Métodos numéricos

Prof. Dr. Wagner Hugo Bonat

Estrutura e objetivos do módulo

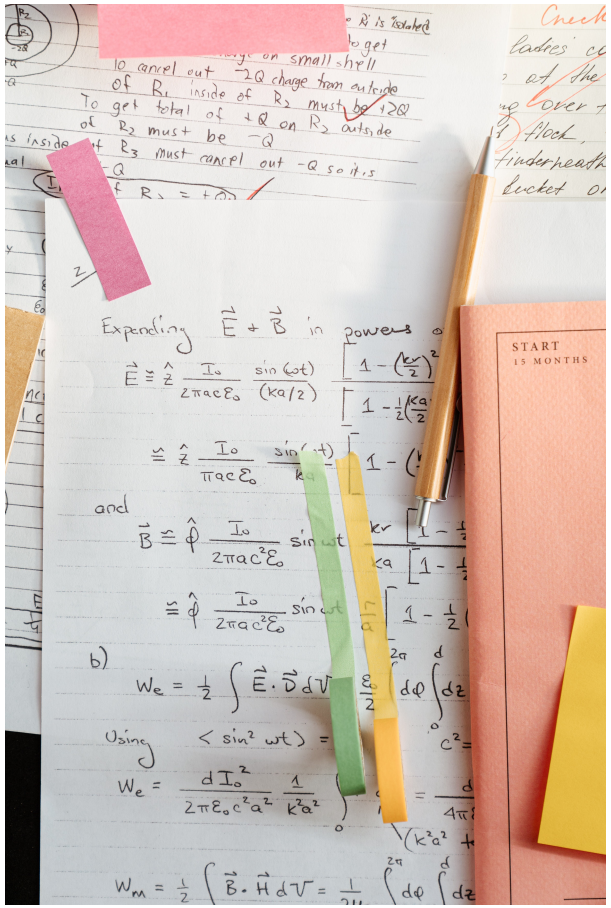
Estrutura do módulo

- Sistemas de equações não lineares.
- Métodos de confinamento.
- Métodos abertos.
  - Método de Newton.
  - Método do gradiente descendente.
- Otimização matemática.
  - Programação linear.
  - Programação quadrática.
  - Programação não linear.

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

Objetivos do módulo

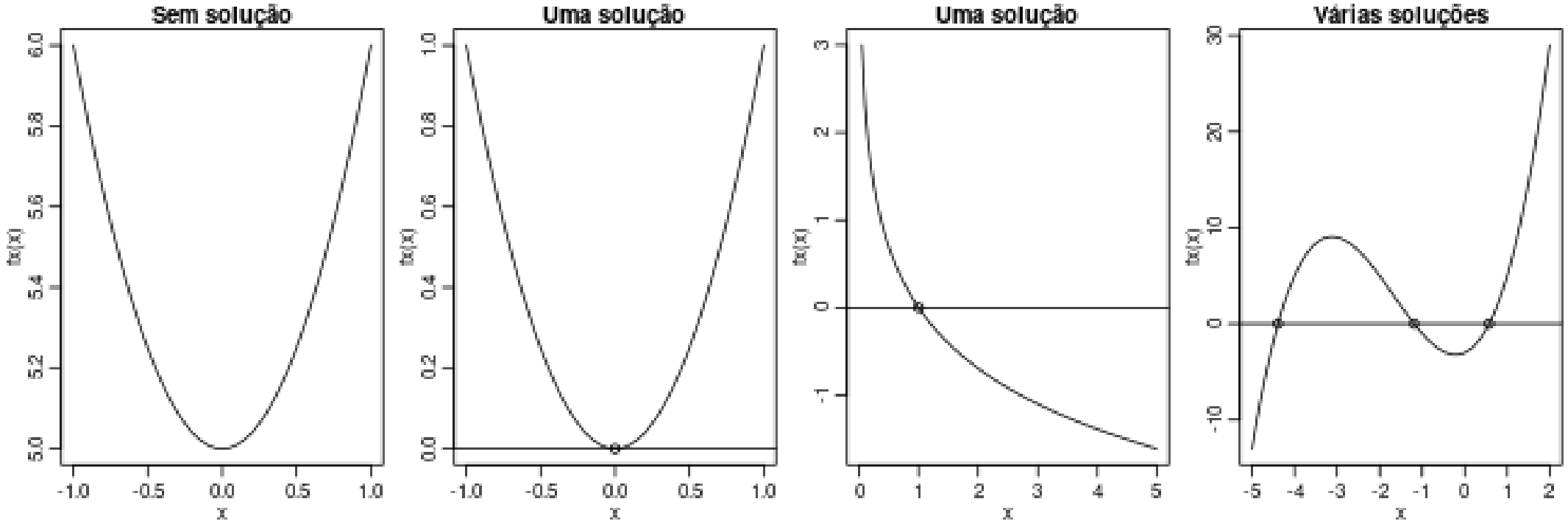
- Apresentar os principais métodos numéricos para resolver sistemas de equações não lineares.
- Apresentar o ecosistema `R` para otimização.
- Resolver problemas simples de programação linear.
- Apresentar os principais algoritmos de programação não linear.
- Resolver problemas de programação não linear usando a função `optim()`.
  - Métodos *gradient free*.
  - Métodos baseados em gradiente.
  - Métodos de Newton e quasi-Newton.
- Comparando métodos de otimização.



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.



