Sistemas de Equações não lineares

Prof. Wagner Hugo Bonat

Bacharelado em Estatística
Universidade Federal do Paraná

15 de outubro de 2018

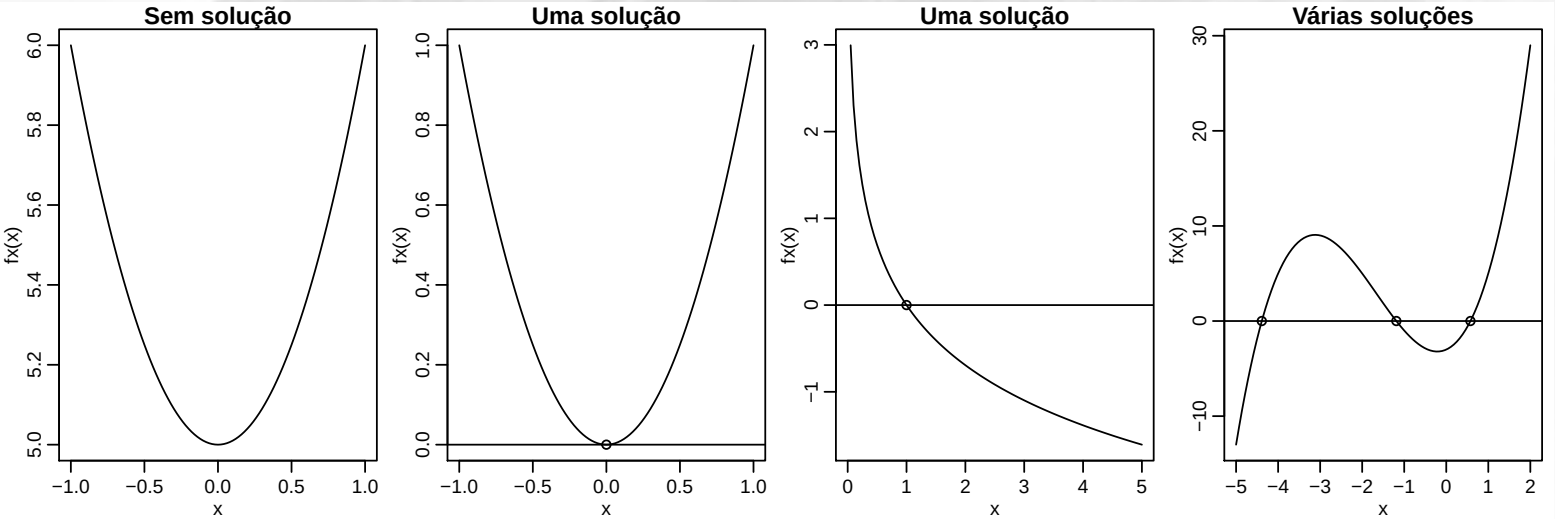
Conteúdo

Conteúdo

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.



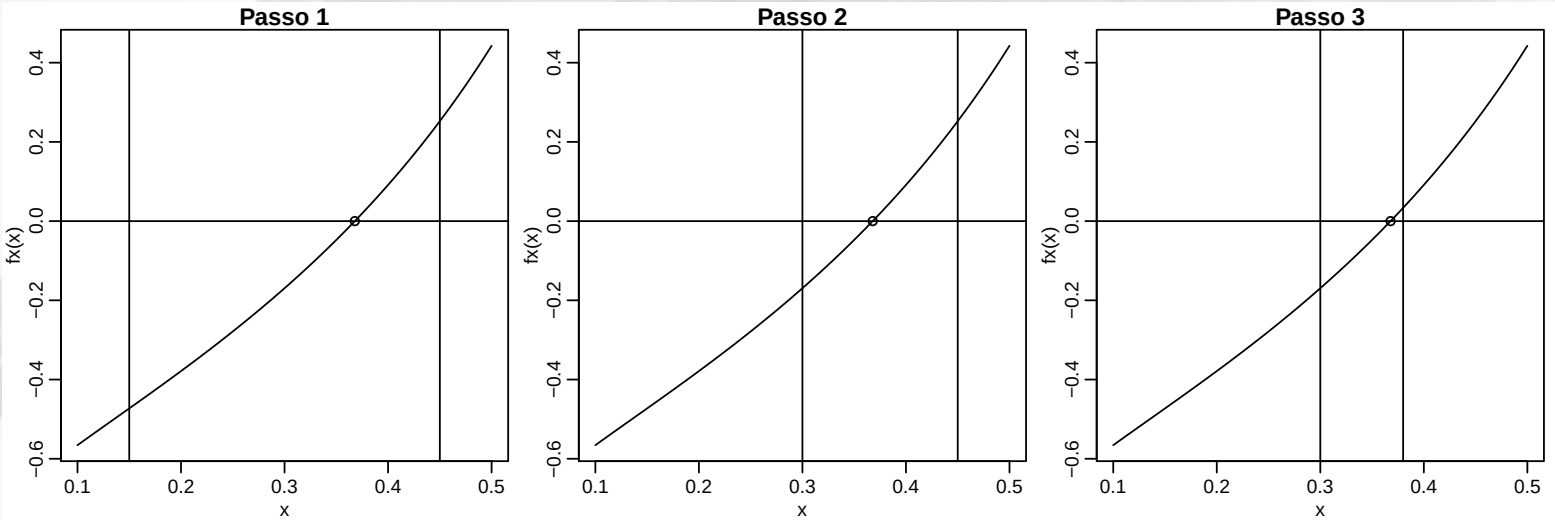
- A solução de uma equação do tipo $f(x) = 0$ é o ponto onde $f(x)$ cruza ou toca o eixo x.

