2(x_1, x_2) &= -2\sum_{i=1}^{10}(y_i - x_1 - x_2 z_i) z_i
\end{aligned}
$$

onde $y_i = (5.15; 6.40; 2.77; 5.72; 6.25; 3.45; 5.00; 6.86; 4.86; 3.72)$ e $z_i = (0.28; 0.78; 0.40; 0.88; 0.94; 0.04; 0.52; 0.89; 0.55; 0.45)$.

Aplicação: método gradiente descendente

- Implementação computacional

```
fx <- function(x) {
y <- c(5.15, 6.40, 2.77, 5.72, 6.25, 3.45, 5.00, 6.86, 4.86, 3.72)
z <- c(0.28, 0.78, 0.40, 0.88, 0.94, 0.04, 0.52, 0.89, 0.55, 0.45)
term1 <- - 2*sum(y - x[1] - x[2]*z)
term2 <- -2*sum( (y - x[1] - x[2]*z)*z)
out <- c(term1, term2)
return(out)
}
sol_grad <- grad_des(fx = fx, x1 = c(5, 0), alpha = 0.05, max_iter = 140)
fx(x = sol_grad[137,])
```

```
## [1]  0.0006924313 -0.0011375970
```

Comentários: método gradiente descendente

- Vantagem: não precisa calcular o Jacobiano!!
- Desvantagem: precisa de *tuning*.
- Em geral precisa de mais iterações que o método de Newton.
- Cada iteração é mais barata computacionalmente.
- Uma variação do método é conhecido como *steepest descent*.
- Avalia a mudança em $f(x)$ para um gride de α e da o passo usando o α que torna $F(x)$ maior/menor.
- O tamanho do passo pode ser adaptativo.
- Cuidado! Supõe que a função subjacente está sendo minimizada!

Diferenciação numérica

Diferenciação numérica

- Derivada dá uma medida da taxa na qual a variável y muda devido a uma mudança na variável x.
- A função a ser diferenciada pode ser dada por uma função $f(x)$, ou apenas por um conjunto de pontos (y_i, x_i).
- Quando devemos usar derivadas numéricas?
  1. $f'(x)$ é difícil de obter analiticamente.
  2. $f'(x)$ é caro para calcular computacionalmente.
  3. Quando a função é especificada apenas por um conjunto de pontos.
- Abordagens para a diferenciação numérica
  1. Aproximação por diferenças finitas.
  2. Aproximar a função por uma outra função de fácil derivação.
  3. Diferenciação automática (fora do nosso escopo).

Aproximação da derivada por diferenças finitas

- Derivada $f'(x)$ de uma função $f(x)$ no ponto $x = a$ é definida como:

$$f'(a) = \lim_{x \to a} \frac{f(x) - f(a)}{x - a}.$$

- Derivada é a inclinação da reta tangente à função em $x = a$.
- Escolhe-se um ponto x próximo a a e calcula-se a inclinação da reta que conecta os dois pontos.
- A precisão do cálculo aumenta quando x aproxima de a.
- Aproximação numérica: a função será avaliada em diferentes pontos próximos a a para aproximar $f'(a)$.

Aproximação da derivada por diferenças finitas

- Fórmulas para diferenciação numérica:
  1. Diferença progressiva: inclinação da reta que conecta os pontos $(x_i, f(x_i))$ e $(x_{i+1}, f(x_{i+1}))$:

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i}.$$

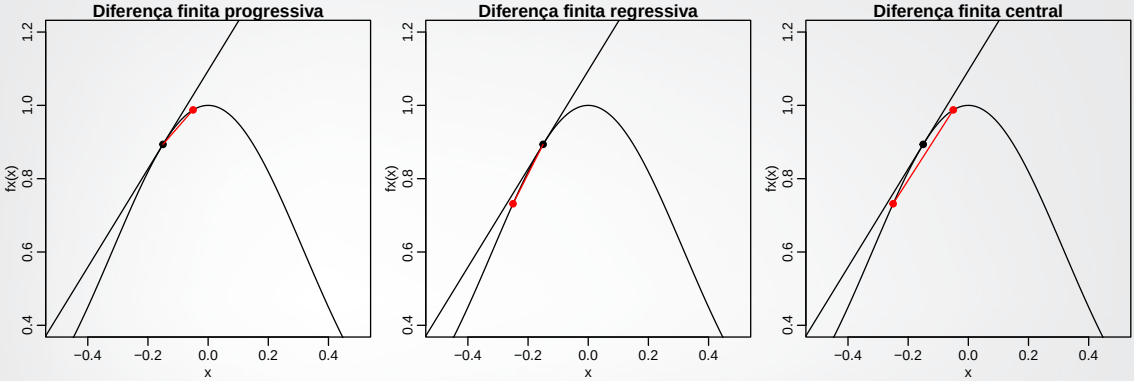
  2. Diferença regressiva: inclinação da reta que conecta os pontos $(x_{i-1}, f(x_{i-1}))$ e $(x_i, f(x_i))$:

$$f'(x_i) = \frac{f(x_i) - f(x_{i-1})}{x_i - x_{i-1}}.$$

  3. Diferença central: inclinação da reta que conecta os pontos $(x_{i-1}, f(x_{i-1}))$ e $(x_{i+1}, f(x_{i+1}))$:

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_{i-1})}{x_{i+1} - x_{i-1}}.$$

Ilustração: derivada por diferenças finitas



Aproximação da derivada por diferenças finitas

- Diferença progressiva

```
dif_prog <- function(fx, x, h) {
  df <- (fx(x + h) - fx(x))/( (x + h) - x)
  return(df)
}
```

- Diferença regressiva

```
dif_reg <- function(fx, x, h) {
  df <- (fx(x) - fx(x - h))/( x - (x - h))
  return(df)
}
```

- Diferença central

```
dif_cen <- function(fx, x, h) {
  df <- (fx(x + h) - fx(x - h))/( (x + h) - (x - h))
  return(df)}
```

Exemplo: aproximação da derivada por diferenças finitas

- Considere $f(x) = x^3$, assim $f'(x) = 3x^2$.

```
fx <- function(x) x^3
dif_prog(fx, x = 2, h = 0.001) ## Diferença progressiva
```

```
## [1] 12.006
```

```
dif_reg(fx, x = 2, h = 0.001) ## Diferença regressiva
```

```
## [1] 11.994
```

```
dif_cen(fx, x = 2, h = 0.001) ## Diferença central
```

```
## [1] 12
```

```
3*2^2 ## Exata
```

```
## [1] 12
```

Diferenças finitas usando expansão em Séries de Taylor

Diferenças finitas usando expansão em série de Taylor

- As fórmulas anteriores podem ser deduzidas usando expansão em série de Taylor.
- O número de pontos para aproximar a derivada pode mudar.
- Vantagem da dedução por série de Taylor é que ela fornece uma estimativa do erro de truncamento.

Diferença finita progressiva com dois pontos

- Aproximação de Taylor para o ponto x_{i+1}

$$f(x_{i+1}) = f(x_i) + f'(x_i)h + \frac{f''(x_i)}{2!}h^2 + \frac{f'''(x_i)}{3!}h^3 + \dots,$$

onde $h = x_{i+1} - x_i$.

- Fixando dois termos e deixando os outros termos como um resíduo, temos

$$f(x_{i+1}) = f(x_i) + f'(x_i)h + \frac{f''(\xi)}{2!}h^2.$$

- Resolvendo para $f'(x_i)$, temos

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_i)}{h} - \frac{f''(\xi)}{2!}h^2.$$

- Erro de truncamento,

$$-\frac{f''(\xi)}{2!}h^2 = O(h).$$

Diferença finita regressiva com dois pontos

- Aproximação de Taylor para o ponto x_{i-1}

$$f(x_{i-1}) = f(x_i) - f'(x_i)h + \frac{f''(x_i)}{2!}h^2 + \frac{f'''(x_i)}{3!}h^3 + \dots,$$

  onde $h = x_i - x_{i-1}$.

- Fixando dois termos e deixando os outros termos como um resíduo, temos

$$f(x_{i-1}) = f(x_i) - f'(x_i)h + \frac{f''(\xi)}{2!}h^2.$$

- Resolvendo para $f'(x_i)$, temos

$$f'(x_i) = \frac{f(x_i) - f(x_{i-1})}{h} + \frac{f''(\xi)}{2!}h^2.$$

- Erro de truncamento,

$$\frac{f''(\xi)}{2!}h^2 = O(h).$$

Diferença finita central com dois pontos

- Aproximação de Taylor para o ponto x_{i+1}

$$f(x_{i+1}) = f(x_i) + f'(x_i)h + \frac{f''(x_i)}{2!}h^2 + \frac{f'''(\xi_1)}{3!}h^3,$$

onde ξ_1 está entre x_i e x_{i+1}.

- Aproximação de Taylor para o ponto x_{i-1}

$$f(x_{i-1}) = f(x_i) - f'(x_i)h + \frac{f''(x_i)}{2!}h^2 + \frac{f'''(\xi_2)}{3!}h^3,$$

onde ξ_2 está entre x_{i-1} e x_i.