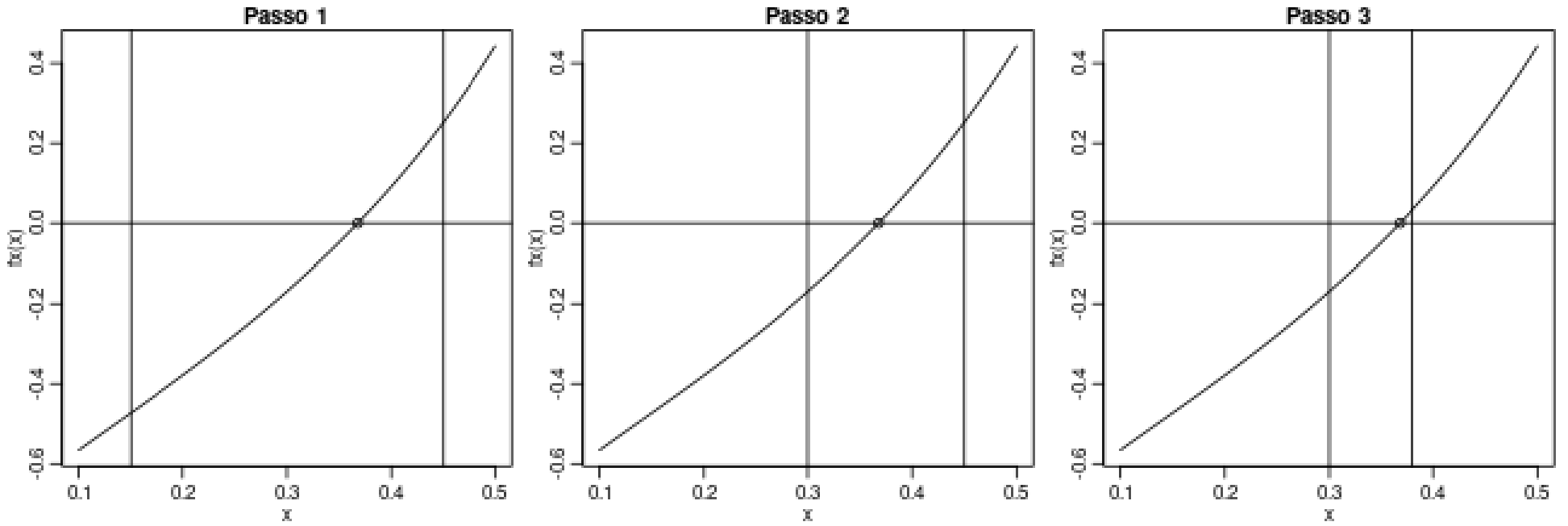
- Solução de uma equação do tipo $f(x) = 0$ é o ponto onde $f(x)$ cruza ou toca o eixo x.

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.

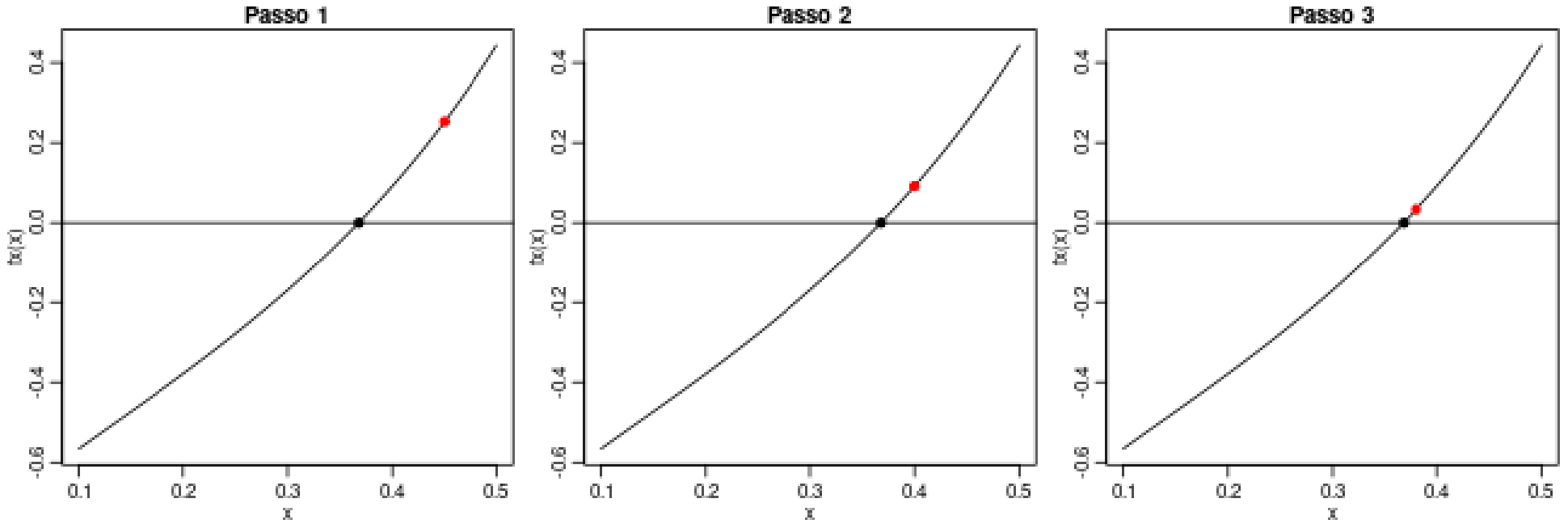
Métodos de confinamento

- 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:

$$\left| \frac{b-a}{2} \right|.$$

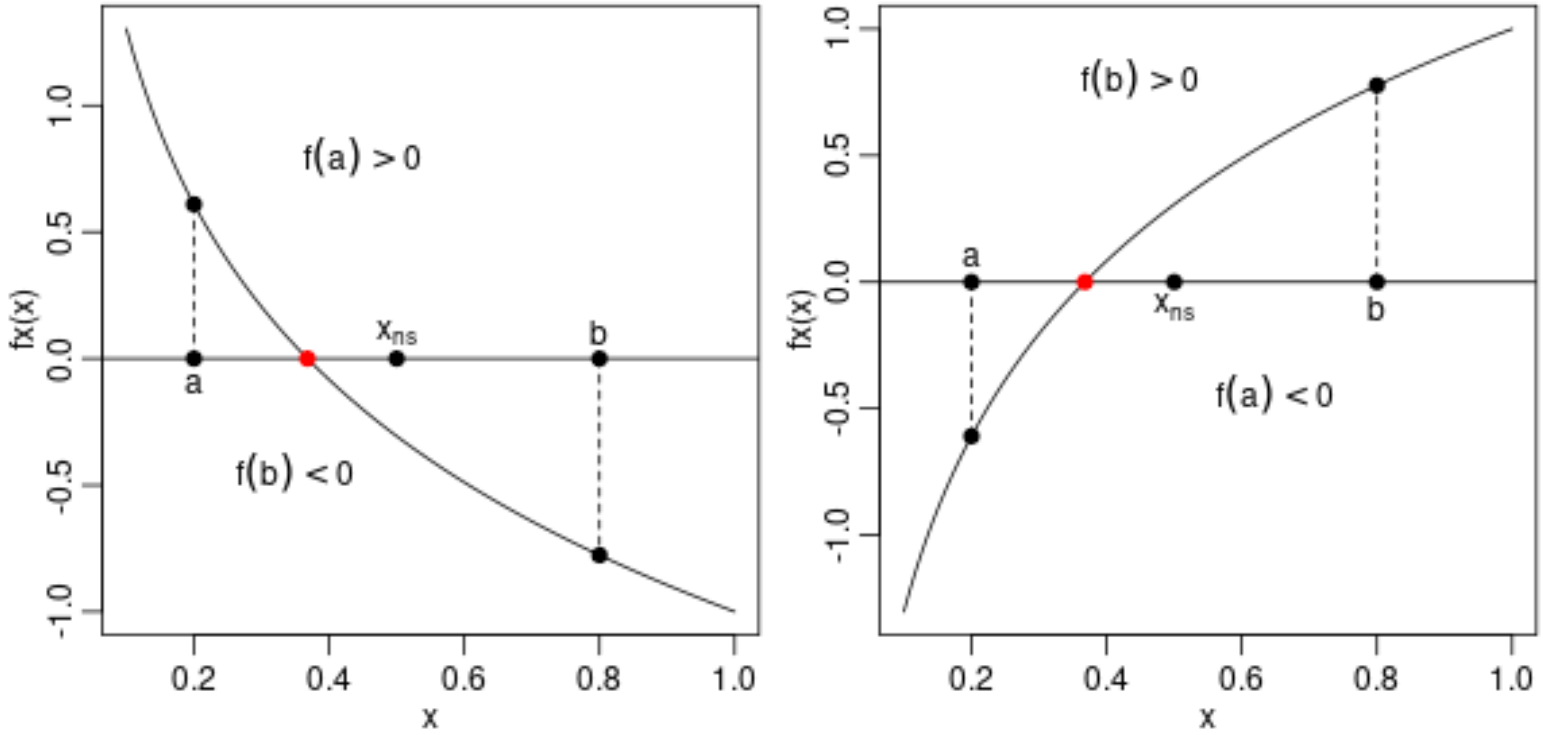
  4. Erro relativo estimado:

$$\left| \frac{x_{ns}^{n} - x_{ns}^{n-1}}{x_{ns}^{n-1}} \right|.$$

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

```r
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

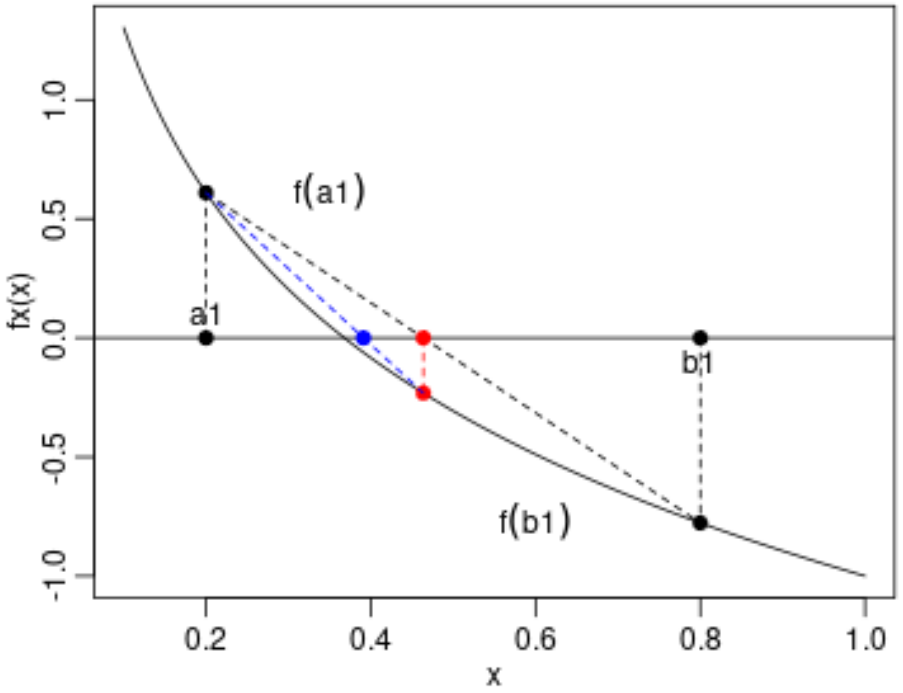
$$D(\theta) = 2n\left[\log\left(\frac{\hat{\theta}}{\theta}\right) + \bar{y}(\theta - \hat{\theta})\right] \leq 3.84.$$