Solução de equações não lineares

- Quando a equação é simples a **raiz** pode ser determinada analiticamente.
- Exemplo trivial $3x + 8 = 0 \rightarrow x = -\frac{8}{3}$.
- Em muitas situações é impossível determinar a **raiz** analiticamente.
- 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.

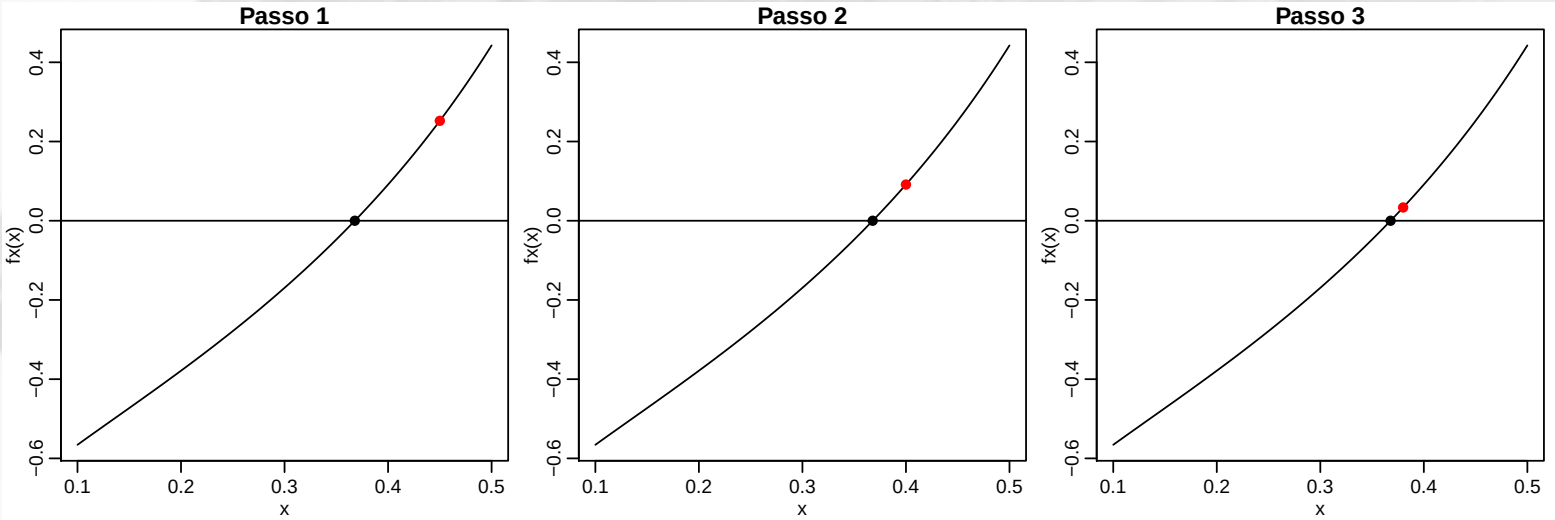
Métodos de confinamento

- 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

- Soluções numéricas não são exatas.
- 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 na solução: Tolerância máxima da qual a solução numérica pode desviar da solução verdadeira. Útil em geral quando métodos de confinamento são usados