Diferença finita central com dois pontos

- Subtraindo as equações, temos

$$f(x_{i+1}) - f(x_{i-1}) = 2f'(x_i)h + \frac{f'''(\xi_1)}{3!}h^3 + \frac{f'''(\xi_2)}{3!}h^3.$$

- Resolvendo para $f'(x_i)$, temos

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_{i-1})}{2h} + O(h^2).$$

Diferença finita progressiva com três pontos

- Aproxima $f'(x_i)$ avaliando a função no ponto e nos dois pontos seguintes x_{i+1} e x_{i+2}.
- Aproximação de Taylor em x_{i+1} e x_{i+2},

$$f(x_{i+1}) = f(x_i) + f'(x_i)h + \frac{f''(x_i)}{2!}h^2 + \frac{f'''(\xi_1)}{3!}h^3, \tag{1}$$

$$f(x_{i+2}) = f(x_i) + f'(x_i)2h + \frac{f''(x_i)}{2!}(2h)^2 + \frac{f'''(\xi_2)}{3!}(2h)^3. \tag{2}$$

- Equações 1 e 2 são combinadas de forma que os termos com derivada segunda desapareçam.
- Multiplicando Eq. 1 por 4 e subtraindo Eq. 2, temos

$$4f(x_{i+1}) - f(x_{i+2}) = 3f(x_i) + 2f'(x_i)h + \frac{4f'''(\xi_1)}{3!}h^3 - \frac{f'''(\xi_2)}{3!}(2h)^3.$$

Diferença finita com três pontos

- Resolvendo em $f'(x_i)$, temos

$$f'(x_i) = \frac{-3f(x_i) + 4f(x_{i+1}) - f(x_{i+2})}{2h} + O(h).$$

- Diferença finita regressiva com três pontos

$$f'(x_i) = \frac{f(x_{i-2}) - 4f(x_{i-1}) + 3f(x_i)}{2h} + O(h).$$

Derivadas de segunda ordem

Fórmulas de diferenças finitas para a segunda derivada

- Usando as mesmas ideias podemos aproximar a derivada segunda de uma função qualquer por diferenças finitas.
- A derivação das fórmulas são idênticas, porém mais tediosas.
- Fórmula da diferença central com três pontos para a derivada segunda

$$f''(x_i) = \frac{f(x_{i-1}) - 2f(x_i) + f(x_{i+1})}{h^2} + O(h^2).$$

- Diferença central com quatro pontos

$$f''(x_i) = \frac{-f(x_{i-2}) + 16f(x_{i-1}) - 30f(x_i) + 16f(x_{i+1}) - f(x_{i+2})}{12h^2} + O(h^4).$$

Fórmulas de diferenças finitas para a segunda derivada

- Diferença progressiva com três pontos

$$f''(x_i) = \frac{f(x_i) - 2f(x_{i+1}) + f(x_{i+2})}{h^2} + O(h).$$

- Diferença regressiva com três pontos

$$f''(x_i) = \frac{f(x_{i-2}) - 2f(x_{i-1}) + f(x_i)}{h^2} + O(h).$$

- Uma infinidade de fórmulas de várias ordens estão disponíveis.
- Fórmulas de diferenciação podem ser obtidas usando polinômios de Lagrange.

Erros na diferenciação numérica

- Em todas as fórmulas o erro de truncamento é função de h.
- h é o espaçamento entre os pontos, i.e. $h = x_{i+1} - x_i$.
- Com h pequeno o erro de truncamento será pequeno.
- Em geral usa-se a precisão da máquina, algo como $1e^{-16}$.
- O erro de arredondamento depende da precisão finita de cada computador.
- Mesmo que h possa ser tão pequeno quanto desejado o erro de arredondamento pode crescer quando se diminue h.

Extrapolação de Richardson

Extrapolação de Richardson

- Extrapolação de Richardson é usada para obter uma aproximação mais precisa da derivada a partir de duas aproximações menos precisas.
- Considere o valor D de uma derivada (desconhecida) calculada pela fórmula

$$D = D(h) + k_2 h^2 + k_4 h^4, \tag{3}$$

onde $D(h)$ aproxima D e k_2 e k_4 são termos de erro.

- O uso da mesma fórmula, porém com espaçamento $h/2$ resulta

$$D = D\left(\frac{h}{2}\right) + k_2 \left(\frac{h}{2}\right)^2 + k_4 \left(\frac{h}{2}\right)^4. \tag{4}$$

Extrapolação de Richardson

- A Eq. (4) pode ser rescrita (após multiplicar por 4):

$$4D = 4D\left(\frac{h}{2}\right) + k_2 h^2 + k_4 \frac{h^4}{4}. \tag{5}$$

- Subtraindo (3) de (5) elimina os termos com h^2 e fornece

$$3D = 4D\left(\frac{h}{2}\right) + D(h) - k_4 \frac{3h^4}{4}. \tag{6}$$

- Resolvendo (6), temos

$$D = \frac{1}{3}\left(4D\left(\frac{h}{2}\right) + D(h)\right) - k_4 \frac{h^4}{4}. \tag{7}$$

Extrapolação de Richardson

- O erro na Eq. (7) é agora $O(h^4)$. O valor de D é aproximado por

$$D = \frac{1}{3}\left(4D\left(\frac{h}{2}\right) + D(h)\right) + O(h^4).$$

- A partir de duas aproximações de ordem inferiores, obtemos uma aproximação de $O(h^4)$ mais precisa.
- Procedimento a partir de duas aproximações com erro $O(h^4)$ mostra que

$$D = \frac{1}{15}\left(16D\left(\frac{h}{2}\right) + D(h)\right) + O(h^6).$$

- Aproximação ainda mais precisa.

Exemplo: extrapolação de Richardson

- Calcule a derivada de $f(x) = \frac{2^x}{x}$ no ponto $x = 2$.
- Solução exata: $\frac{\log(2)2^x}{x} - \frac{2^x}{x^2}$.
- Solução numérica usando diferença central

```
fx <- function(x) (2^x)/x
fpx <- function(x)(log(2)*(2^x))/
                    x - (2^x)/x^2
erro <- fpx(x = 2)/
          dif_cen(fx = fx, x = 2, h = 0.2)
(erro-1)*100
```

```
## [1] 0.345544
```

- Extrapolação de Richardson

```
D2 <- dif_cen(fx = fx, x = 2, h = 0.2/2)
D <- dif_cen(fx = fx, x = 2, h = 0.2)
der <- (1/3)*( 4*D2 - D)
erro2 <- fpx(x = 2)/der
(erro2-1)*100
```

```
## [1] -0.001585268
```

```
c("Exata" = fpx(x = 2), "Richardson" = der,
  "Central" = dif_cen(fx = fx, x = 2, h = 0.2))
```

```
##      Exata Richardson    Central
##  0.3862944  0.3863005  0.3849641
```

Derivadas parciais

Derivadas parciais

- Para funções com muitas variáveis, a derivada parcial da função em relação a uma das variáveis representa a taxa de variação da função em relação a essa variável, mantendo as demais constantes.
- Assim, as fórmulas de diferenças finitas podem ser usadas no cálculo das derivadas parciais.
- As fórmulas são aplicadas em cada uma das variáveis, mantendo as outras fixas.
- A mesma ideia se aplica para derivadas de mais alta ordem.

Implementação: derivadas parciais

- Derive $f(\beta_0, \beta_1) = \sum_{i=1}^{n} |y_i - (\beta_0 + \beta_1 x_i)|$.
- Fórmula dois pontos central

```
dif_cen <- function(fx, pt, h, ...) {
  df <- (fx(pt + h, ...) - fx(pt - h, ...))/( (pt + h) - (pt - h))
  return(df)
}
```

- Função a ser diferenciada

```
fx <- function(par, y, x1) {sum ( abs( y - (par[1] + par[2]*x1)) )}
```

Implementação: derivadas parciais

- Gradiente usando diferenças finita.

```
grad_fx <- function(fx, par, h, ...) {
  fbeta0 <- function(beta0, beta1, y, x) fx(par = c(beta0, beta1), y = y, x = x)
  fbeta1 <- function(beta1, beta0, y, x) fx(par = c(beta0, beta1), y = y, x = x)
  db0 <- dif_cen(fx = fbeta0, pt = par[1], h = h, beta1 = par[2], y = y, x = x)
  db1 <- dif_cen(fx = fbeta1, pt = par[2], h = h, beta0 = par[1], y = y, x = x)
  return(c(db0, db1))
}
```

Exemplo: derivadas parciais

- Simulando y_i's e x_i's.

```
set.seed(123)
x <- runif(100)
y <- rnorm(100, mean = 2 + 3*x, sd = 1)
```

- Gradiente numérico

```
grad_fx(fx = fx, par = c(2, 3), h = 0.001, y = y, x1 = x)
```

```
## [1] 6.000000 2.272805
```

- Gradiente analítico

```
c(sum(((y - 2 - 3*x)/abs(y - 2 -3*x))*(-1)),
  sum(((y - 2 - 3*x)/abs(y - 2 -3*x))*(-x)))
```

```
## [1] 6.000000 2.272805
```

Uso de funções residentes do R para diferenciação numérica

Uso de funções residentes do R para diferenciação numérica

- Pacote numDeriv implementa derivadas por diferença finita.
- Gradiente

```
require(numDeriv)
args(grad)
```

```
## function (func, x, method = "Richardson", side = NULL, method.args = list(),
##     ...)
## NULL
```

- Hessiano

```
args(hessian)
```

```
## function (func, x, method = "Richardson", method.args = list(),
##     ...)
## NULL
```

Exemplo de aplicação

- Aplicação

```
grad(func = fx, x = c(2, 3), y = y, x1 = x)
```

```
## [1] 6.000000 2.272805
```

```
hessian(func = fx, x = c(2, 3), y = y, x1 = x)
```

```
##          [,1]     [,2]
## [1,] 58.91271 29.53710
## [2,] 29.53710 48.86648
```

Integração numérica

Integração numérica

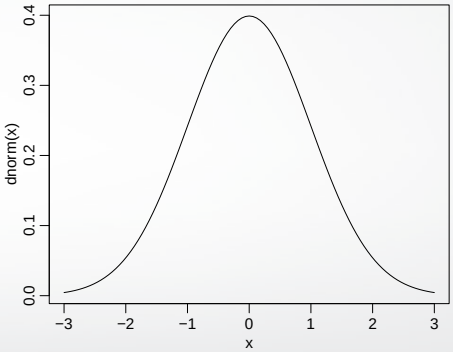
- Integrais aparecem com frequência em cálculo de probabilidades.
- A probabilidade de um evento é a área abaixo de uma curva.
- Em modelos complicados a integral pode não ter solução analítica.
- Os métodos de integração numérica, podem ser divididos em três grupos:
  1. Métodos baseados em soma finita.
  2. Aproximar a função por uma outra de fácil integração.
  3. Estimar o valor da integral.