```
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
```

Algoritmo: método regula falsi

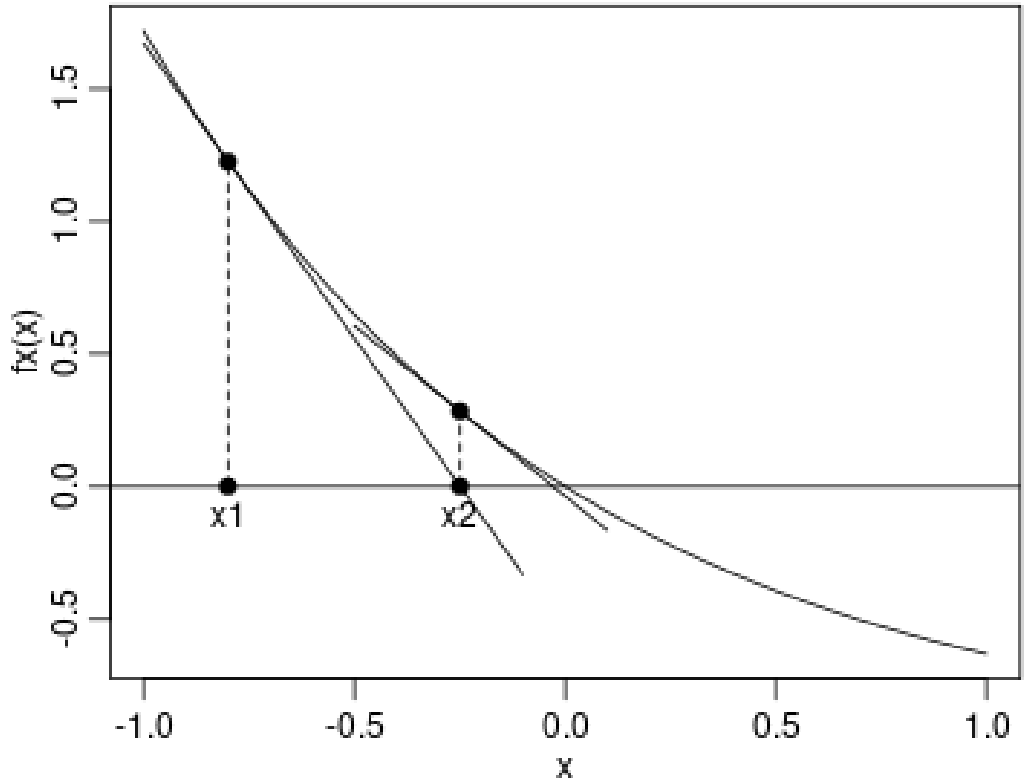
- Sabe-se que dentro de um intervalo $[a, b]$, $f(x)$ é contínua e possui uma solução.
  - Escolha os pontos a e b entre os quais existe uma solução.
  - Calcule a primeira estimativa: $x^{(i)} = \frac{af(b) - bf(a)}{f(b) - f(a)}$.
  - Determine se a solução está entre a e x^i, ou entre $x^{(i)}$ e b.
    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.



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 $i = 1, 2, \ldots$ 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 $i = 1, 2, \ldots$ 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:

$$f_1(x_1, x_2) = 0$$
$$f_2(x_1, x_2) = 0.$$

- A solução numérica consiste em encontrar $\hat{x}_1$ e $\hat{x}_2$ que satisfaça o sistema de equações.
- A ideia é facilmente estendida para um sistema com n equações

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

- Genericamente, tem-se

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

Algoritmo: método de Newton

- Escolha um vetor $\boldsymbol{x}_1$ como inicial.
- Para $i = 1, 2, \ldots$ até que o erro seja menor que um valor especificado, calcule

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

onde

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

é chamado Jacobiano de $\mathbf{f}(\boldsymbol{x})$.

- 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

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

- Precisamos obter o Jacobiano, assim tem-se

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

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
```

Método gradiente descendente

- Escolha um vetor $\boldsymbol{x}^{(1)}$ como inicial.
- Para $i = 1, 2, \ldots$ até que o erro seja menor que um valor especificado, calcule

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

- 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

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

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!

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.

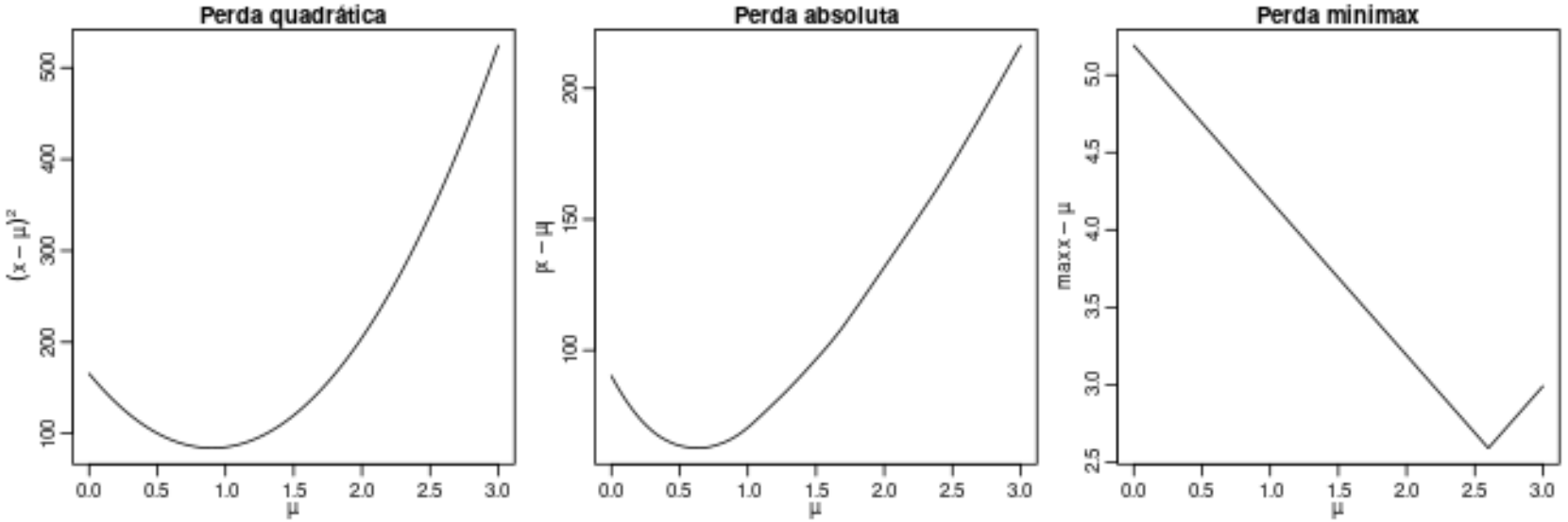
Otimização matemática

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 $max(|y_i - \mu|)$.

Motivação

- 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.
  - $\min_{\boldsymbol{x}} \quad \boldsymbol{c}^\top \boldsymbol{x}$, sujeito a $\boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}$ e $\boldsymbol{x} \geq 0$.
- Programação quadrática (QP)
  - Função objetivo é quadrática e as restrições são lineares.
  - $\min_{\boldsymbol{x}} \quad \boldsymbol{x}^\top \boldsymbol{Q}\boldsymbol{x} + \boldsymbol{c}^\top \boldsymbol{x}$, sujeito a $\boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}$ e $\boldsymbol{x} \geq 0$.
- 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

- 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
```

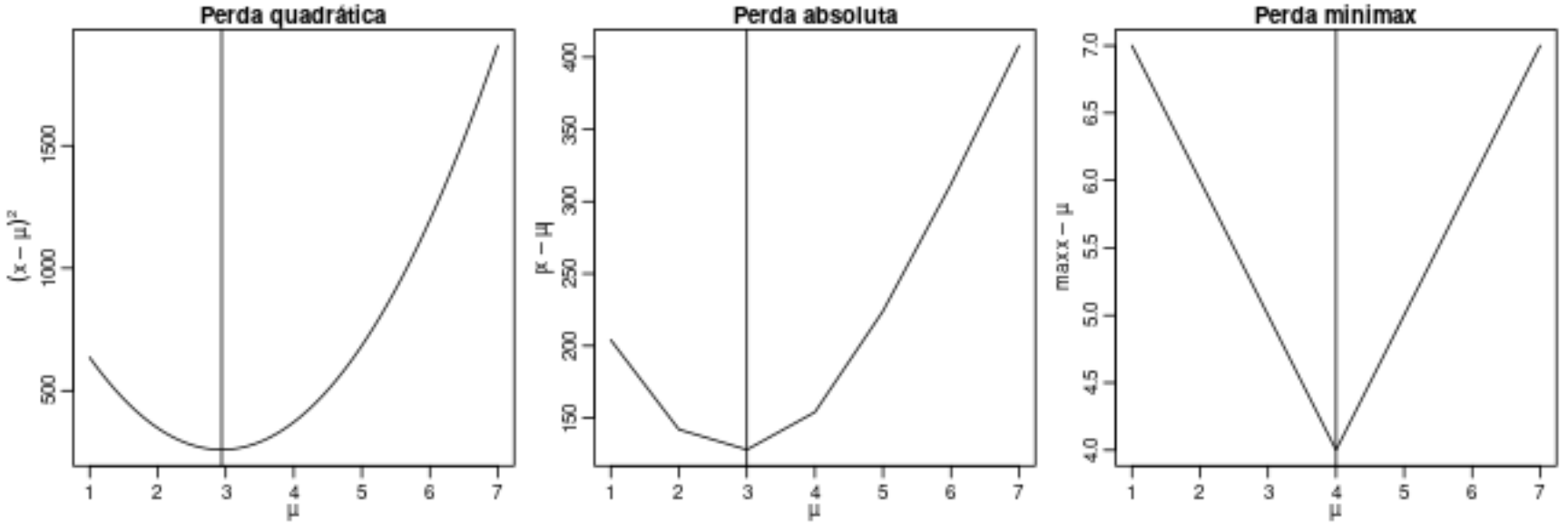
```
## $minimum
## [1] 2.999952
##
## $objective
## [1] 128.0007
```

- Perda minimax

```
fit_minimax
```

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

Otimizando funções perda: redução de dados



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.

$$\min_x c^\top x = \min_x c_1x_1 + c_2x_2 + \dots + c_nx_n$$
$$\text{s.t} \quad Ax \geq b, x \geq 0$$

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.

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

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.

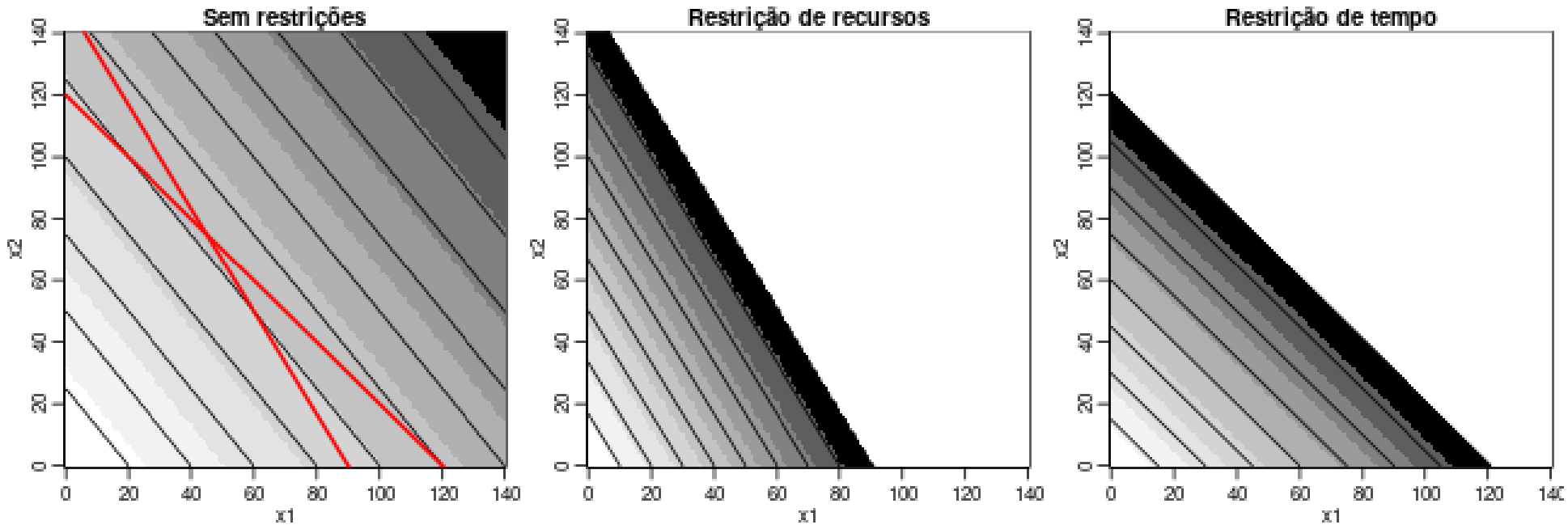
```
recurso <- function(x) {
  out <- 20*x[1] + 12*x[2]
  if(out > 1800) out = 0
  return(out)
}
```

- 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)
```