Integração numérica

- Considere a distribuição Gaussiana com função densidade probabilidade dada por

$$f(x;\mu,\sigma^2) = \frac{1}{\sqrt{2\pi}\sigma}\exp\left(-\frac{1}{2\sigma^2}(x-\mu)^2\right).$$

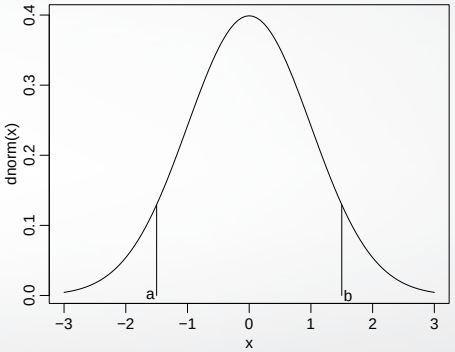
- Gráfico da função



Integração numérica

- O cálculo de uma probabilidade qualquer baseado nesta distribuição é dado pela seguinte integral

$$P(a < x < b) = \int_a^b \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{1}{2\sigma^2}(x-\mu)^2\right) dx.$$



Método Trapezoidal

Método Trapezoidal

- Usa uma função linear para aproximar o integrando.
- O integrando pode ser aproximado por Série de Taylor

$$f(x) \approx f(a) + (x-a)\left[\frac{f(b)-f(a)}{b-a}\right].$$

- Integrando analiticamente essa aproximação, tem-se

$$\begin{aligned} I(f) &\approx \int_a^b f(a) + (x-a)\left[\frac{f(b)-f(a)}{b-a}\right] dx \\ &= f(a)(b-a) + \frac{1}{2}[f(b)-f(a)](b-a). \end{aligned}$$

- Simplificando, obtem-se

$$I(f) \approx \frac{[f(a)+f(b)]}{2}(b-a).$$

Método de Simpson 1/3

Método de Simpson $1/3$

- Aproxima o integrando por um polinômio de segunda ordem.
- Pontos finais $x_1 = a$, $x_3 = b$, e o ponto central, $x_2 = (a + b)/2$.
- O polinômio pode ser escrito na forma:

$$p(x) = \alpha + \beta(x - x_1) + \lambda(x - x_1)(x - x_2) \tag{8}$$

  onde α, β e λ são constantes desconhecidas.
- Impomos a condição que o polinômio deve passar por todos os pontos, $p(x_1) = f(x_1)$, $p(x_2) = f(x_2)$ e $p(x_3) = f(x_3)$.

Método de Simpson $1/3$

- Isso resulta em:

$$\alpha = f(x_1), \quad \beta = [f(x_2) - f(x_1)]/(x_2 - x_1) \quad \text{e}$$

$$\lambda = \frac{f(x_3) - 2f(x_2) + f(x_1)}{2(h)^2}$$

onde $h = (b - a)/2$.

- Substituindo em 8 e integrando $p(x)$, obtém-se

$$I = \int_a^b f(x)dx \approx \int_a^b p(x)dx = \frac{h}{3}\left[f(a) + 4f(\frac{a+b}{2}) + f(b)\right].$$

Implementação: método de Simpson $1/3$

- A integral é facilmente calculada com apenas três avaliações da função.

```
simpson <- function(integrando, a, b, ...){
  h <- (b-a)/2
  x2 <-(a+b)/2
  integral <- (h/3)*(integrando(a,...) +
               4*integrando(x2, ...) + integrando(b, ...))
  return(integral)
}
```

- Exemplo: calcule $\int_2^3 x^2 dx$

```
fx <- function(x) x^2
simpson(integrando = fx, a = 2, b = 3)
```

```
## [1] 6.333333
```

- Solução exata: $\int_2^3 x^2 dx = \frac{x^3}{3}|_2^3 = \frac{3^3}{3} - \frac{2^3}{3} = 6.34$

Quadratura Gaussiana

Quadratura de Gauss

- Os métodos trapezoidal e Simpson são muito simples.
- Aproximam o integrando por um polinômio de fácil integração.
- Resolvem a integral aproximada.
- Os pontos são igualmente espaçados.
- São simples e intuitivos, porém de difícil generalização.
- A Quadratura Gaussiana é um dos métodos mais populares de integração numérica.
- Aplicações: modelos mistos não-lineares, análise de dados longitudinais, medidas repetidas, modelos lineares generalizados mistos, etc.

Quadratura de Gauss

- Forma geral da quadratura de Gauss:

$$\int_a^b f(x)dx \approx \sum_{i=1}^{n} C_i f(x_i), \tag{9}$$

onde C_i são pesos e x_i são os pontos de Gauss em $[a,b]$.

- Exemplo 1: para $n = 2$ a Eq. 9 tem a forma:

$$\int_a^b f(x)dx \approx C_1 f(x_1) + C_2 f(x_2).$$

- Exemplo 2: para $n = 3$ a Eq. 9 tem a forma:

$$\int_a^b f(x)dx \approx C_1 f(x_1) + C_2 f(x_2) + C_3 f(x_3).$$

Quadratura de Gauss

- Coeficientes C_i e a localização dos pontos x_i depende dos valores de n, a e b.
- C_i e x_i são determinados de forma que o lado direito da Eq. 9 seja igual ao lado esquerdo para funções $f(x)$ especificadas.
- A especificação de $f(x)$ vai depender do domínio de integração.
- Diferentes domínios levam a diferentes variações do método.

Quadratura de Gauss

- Domínios comuns:
  1. Gauss-Legendre, Gauss-Jacobi e Gauss-Chebyshev

$$\int_a^b f(x)dx.$$

  2. Gauss-Laguerre

$$\int_0^\infty f(x)e^{-x}dx.$$

  3. Gauss-Hermite

$$\int_{-\infty}^\infty f(x)e^{-x^2}dx.$$

Quadratura de Gauss

- No domínio $[-1,1]$ a forma da quadratura de Gauss é

$$\int_{-1}^{1} f(x)dx \approx \sum_{i=1}^{n} C_i f(x_i).$$

- C_i e x_i são determinados fazendo com que a Eq. 9 seja exata quando $f(x) = 1, x, x^2, x^3 \ldots$.
- O número de casos depende do valor de n.
- Para $n = 2$, tem-se

$$\int_{-1}^{1} f(x)dx \approx C_1 f(x_1) + C_2 f(x_2). \tag{10}$$

Quadratura de Gauss-Legendre

Quadratura de Gauss-Legendre

- As quatro constantes C_1, C_2, x_1 e x_2 são determinadas fazendo Eq. 10 exata quando aplicada aos quatro casos:

$$\begin{array}{lll} \text{Caso 1} & f(x) = 1 & \int_{-1}^{1} 1dx = 2 = C_1 + C_2 \\ \text{Caso 2} & f(x) = x & \int_{-1}^{1} xdx = 0 = C_1 x_1 + C_2 x_2 \\ \text{Caso 3} & f(x) = x^2 & \int_{-1}^{1} x^2 dx = \frac{2}{3} = C_1 x_1^2 + C_2 x_2^2 \\ \text{Caso 4} & f(x) = x^3 & \int_{-1}^{1} x^3 dx = 0 = C_1 x_1^3 + C_2 x_2^3 \end{array}$$

- Sistema não-linear de quatro equações e quatro incógnitas.
- Podem existir múltiplas soluções.
- Uma solução particular é obtido por impor que $x_1 = -x_2$.
- Pela equação 2, implica que $C_1 = C_2$ e a solução é

$$C_1 = 1, \quad C_2 = 1, \quad x_1 = -\frac{1}{\sqrt{3}} \quad \text{e} \quad x_2 = \frac{1}{\sqrt{3}}.$$

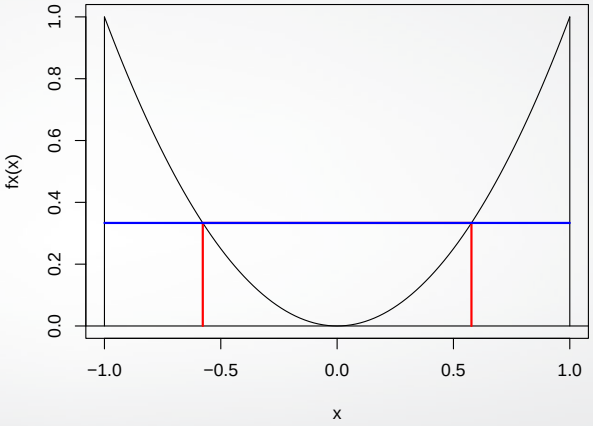
Exemplo: quadratura de Gauss-Legendre

- Calcule $\int_{-1}^{1} x^2 dx$.
- Usando Gauss-Legendre com dois pontos, tem-se

$$\int_{-1}^{1} x^2 dx = 1\left(\frac{-1}{\sqrt{3}}\right)^2 + 1\left(\frac{1}{\sqrt{3}}\right)^2 = \frac{2}{3}.$$