- 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)
```

- Solução

```
optimum$solution # Solução
```

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

```
optimum$objval # Lucro
```

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

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.

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

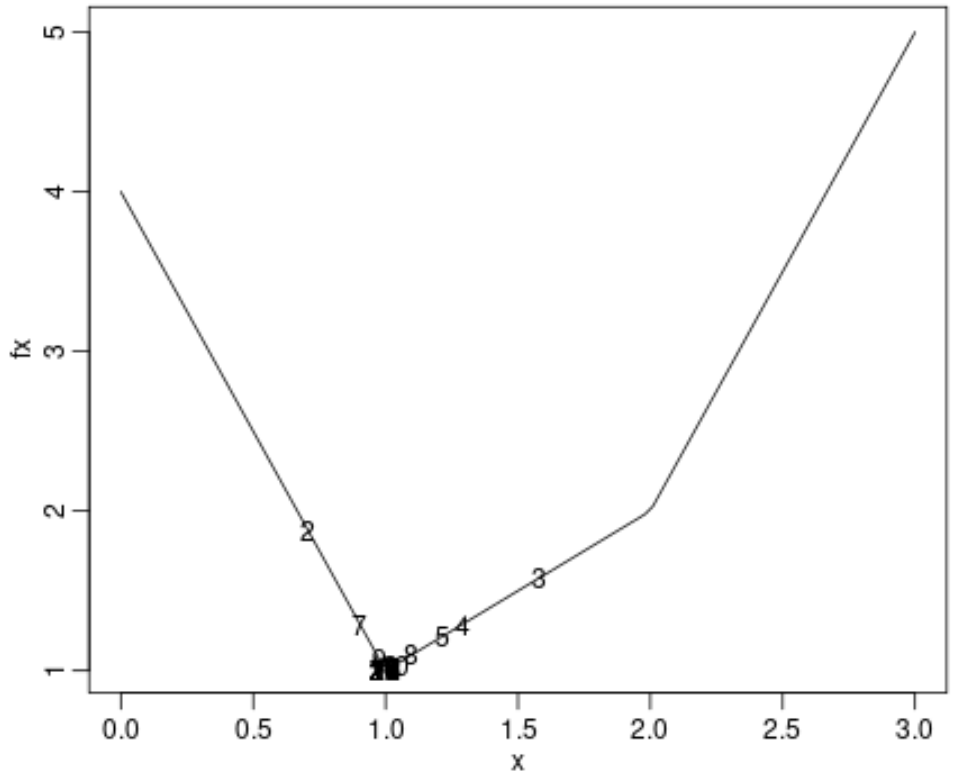
- Minize a função $f(x) = |x - 2| + 2|x - 1|$.
- Implementando e otimizando.

```
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)

- Algoritmo de Nelder-Mead

1. Escolha um simplex com $n+1$ pontos $p_1(x_1, y_1), \ldots p_{n+1}(x_{n+1}, y_{n+1})$, sendo n o número de parâmetros.
2. Calcule $f(p_i)$ e ordene por tamanho $f(p_1) \leq \ldots f(p_n)$.
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