Exemplo: quadratura de Gauss-Legendre

- Ilustração



Exemplo: quadratura de Gauss-Legendre

- Quando $f(x)$ é uma função do tipo $f(x) = 1$, $f(x) = x$, $f(x) = x^2$ ou $f(x) = x^3$ ou qualquer combinação linear destas a aproximação é exata.
- Caso contrário o procedimento fornece uma aproximação.
- Exemplo: $f(x) = cos(x)$ valor exato é

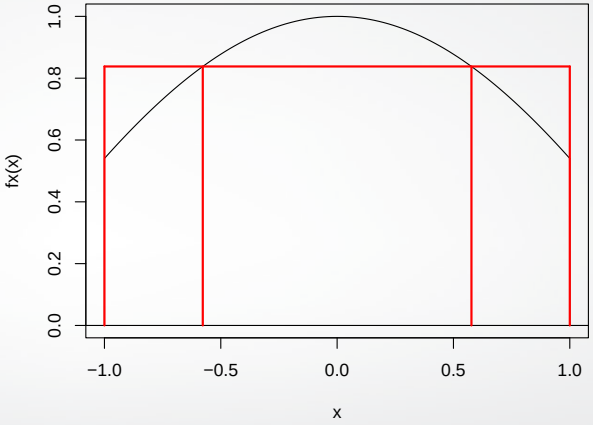
$$\int_{-1}^{1} cos(x)dx = sen(x)|_{-1}^{1} = sen(1) - sen(-1) = 1.682841.$$

- Usando Quadratura de Gauss-Legendre com $n = 2$, tem-se

$$\int_{-1}^{1} cos(x)dx \approx cos(-1/\sqrt{3}) + cos(1/\sqrt{3}) = 1.675823.$$

Exemplo: quadratura de Gauss-Legendre

- Graficamente, tem-se



Quadratura de Gauss-Legendre

- O número de pontos de integração controla a precisão da aproximação.
- Em R o pacote pracma fornece os pesos e pontos de integração.
- Exemplo

```
require(pracma)
gaussLegendre(n = 2, a = -1, b = 1)
```

```
## $x
## [1] -0.5773503  0.5773503
##
## $w
## [1] 1 1
```

- Baseado nos pontos e pesos de integração é fácil construir funções genéricas para integração numérica.

Implementação: quadratura de Gauss-Legendre

- Função genérica

```
gauss_legendre <- function(integrando, n.pontos, a, b, ...){
  pontos <- gaussLegendre(n.pontos, a = a, b = b)
  integral <- sum(pontos$w*integrando(pontos$x,...))
  return(integral)
}
```

- Exemplo: $\int_{-1}^{1} cos(x)dx$.

```
## n = 2
gauss_legendre(integrando = cos, n.pontos = 2, a = -1, b = 1)
```

```
## [1] 1.675824
```

```
## n = 10
gauss_legendre(integrando = cos, n.pontos = 10, a = -1, b = 1)
```

```
## [1] 1.682942
```

Quadratura de Gauss-Laguerre

Quadratura de Gauss-Laguerre

- Gauss-Laguerre resolve integrais do tipo:

$$\int_0^\infty e^{-x} f(x) dx.$$

- Integral é aproximada por uma soma ponderada.

$$\int_0^\infty e^{-x} f(x) dx \approx \sum_{i=1}^{n} w_i f(x_i).$$

- A função é avaliada nos pontos de Gauss e pesos de integração.
- Os pesos e pontos de integração são obtidos de forma similar ao caso de Gauss-Legendre, porém baseado no polinômio de Laguerre.

Implementação: quadratura de Gauss-Laguerre

- Função genérica para integração de Gauss-Laguerre

```
gauss_laguerre <- function(integrando, n.pontos, ...){
  pontos <- gaussLaguerre(n.pontos)
  integral <- sum(pontos$w*integrando(pontos$x,...)
                  /exp(-pontos$x))
  return(integral)
}
```

Implementação: quadratura de Gauss-Laguerre

- Exemplo: $\int_0^{\infty} \lambda \exp(-\lambda x) dx$.

```
fx <- function(x, lambda) lambda*exp(-lambda*x)
## n = 2
gauss_laguerre(integrando = fx, n.pontos = 2, lambda = 10)
```

```
## [1] 0.04381233
```

```
## n = 10
gauss_laguerre(integrando = fx, n.pontos = 10, lambda = 10)
```

```
## [1] 0.8981046
```

```
## n = 100
gauss_laguerre(integrando = fx, n.pontos = 100, lambda = 10)
```

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

Quadratura de Gauss-Hermite

Quadratura de Gauss-Hermite

- Gauss-Hermite resolve integrais do tipo:

$$\int_{-\infty}^{\infty} e^{-x^2} f(x) dx.$$

- Integral é aproximada por uma soma ponderada.

$$\int_{-\infty}^{\infty} e^{-x^2} f(x) dx \approx \sum_{i=1}^{n} w_i f(x_i).$$

- A função é avaliada nos pontos de Gauss e pesos de integração.
- Os pesos e pontos de integração são obtidos de forma similar ao caso de Gauss-Legendre, porém baseado no polinômio de Hermite.

Implementação: quadratura de Gauss-Hermite

- Função genérica para integração de Gauss-Hermite

```
gauss_hermite <- function(integrando, n.pontos, ...) {
  pontos <- gaussHermite(n.pontos)
  integral <- sum(pontos$w*integrando(pontos$x,...)
                  /exp(-pontos$x^2))
  return(integral)
}
```

Implementação: quadratura de Gauss-Hermite

- Exemplo: $\int_{-\infty}^{\infty} \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{1}{2}y^2\right) dy$.

```
## n = 2
gauss_hermite(integrando = dnorm, n.pontos = 2)
```

```
## [1] 0.9079431
```

```
## n = 10
gauss_hermite(integrando = dnorm, n.pontos = 10)
```

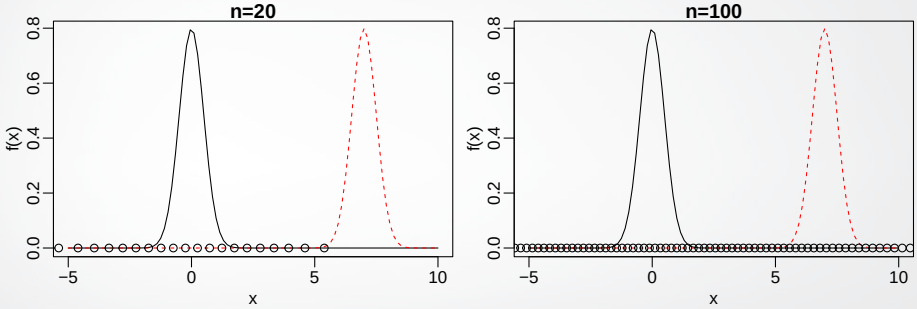
```
## [1] 0.9999876
```

```
## n = 100
gauss_hermite(integrando = dnorm, n.pontos = 100)
```

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

Limitações: quadratura de Gauss

- Quadratura de Gauss apresenta duas grandes limitações:
  1. Os pontos são escolhidos ignorando a função a ser integrada.
  2. O número de pontos necessários para a integração cresce como uma potência da dimensão da integral.
  3. 20 pontos em uma dimensão demanda $20^2 = 400$ pontos em duas dimensões.



- Espalhar os pontos de forma inteligente diminui o número de pontos necessários.

Quadratura de Gauss-Hermite Adaptativa

- Os pontos de integração são centrados e escalonados como se $f(x)e^{-x^2}$ fosse a distribuição Gaussiana.
- A média da aproximação Gaussiana será a moda $\hat{x}$ de $ln[f(x)e^{-x^2}]$.
- A variância da aproximação Gaussiana será

$$\left[-\frac{\partial^2}{\partial x^2} ln[f(x)e^{-x^2}]|_{z=\hat{z}}\right]^{-1}.$$

- Novos pontos de integração adaptados serão dados por

$$x_i^+ = \hat{x} + \left[-\frac{\partial^2}{\partial x^2} ln[f(x)e^{-x^2}]|_{x=\hat{x}}\right]^{-1/2} x_i$$

com correspondentes pesos,

$$w_i^+ = \left[-\frac{\partial^2}{\partial x^2} ln[f(x)e^{-x^2}]|_{x=\hat{x}}\right]^{-1/2} \frac{e^{x_i^+}}{e^{-x_i}} w_i.$$

Quadratura de Gauss-Hermite Adaptativa

- Como antes, a integral é aproximada por

$$\int f(x)e^{-x^2}dx \approx \sum_{i=1}^{n} w_i^+ f(x_i^+).$$

- Problema: como encontrar a moda e o hessiano de $ln[f(x)e^{-x^2}]$?
- Analiticamente ou numericamente.
- Caso especial Gauss-Hermite Adaptativa com $n = 1 \rightarrow$ Aproximação de Laplace.

Aproximação de Laplace

Aproximação de Laplace

- Denote $f(x)e^{-x^2}$ por $Q(x)$.
- Como $n = 1$, $x_1 = 0$ e $w_1 = 1$, obtemos $x_1^+ = \hat{x}$.
- Pesos de integração são iguais a

$$w_1^+ = |Q''(\hat{x})|^{-1/2} \frac{e^{-\hat{x}}}{e^{-0}} = (2\pi)^{n/2} |Q''(\hat{x})|^{-1/2} \frac{e^{Q(\hat{x})}}{f(\hat{x})}.$$

- Assim, a aproximação fica dada por

$$\begin{aligned} \int f(x)e^{-x^2} dx &= \int e^{Q(x)} dx \\ &\approx w_1^+ f(x_1^+) = (2\pi)^{n/2} |Q''(\hat{x})|^{-1/2} e^{Q(\hat{x})}. \end{aligned}$$

Implementação: aproximação de Laplace

- Função `optim()` encontra o máximo e o hessiano de $Q(x)$.

```
laplace <- function(funcao, otimizador,n.dim, ...){
  integral <- -999999
  inicial <- rep(0,n.dim)
  temp <- try(optim(inicial, funcao,..., method=otimizador,
            hessian=TRUE, control=list(fnscale=-1)))
  if(class(temp) != "try-error"){
  integral <- exp(temp$value) * (exp((n.dim/2)*log(2*pi) -
                0.5*determinant(-temp$hessian)$modulus))}
  return(integral)
}
```

Implementação: aproximação de Laplace

- Exemplo: $\int_{-\infty}^{\infty} \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{1}{2}y^2\right) dy$.

```
laplace(dnorm, otimizador = "BFGS", n.dim = 1, log = TRUE)
```

```
## [1] 1
## attr(,"logarithm")
## [1] TRUE
```

Integração Monte Carlo

Integração Monte Carlo

- Método simples e geral para resolver integrais.
- Objetivo: estimar o valor da integral de uma função $f(x)$ em algum domínio D qualquer, ou seja,

$$I = \int_D f(x)dx \tag{11}$$

- Seja $p(x)$ uma fdp cujo domínio coincide com D.
- Então, a integral em Eq. 11 é equivalente a

$$I = \int_D \frac{f(x)}{p(x)} p(x)dx.$$

- A integral corresponde a $E\left(\frac{f(x)}{p(x)}\right)$.

Algoritmo: integração Monte Carlo

- Algoritmo: integração Monte Carlo
  1. Gere números aleatórios de $p(x)$;
  2. Calcule $m_i = f(x_i)/p(x_i)$ para cada amostra, $i = 1, \ldots, n$.
  3. Calcule a média $\sum_{i=1}^{n} \frac{m_i}{n}$.
- Implementação para funções com $D = \Re$.

```
monte.carlo <- function(funcao, n.pontos, ...) {
  pontos <- rnorm(n.pontos)
  norma <- dnorm(pontos)
  integral <- mean(funcao(pontos,...)/norma)
  return(integral)
}
```

Algoritmo: integração Monte Carlo

- Exemplo: $\int_{-\infty}^{\infty} \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{1}{2}y^2\right) dy$.

```
## Integrando a Normal padrão
monte.carlo(funcao = dnorm, n.pontos = 1000)
```

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

```
## Integrando distribuição t com df = 30
monte.carlo(funcao = dt, n.pontos = 1000, df = 30)
```

```
## [1] 0.9982789
```

Função do R para integração numérica

Função do R para integração numérica

- Função `integrate()` implementa integração numérica usando quadratura adaptativa.
- O algoritmo depende do tipo da função.

```
args(integrate)
```

```
## function (f, lower, upper, ..., subdivisions = 100L, rel.tol = .Machine$double.eps^0.25,
##     abs.tol = rel.tol, stop.on.error = TRUE, keep.xy = FALSE,
##     aux = NULL)
## NULL
```

Função do R para integração numérica

- Exemplo 1: $\int_{-\infty}^{\infty} \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{1}{2}y^2\right) dy$.

```
integrate(f = dnorm, lower = -Inf, upper = Inf)
```

```
## 1 with absolute error < 9.4e-05
```

- Exemplo 2: $\int_{-1}^{1} x^2 dx$.

```
fx <- function(x)x^2
integrate(f = fx, lower = -1, upper = 1)
```

```
## 0.6666667 with absolute error < 7.4e-15
```

Discussão

- Integração numérica aparece com frequência em modelos mistos não Gaussianos.
- Método de Gauss-Hermite (GH) é muito popular.
- GH é limitado a integrais de baixa dimensão $n < 10$.
- GH é computacionalmente caro.
- GH adaptativo é mais eficiente, porém ainda limitado.
- Aproximação de Laplace é excelente para integrando simétrico.
- Laplace resolve problemas em alta dimensão.
- Pode ser inacurada para integrando assimétricos.
- Integração Monte Carlo depende da escolha da *proposal*.
- Computacionalmente intensivo.
- Resolve problemas de alta dimensão a um alto custo computacional.

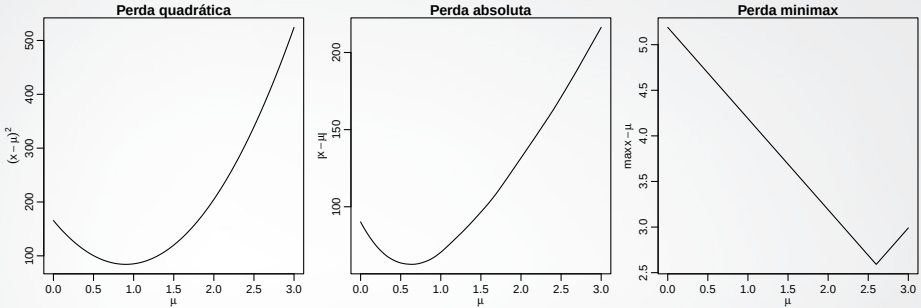
Otimização

Motivação

- Otimização usa um modelo matemático rigoroso para determinar a solução mais eficiente para um dado problema.
- Precisamos identificar um objetivo.
- Criar uma medida que mensure a performance. Ex. rendimento, tempo, custo, etc.
- Em geral, qualquer quantidade ou combinação de quantidades representada por um simples número.
- Funções perda usuais.
  - Perda quadrática: $\sum_{i=1}^{n}(y_i - \mu)^2$.
  - Perda absoluta: $\sum_{i=1}^{n}|y_i - \mu|$.
  - Perda minimax: minimize $max(|y_i - \mu|)$.

Otimização: funções perda

- Graficamente, tem-se



- Objetivo: encontrar o ponto de mínimo da função perda.

Classificação dos problemas de otimização

- Programação linear (LP)
  - Função objetivo e as restrições são lineares.
  - $\min_x \quad c^\top x$, sujeito a $Ax \leq b$ e $x \geq 0$.
- Programação quadrática (QP)
  - Função objetivo é quadrática e as restrições são lineares.
  - $\min_x \quad x^\top Q x + c^\top x$, sujeito a $Ax \leq b$ e $x \geq 0$.
- Programação não-linear (NLP): função objetivo ou ao menos uma restrição é não linear.
- Cada classe de problemas tem seus próprios métodos de solução.
- Em `R` temos pacotes específicos para cada tipo de problema.
- Frequentemente, também distinguimos se o problema tem ou não restrições.
  - Otimização restrita refere-se a problemas com restrições de igualdade ou desigualdades.

Otimização em R

- Pacotes populares para otimização em R.

Tipo de problema	Pacote	Função
Propósito geral (1 dim)	Built in	optimize(...)
Propósito geral (n dim)	Built in	optim(...)
Programação Linear	lpSolve	lp(...)
Programação quadrática	quadprog	solve.QP(...)
Programação não-linear	optimize	optimize()
	optimx	optimx(...)

- Existe uma infinidade de pacotes com os mais diversos algoritmos implementados em R.
- Todos estão listados no Task View - Optimization and Mathematical programming

Otimização em R

- A estrutura básica de um otimizador é sempre a mesma.

```
optimizer(objective, constraints, bounds = NULL, types = NULL, maximum = FALSE)
```

- As funções em geral apresentam algum argumento que permite trocar o algoritmo de otimização.
- Funções nativas do R:
  - `optimize()` restrita a problemas unidimensionais.
  - Baseado no esquema *Golden section search*.
  - `optim()` problemas n-dimensionais.
  - Restrita a funções com argumentos contínuous.

Otimizando funções perda: redução de dados

- Considere as funções perda:
  - Perda quadrática: $\sum_{i=1}^{n}(y_i - \mu)^2$.
  - Perda absoluta: $\sum_{i=1}^{n}|y_i - \mu|$.
  - Perda minimax: Minimize $max(|y_i - \mu|)$.
- Seja um conjunto de observações y_i.
- Encontre o melhor resumo de um número baseado em cada uma das funções perda anteriores.

Otimizando funções perda: redução de dados

- Passo 1: implementar as funções objetivo.
  - Perda quadrática

```
perda_quad <- function(mu, dd) { sum((dd-mu)^2) }
```

  - Perda absoluta

```
perda_abs <- function(mu, dd) { sum(abs(dd-mu)) }
```

  - Perda minimax

```
perda_minimax <- function(mu, dd) { max(abs(dd-mu)) }
```

- Passo 2: obter o conjunto de observações.

```
set.seed(123)
y <- rpois(100, lambda = 3)
```

Otimizando funções perda: redução de dados

- Passo 3: otimizando a função perda.

```
# Perda quadrática
fit_quad <- optimize(f = perda_quad, interval = c(0, 20), dd = y)
# Perda absoluta
fit_abs <- optimize(f = perda_abs, interval = c(0, 20), dd = y)
# Perda minimax
fit_minimax <- optimize(f = perda_minimax, interval = c(0, 20), dd = y)
```

Otimizando funções perda: redução de dados

- Perda quadrática

```
fit_quad
```

```
## $minimum
## [1] 2.94
##
## $objective
## [1] 259.64
```

- Perda absoluta

```
fit_abs
```

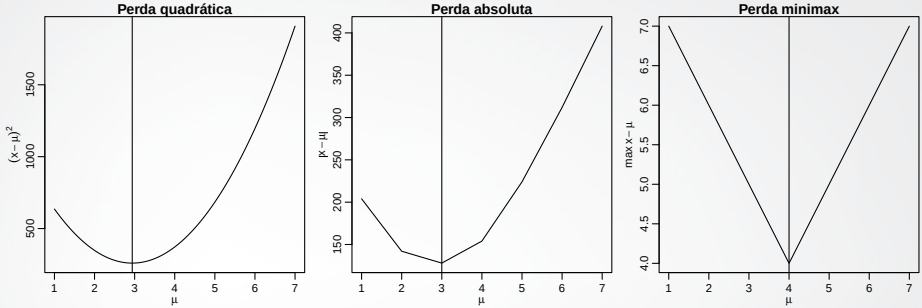
```
## $minimum
## [1] 2.999952
##
## $objective
## [1] 128.0007
```