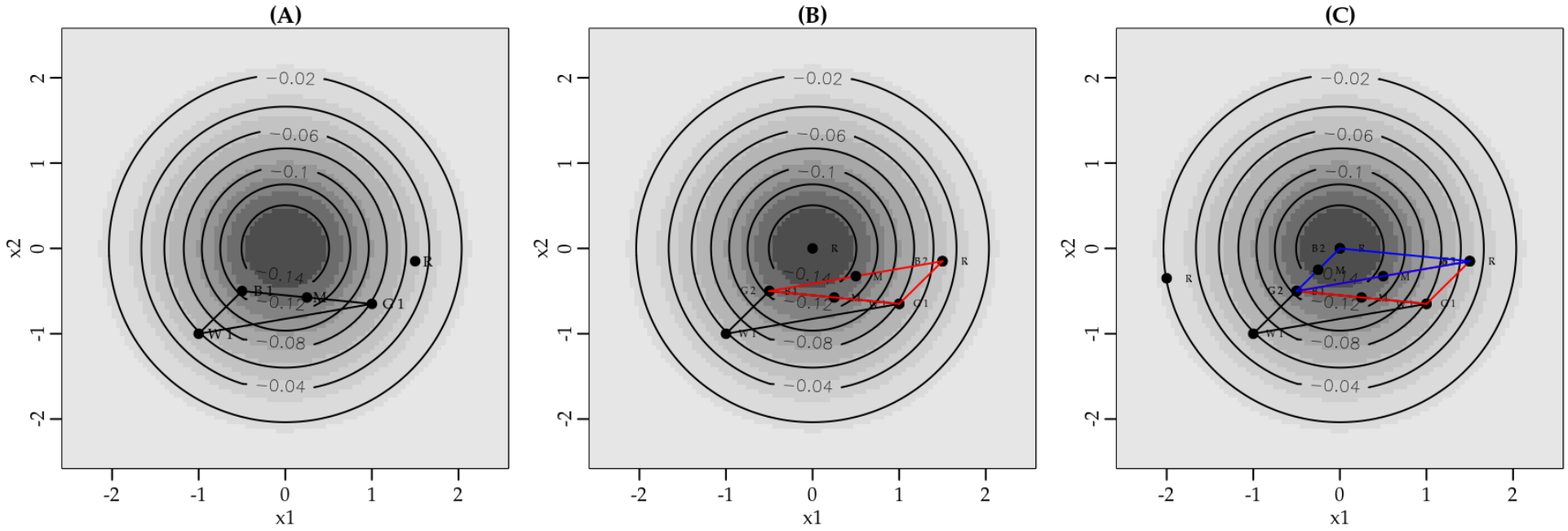
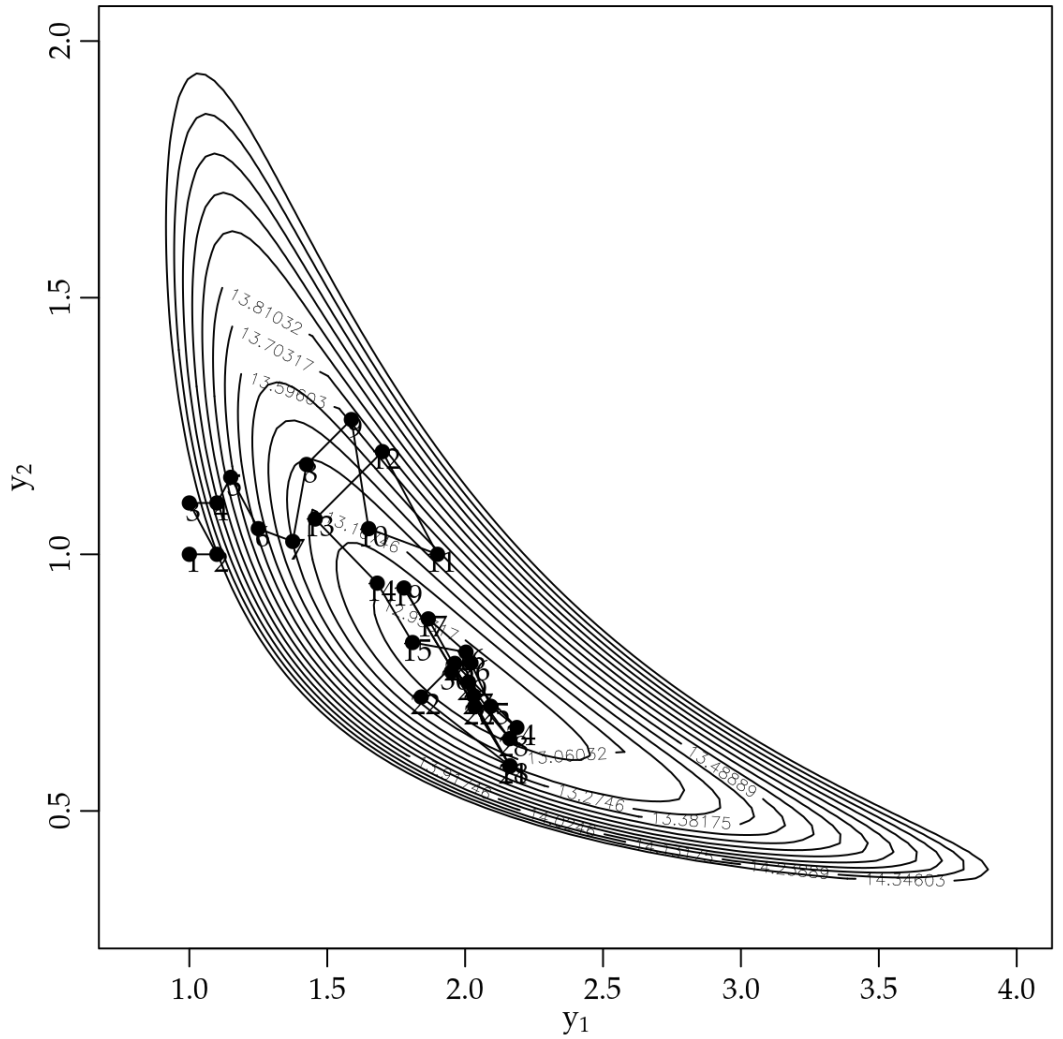


Ilustração método de Nelder-Mead.

Algoritmo de Nelder-Mead: ilustração



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

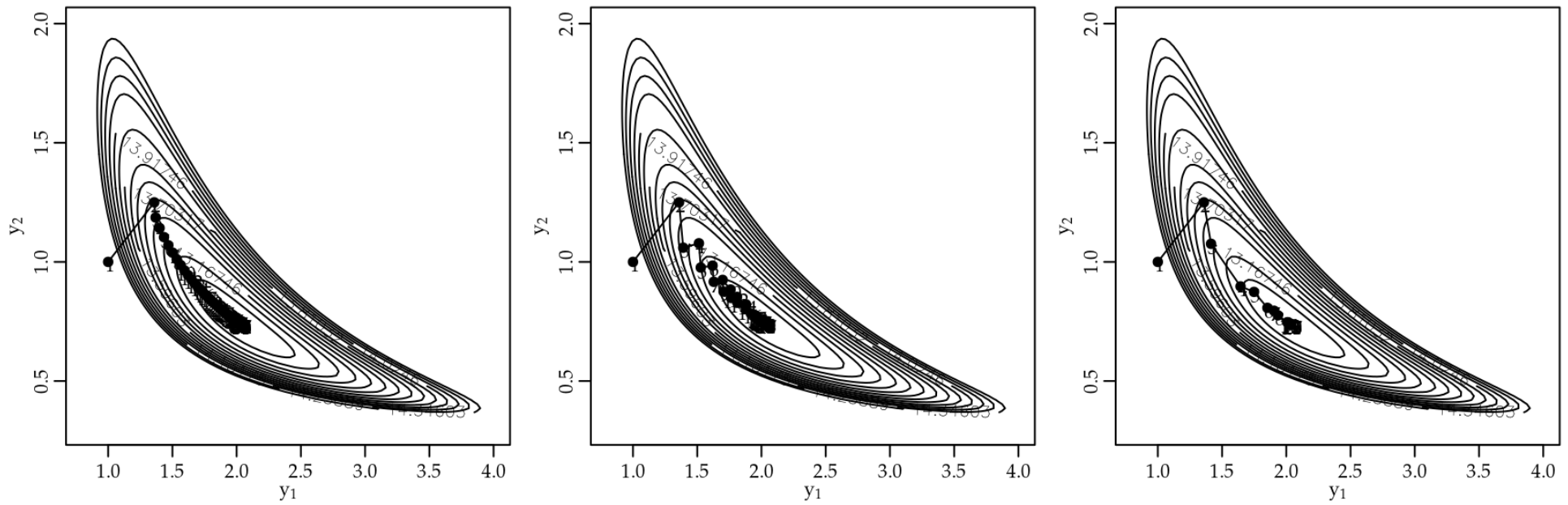


Ilustração método gradiente descendente com diferente estratégias de tuning.