- Perda minimax

```
fit_minimax
```

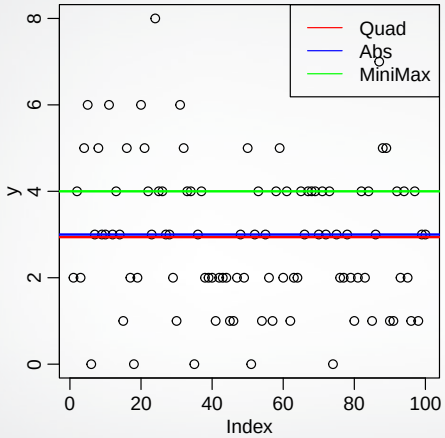
```
## $minimum
## [1] 4.000013
##
## $objective
## [1] 4.000013
```

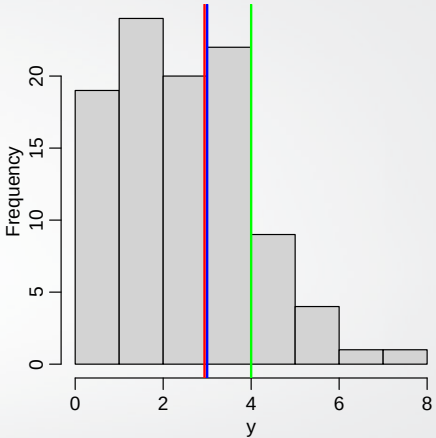
Otimizando funções perda: redução de dados



Otimizando funções perda: redução de dados

- Outra forma de visualizar.





Otimização numérica

- Muito fácil usar o otimizador numérico.
- Não precisamos calcular nada.
- Solução para quem não gosta de matemática?
- Como isso é possível?
- O que vocês acham?
- Vamos investigar caso a caso.

Programação linear

Programação linear

- Especificação matemática
- Notação matricial.

$$\min_x \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix}^\top \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \quad \text{s.t} \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \ddots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \geq \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}, \quad \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \geq 0$$

- Notação mais compacta.

$$\begin{aligned} \min_x c^\top x \quad &= \quad \min_x \quad c_1 x_1 + c_2 x_2 + \dots + c_n x_n \\ &\text{s.t} \qquad Ax \geq b, x \geq 0 \end{aligned}$$

Exemplo: programação linear

- Função objetivo
  - Objetivo: maximimizar o lucro total.
  - Produtos A e B são vendidos por R$ 25 e R$ 20.
- Restrição de recursos
  - Produto A precisa de 20 u.p e produto B precisa 12 u.p.
  - Apenas 1800 u.p estão disponíveis por dia.
- Restrição de tempo
  - Produtos A e B demoram 1/15 hrs para produzir.
  - Um dia de trabalho tem 8 hrs.

Exemplo: programação linear

- Formulação do problema
  - Denote x_1 e x_2 como número de itens A e B produzidos.
  - **Função objetivo**: maximizar o total de vendas

$$\max_{x_1,x_2} \quad 25x_1 + 20x_2.$$

  - Sujeito a restrições de recursos e tempo.

$$\begin{aligned} 20x_1 + 12x_2 &\leq 1800 \\ \tfrac{1}{15}x_1 + \tfrac{1}{15}x_2 &\leq 8 \end{aligned}$$

- Restrições escritas de forma matricial.

$$\underbrace{\begin{bmatrix} 20 & 12 \\ \frac{1}{15} & \frac{1}{15} \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}}_{x} \leq \underbrace{\begin{bmatrix} 1800 \\ 8 \end{bmatrix}}_{b}.$$

Exemplo: programação linear

- Solução força bruta !!

```
x1 <- 0:140
x2 <- 0:140
grid <- expand.grid(x1,x2)
lucro <- function(x) 25*x[1] + 20*x[2]
```

- Restrição de recursos.

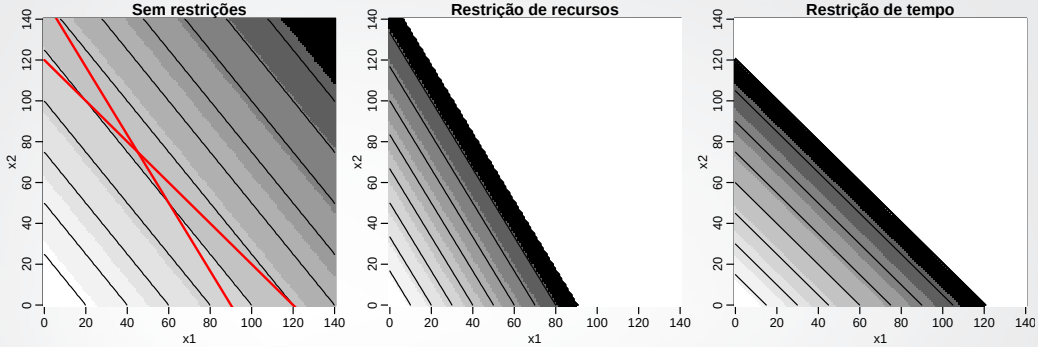
```
recurso <- function(x) {
  out <- 20*x[1] + 12*x[2]
  if(out > 1800) out = 0
  return(out)
}
```

- Restrição de tempo.

```
tempo <- function(x) {
  out <- (1/15)*x[1] + (1/15)*x[2]
  if(out > 8) out = 0
  return(out)
}
```

Exemplo: programação linear

- Graficamente, tem-se



- A ideia pode ser generalizada para n restrições.
- Algoritmo Simplex.
- Pacote `lpSolve` em `R`.

Exemplo: programação linear

- Função `lp(...)` do pacote `lpSolve`.
- Sintaxe geral

```
require(lpSolve)
lp(direction = "min", objective.in, const.mat, const.dir, const.rhs)
```

Exemplo: programação linear

- Para o nosso exemplo, tem-se

```
require(lpSolve)
```

```
## Carregando pacotes exigidos: lpSolve
```

```
objective.in <- c(25, 20) # c's
const.mat <- matrix(c(20, 12, 1/15, 1/15), nrow=2, byrow=TRUE)
const.rhs <- c(1800, 8)
const.dir <- c("<=", "<=")
optimum <- lp(direction = "max", objective.in, const.mat,
              const.dir, const.rhs)
```

Exemplo: programação linear

- Solução

```
optimum$solution # Solução
```

```
## [1] 45 75
```

```
optimum$objval # Lucro
```

```
## [1] 2625
```

Programação quadrática

Programação quadrática

- Especificação matemática

$$\min_x \frac{1}{2} x^\top \mathrm{D} x - d^\top x, \text{s.t} \quad \mathrm{A} x \geq b.$$

- Coeficientes quadráticos D $\rightarrow$ **Dmat**.
- Coeficientes lineares $d \rightarrow$ **dvec**.
- Matriz de constantes A $\rightarrow$ **Amat**.
- Restrições de igualdade ou desigualdade $b \rightarrow$ **bvec**.

Programação quadrática: implementação

- Função `solve.QP(...)` pacote quadprog.
- Argumento `meq = n` fixa as primeiras n restrições como lineares.

```
require(quadprog)
```

```
## Carregando pacotes exigidos: quadprog
```

```
args(solve.QP)
```

```
## function (Dmat, dvec, Amat, bvec, meq = 0, factorized = FALSE)
## NULL
```

Programação quadrática: regressão com restrição

- Popular na literatura de *Machine Learning*.
- Caso particular das máquinas de vetores de suporte.
- O problema de regressão é resolvido por minimizar a perda quadrática sob uma restrição.
- Por exemplo, tem-se

$$\min_{\beta} (y - \mathrm{X}\beta)^{\top}(y - \mathrm{X}\beta), \quad \text{sujeito a} \quad \sum f(\beta) \leq s.$$

- A parte da regressão pode ser escrita como

$$\min_{\beta} \quad y^{\top}y - 2y^{\top}\mathrm{X}\beta + \beta^{\top}\mathrm{X}^{\top}\mathrm{X}\beta.$$

Programação quadrática: regressão com restrição

- Neste caso, tem-se

$$\mathrm{D} = \mathrm{X}^\top \mathrm{X} \quad \text{e} \quad d = y^\top \mathrm{X}.$$

```
n <- 100
x1 <- runif(n)
x2 <- runif(n)
y <- 0 + x1 + x2 + rnorm(n)
X <- cbind( rep(1,n), x1, x2 )
# Regression
r <- lm(y ~ x1 + x2)
```

- Exemplo sem restrições.

```
# Optimization
library(quadprog)
# Sem restrição apenas como exemplo
Amat =  matrix(nr=3,nc=0)
b = numeric()
s <- solve.QP(t(X) %*% X, t(y) %*% X,
              Amat = Amat, bvec = b, meq = 0)
# Comparison
coef(r)
```

```
## (Intercept)          x1          x2
##  -0.4138375   1.6987259   1.1433559
```

```
s$solution
```

```
## [1] -0.4138375  1.6987259  1.1433559
```

Programação quadrática: regressão com restrição

- Exemplo com restrições (soma igual a 1);

```
# Optimization
# Soma dos betas = 1
Amat <- matrix(c(1,1,1),ncol = 1, nrow = 3)
b = c(1)
s <- solve.QP(t(X) %*% X, t(y) %*% X,
              Amat = Amat, bvec = b, meq = 1)
# Comparison
s$solution
```

```
## [1]  0.67483269  0.34432273 -0.01915542
```

```
sum(s$solution)
```

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

- Exemplo com restrições (soma ponderada).

```
# Optimization
# Soma dos betas = 1
Amat <- matrix(c(0.4,0.3,0.2),
               ncol = 1, nrow = 3)
b = c(1)
s <- solve.QP( t(X) %*% X, t(y) %*% X,
               Amat = Amat, bvec = b, meq = 1)
# Comparison
s$solution
```

```
## [1] -1.014036  3.732834  1.428822
```

```
sum(s$solution*c(0.4,0.3,0.2))
```

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

Programação não-linear

Métodos de programação não-linear

- Os métodos são em geral categorizados baseado na dimensionalidade
  1. Unidimensional: Golden Section search.
  2. Multidimensional.
- Caso multidimensional, tem-se pelo menos quatro tipos de algoritmos:
  1. Não baseados em gradiente: Nelder-Mead;
  2. Baseados em gradiente: gradiente descendente e variações;
  3. Baseados em hessiano: Newton e quasi-Newton (BFGS);
  4. Algoritmos baseados em simulação e ideias genéticas: Simulating Annealing (SANN).
- A função genérica `optim()` em `R` fornece interface aos principais algoritmos de otimização.
- Vamos discutir as principais ideias por trás de cada tipo de algoritmo.
- Existe uma infinidade de variações e implementações.

Método Golden Section Search

Programação não-linear: problemas unidimensionais

- O Golden Section Search é o mais popular e muito eficiente.
- Algoritmo
  1. Defina a razão de ouro $\psi = \frac{\sqrt{5}-1}{2} = 0.618$;
  2. Escolha um intervalo $[a,b]$ que contenha a solução;
  3. Avalie $f(x_1)$ onde $x_1 = a + (1-\psi)(b-a)$ e compare com $f(x_2)$ onde $x_2 = a + \psi(b-a)$;
  4. Se $f(x_1) < f(x_2)$ continue a procura em $[a,x_1]$ caso contrário em $[x_2,b]$.
- Em `R` a função `optimize()` implementa este método.

```
args(optimize)
```

```
## function (f, interval, ..., lower = min(interval), upper = max(interval),
##     maximum = FALSE, tol = .Machine$double.eps^0.25)
## NULL
```

- Na função `optim()` esse método é chamado de `Brent`.

Exemplo: otimização unidimensional

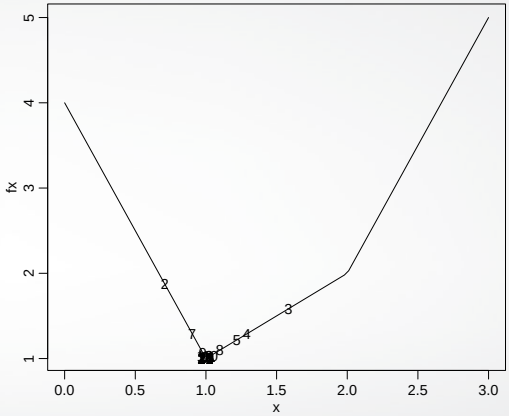
- Minize a função $f(x) = |x - 2| + 2|x - 1|$.
  - Implementando e otimizando.

```
xx <- c()
fx <- function(x) {
  out <- abs(x-2) + 2*abs(x-1)
  xx <<- c(xx, x)
  return(out)
}
out <- optimize(f = fx, interval = c(-3,3))
out
```

```
## $minimum
## [1] 1.000021
##
## $objective
## [1] 1.000021
```

Exemplo: otimização unidimensional

- Traço do algoritmo.



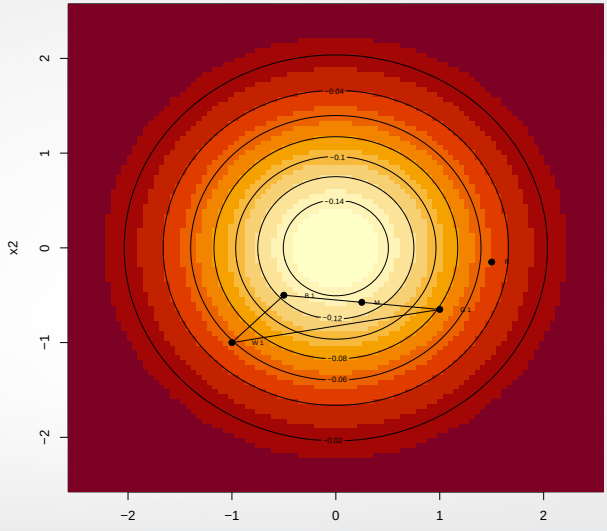
Método de Nelder-Mead (gradient free)

Método de Nelder-Mead (gradient free)

- Algoritmo de Nelder-Mead

1. Escolha um simplex com $n+1$ pontos $p_1(x_1, y_1), ... p_{n+1}(x_{n+1}, y_{n+1})$, sendo n o número de parâmetros.
2. Calcule $f(p_i)$ e ordene por tamanho $f(p_1) \leq ... f(p_n)$.
3. Avalie se o melhor valor é bom o suficiente, se for, pare.
4. Delete o ponto com maior/menor $f(p_i)$ do simplex.
5. Escolha um novo ponto pro simplex.
6. Volte ao passo 2.

Algoritmo de Nelder-Mead: ilustração



Algoritmo de Nelder-Mead: escolhendo o novo ponto

- Ponto central do lado melhor (B):

$$M = \frac{B+G}{2} = \left(\frac{x_1+x_2}{2}, \frac{y_1+y_2}{2}\right).$$

  - Refletir o simplex para o lado BG.

$$R = M + (M - W) = 2M - W.$$

  - Se a função em R é menor que em W movemos na direção correta.

1. Opção 1: faça W = R e repita.
2. Opção 2: expandir usando o ponto $E = 2R - M$ e $W = E$, repita.

- Se a função em R e W são iguais contraia W para próximo a B, repita.
- A cada passo uma decisão lógica precisa ser tomada.

Algoritmo de Nelder-Mead: ilustração

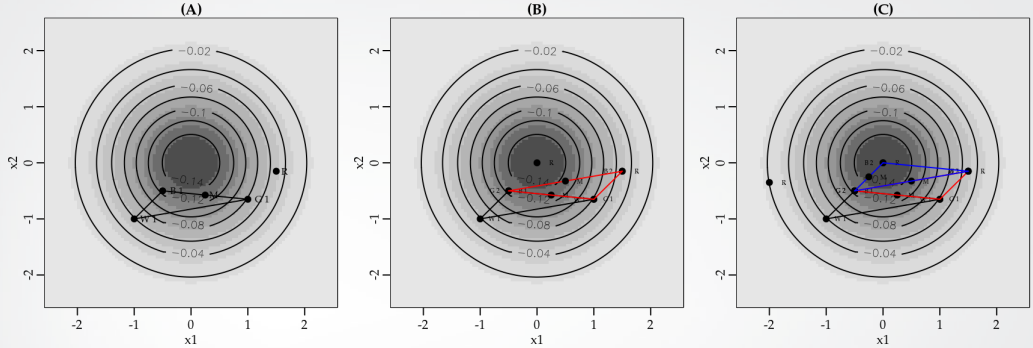
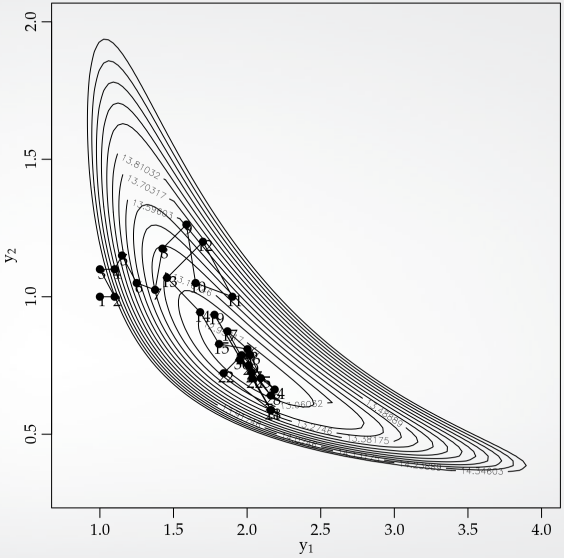


Figura 1. Ilustração método de Nelder-Mead.

Algoritmo de Nelder-Mead: ilustração



Métodos baseado em gradiente

Métodos baseado em gradiente

- Use o gradiente de $f(x)$, ou seja, $f'(x)$ para obter a direção de procura.
  1. $f'(x)$ pode ser obtido analiticamente;
  2. $f'(x)$ qualquer aproximação númerica.
- A direção de procura s_n é o negativo do gradiente no último ponto.
- Passos básicos
  1. Calcule a direção de busca $-f'(x)$.
  2. Obtenha o próximo passo $x^{(n+1)}$ movendo com passo α_n na direção de $-f'(x)$.
  3. Tamanho do passo α_n pode ser fixo ou variável.
  4. Repita até $f'(x^i) \approx 0$ seja satisfeito.

Ilustração: métodos baseado em gradiente

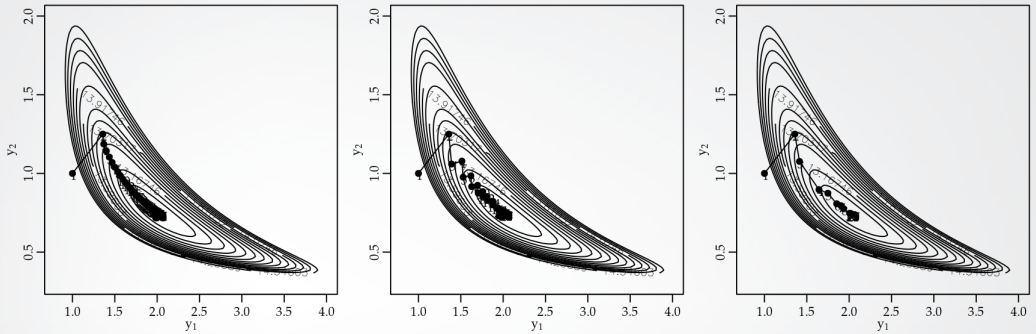


Figura 3. Ilustração método gradiente descendente com diferente estratégias de tuning.