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

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

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

Ilustração: métodos baseado em hessiano (BFGS)

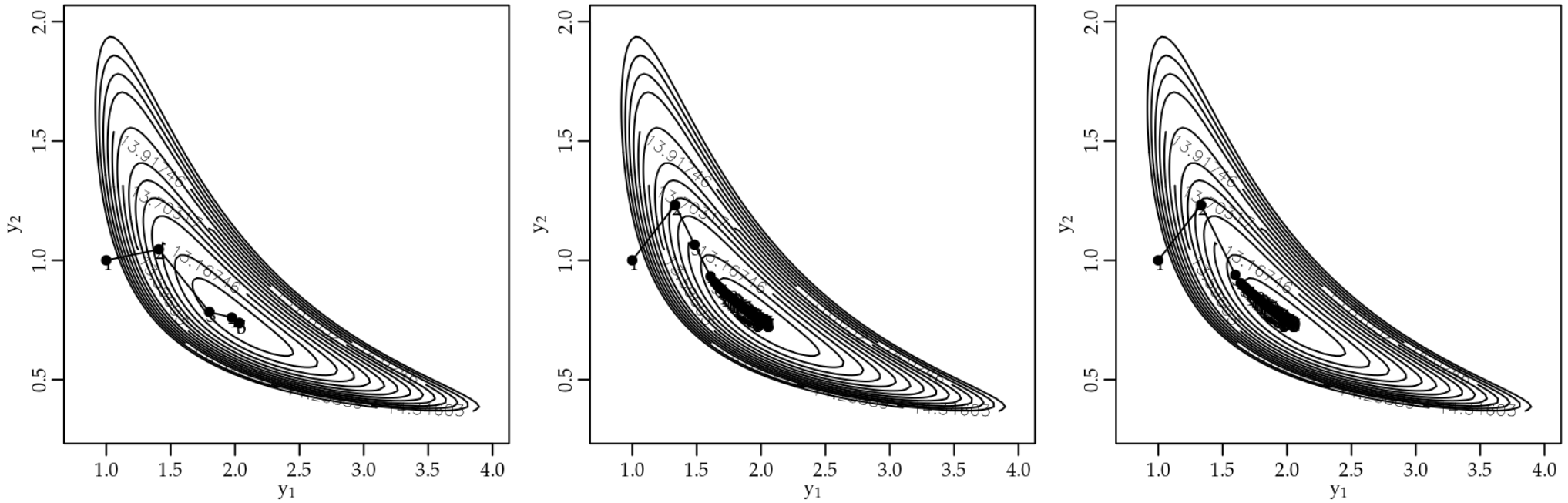


Ilustração métodos Newton, DFP e BFGS.

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.

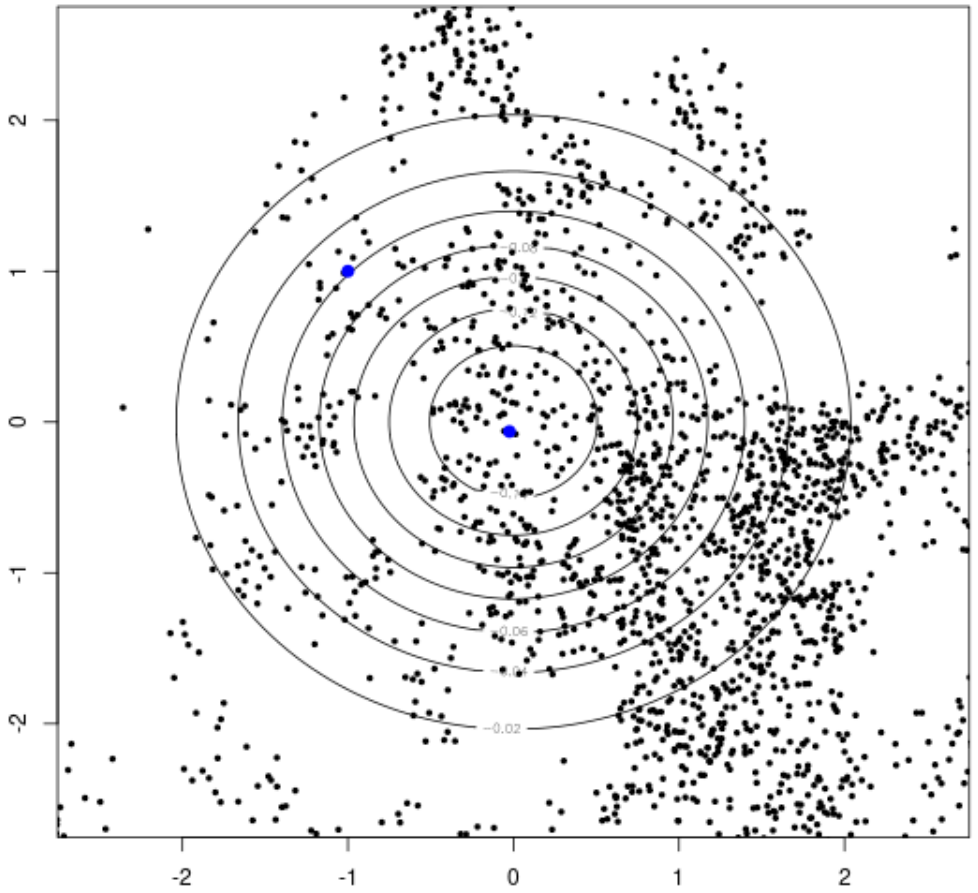
Métodos baseado em simulação: Simulating Annealing

- 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

- 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 niter convcode
## BFGS         -1.772901e-06  1.772901e-06 -0.1591549     13     11    NA        0
## Nelder-Mead   1.134426e-04 -1.503306e-04 -0.1591549     55     NA    NA        0
## CG           -8.423349e-06  8.423349e-06 -0.1591549     97     49    NA        0
##              kkt1 kkt2 xtime
## BFGS         TRUE TRUE 0.005
## Nelder-Mead TRUE TRUE 0.003
## CG           TRUE TRUE 0.010
```

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.