Métodos baseado em hessiano

Métodos baseado em hessiano

- Algoritmo de Newton-Raphson.
- Maximizar/minimizar uma função $f(x)$ é o mesmo que resolver a equação não-linear $f'(x) = 0$.
- Equação de iteração

$$x^{(i+1)} = x^{(i)} - \mathrm{J}(\mathrm{x}^{(\mathrm{i})})^{-1} f'(x^{(i)}),$$

  onde J é a segunda derivada (hessiano) de $f(x)$.
- $\mathrm{J}(\mathrm{x}^{(\mathrm{i})})$ pode ser obtida analitica ou numericamente.
- $\mathrm{J}(\mathrm{x}^{(\mathrm{i})})$ pode ser aproximada por uma função mais simples de calcular.
- Métodos Quasi-Newton (mais famoso BFGS).

Métodos Quasi-Newton

- Métodos quasi-Newton tentam imitar o método de Newton.
- A equação de iteração é dada por

$$x^{(i+1)} = x_{(i)} - \alpha_i \mathrm{H}_i f'(x^{(i)}),$$

  onde H_i é alguma aproximação para o inverso do Hessiano.
- α_i é o tamanho do passo.
- Denote $\delta_i = x^{(i+1)} - x^{(i)}$ e $\gamma_i = f'(x^{(i+1)}) - f'(x^{(i)})$.
- Para obter H_{i+1} o algoritmo impõe que

$$\mathrm{H}_{i+1}\gamma_i = \delta_i.$$

- Algoritmo DFP

$$\mathrm{H}_{i+1} = \mathrm{H}_i - \frac{\mathrm{H}_i\gamma_i\gamma_i^\top\mathrm{H}}{\gamma_i^\top\mathrm{H}_i\gamma_i} + \frac{\delta_i\delta_i^\top}{\delta_i^\top\gamma_i}.$$

Métodos Quasi-Newton

- Versão melhorada do DFP devido a Broyden, Fletcher, Goldfarb e Shanno (BFGS).
- Aproxima o hessiano por

$$\mathrm{H}_{i+1} = \mathrm{H}_i - \frac{\delta_i \gamma_i^\top \mathrm{H}_i + \mathrm{H}_i \gamma_i \delta_i^\top}{\delta_i^\top \gamma_i} + \left(1 + \frac{\gamma_i^\top \mathrm{H}_i \gamma_i}{\delta_i^\top \gamma_i}\right) \left(\frac{\delta_i \delta_i^\top}{\delta_i^\top \gamma_i}\right).$$

- Implementações modernas do BFGS usando *wolfe line search* para encontrar α_i.
- Considere $\psi(\alpha) = f(x^{(i)} - \alpha \mathrm{H}_i f'(x^{(i)}))$, encontre α_i tal que

$$\psi_i(\alpha_i) \leq \psi_i(0) + \mu \psi_i'(0)\alpha_i \quad \text{e} \quad \psi_i'(\alpha_i) \geq \eta \psi_i'(0),$$

onde μ e η são constantes com $0 < \mu \leq \eta < 1$.

Ilustração: métodos baseado em hessiano (BFGS)

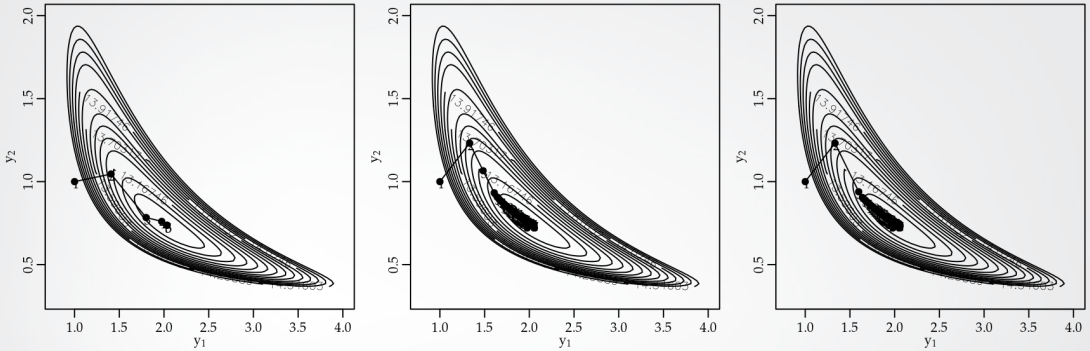


Figura 4. Ilustração métodos Newton, DFP e BFGS.

Métodos baseados em simulação

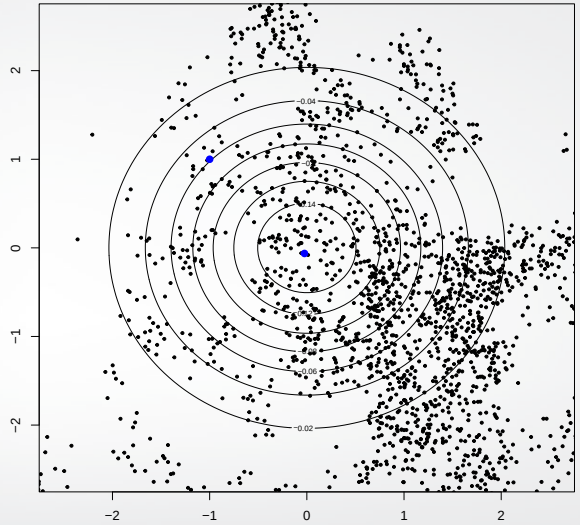
Métodos baseados em simulação

- Algoritmo genérico (maximização):
  1. Gere uma solução aleatória (x_1);
  2. Calcule a função objetivo no ponto simulado $f(x_1)$;
  3. Gere uma solução na vizinhança (x_2) do ponto em (1);
  4. Calcule a função objetivo no novo ponto $f(x_2)$:
  - Se $f(x_2) > f(x_1)$ mova para x_2.
  - Se $f(x_2) < f(x_1)$ TALVEZ mova para x_2.
  5. Repita passos 3-4 até atingir algum critério de convergência ou número máximo de iterações.

Métodos baseado em simulação: Simulating Annealing

- Para decidir se um ponto x_2 quando $f(x_2) < f(x_1)$ será aceito, usa-se uma probabilidade de aceitação
$$a = \exp\left(f(x_2) - f(x_1)\right)/T,$$
onde T é a *temperatura*}* (pense como um *tuning*).
- Se $f(x_2) > f(x_1)$ então $a > 1$, assim o x_2 será aceito com probabilidade 1.
- Se $f(x_2) < f(x_1)$ então $0 < a < 1$.
- Assim, x_2 será aceito se $a > U(0,1)$.
- Amostrador de Metropolis no contexto de MCMC (*Markov Chain Monte Carlo*).

Ilustração: métodos baseado em simulação (SANN)



Escolhendo o melhor método

Escolhendo o melhor método

- Método de Newton é o mais eficiente (menos iterações).
- Porém, cada iteração pode ser cara computacionalmente.
- Cada iteração envolve a solução de um sistema $p \times p$.
- Métodos quasi-Newton são eficientes, principalmente se o gradiente for obtido analiticamente.
- Quando a função é suave os métodos de Newton e quasi-Newton geralmente convergem.
- Métodos baseados apenas em gradiente são simples computacionalmente.
- Em geral precisam de *tuning* o que pode ser dificil na prática.
- Método de Nelder-Mead é simples e uma escolha razoável.
- Métodos baseados em simulação são ideal para funções com máximos/minimos locais.
- Em geral são caros computacionalmente e portanto lentos.

Escolhendo o melhor método

- Em R o pacote `optimx()` fornece funções para avaliar e comparar o desempenho de métodos de otimização.
- Exemplo: minimizando a Normal bivariada.
- Escrevendo a função objetivo

```
require(mvtnorm)
fx <- function(xx){-dmvnorm(xx)}
```

Escolhendo o melhor método

- Comparando os diversos algoritmos descritos.

```
require(optimx)
```

```
## Carregando pacotes exigidos: optimx
```

```
res <- optimx(par = c(-1,1), fn = fx,
              method = c("BFGS","Nelder-Mead","CG"))
res
```

```
##                         p1            p2      value fevals gevals
## BFGS         -1.772901e-06  1.772901e-06 -0.1591549     13     11
## Nelder-Mead   1.134426e-04 -1.503306e-04 -0.1591549     55     NA
## CG           -8.423349e-06  8.423349e-06 -0.1591549     97     49
##              niter convcode kkt1 kkt2 xtime
## BFGS            NA        0 TRUE TRUE 0.004
## Nelder-Mead     NA        0 TRUE TRUE 0.002
## CG              NA        0 TRUE TRUE 0.011
```

Algumas recomendações

- Otimização trata todos os parâmetros da mesma forma.
- Cuidado com parâmetros em escalas muito diferentes.
- Padronizar entradas pode ser uma opção.
- Cuidado com parâmetros restritos.
- Recomendações
  - Torne todos os parâmetros irrestritos.
  - Faça sua função a prova de erros.
  - Entenda quais são as regiões que o algoritmo pode falhar.
  - Use o máximo possível de resultados analiticos.
  - Estude o comportamento da sua função.