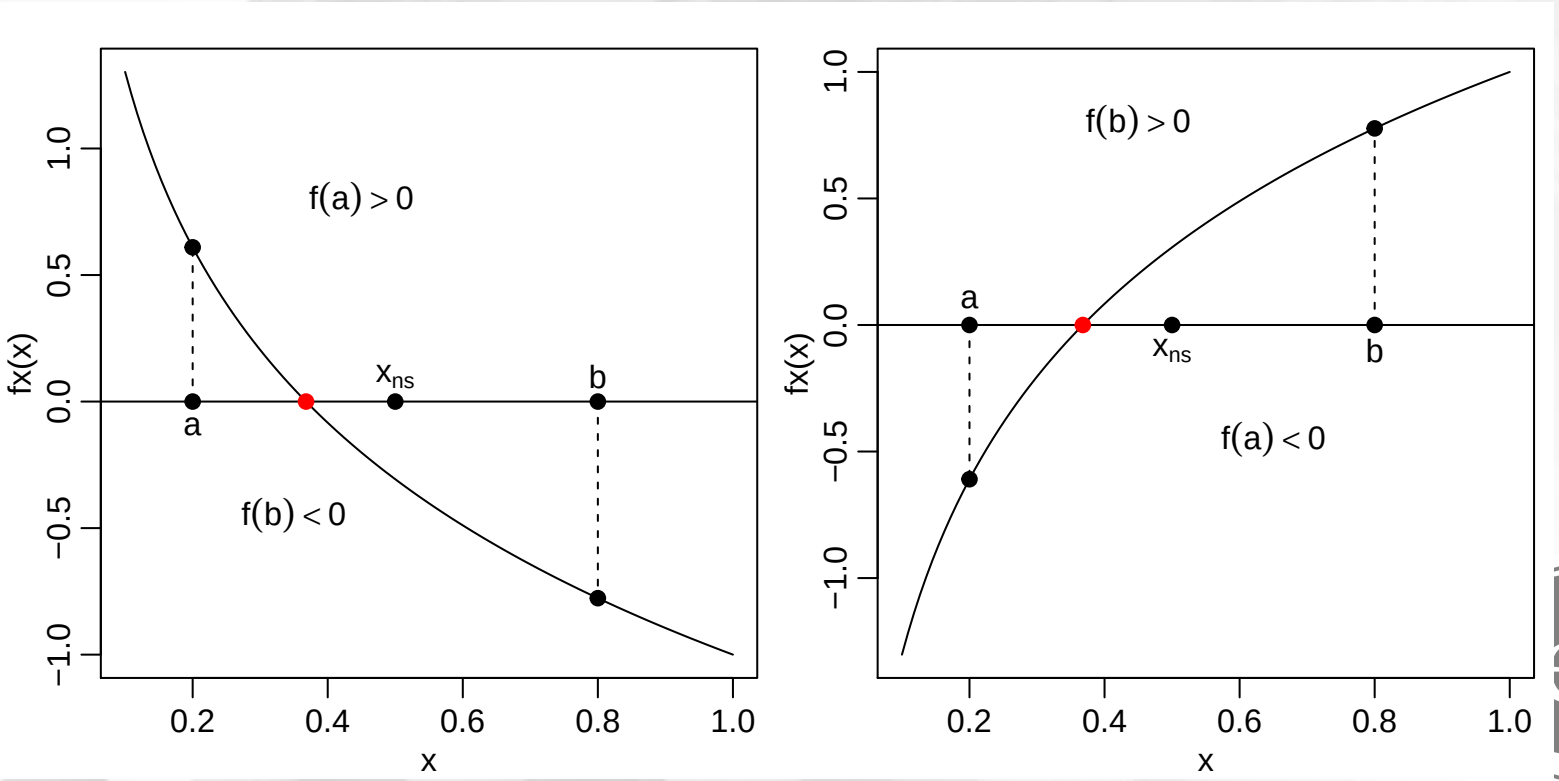
$$\left|\frac{b-a}{2}\right|.$$

  4. Erro relativo estimado:

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

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

1. Encontre $[a, b]$, tal que $f(a)f(b) < 0$.
2. Calcule a primeira estimativa $x_{ns}^{(1)}$ usando $x_{ns}^{(1)} = \frac{a+b}{2}$.
3. 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 $f(a)f(x_{ns}^{(1)})$:
   - Se $f(a)f(x_{ns}^{(1)}) < 0$, a solução está entre a e $x_{ns}^{(1)}$.
   - Se $f(a)f(x_{ns}^{(1)}) > 0$, a solução está entre $x_{ns}^{(1)}$ e b.
4. Selecione o subintervalo que contém a solução e volte ao passo 2.
5. 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)
}
```

Encontrando a raiz de $f(x) = -\ln(x) - 1 = 0$

```r
# Implementando a função
fx <- function(x){-log(x) - 1}
# Resolvendo numericamente
resul <- bissecao(fx = fx, a = 0.1, b = 1)
resul$Tentativas
```

```
##  [1] 0.5500000 0.3250000 0.4375000 0.3812500 0.3531250 0.3671875 0.3742187 0.3707031
##  [9] 0.3689453 0.3680664 0.3676270 0.3678467 0.3679565 0.3679016
```

```r
resul$Limites[1:12,]
```

```
##            [,1]      [,2]
##  [1,] 0.1000000 0.5500000
##  [2,] 0.3250000 0.5500000
##  [3,] 0.3250000 0.4375000
##  [4,] 0.3250000 0.3812500
##  [5,] 0.3531250 0.3812500
##  [6,] 0.3671875 0.3812500
##  [7,] 0.3671875 0.3742187
##  [8,] 0.3671875 0.3707031
##  [9,] 0.3671875 0.3689453
## [10,] 0.3671875 0.3680664
## [11,] 0.3676270 0.3680664
## [12,] 0.3678467 0.3680664
```

```r
resul$Raiz # Solução aproximada
```

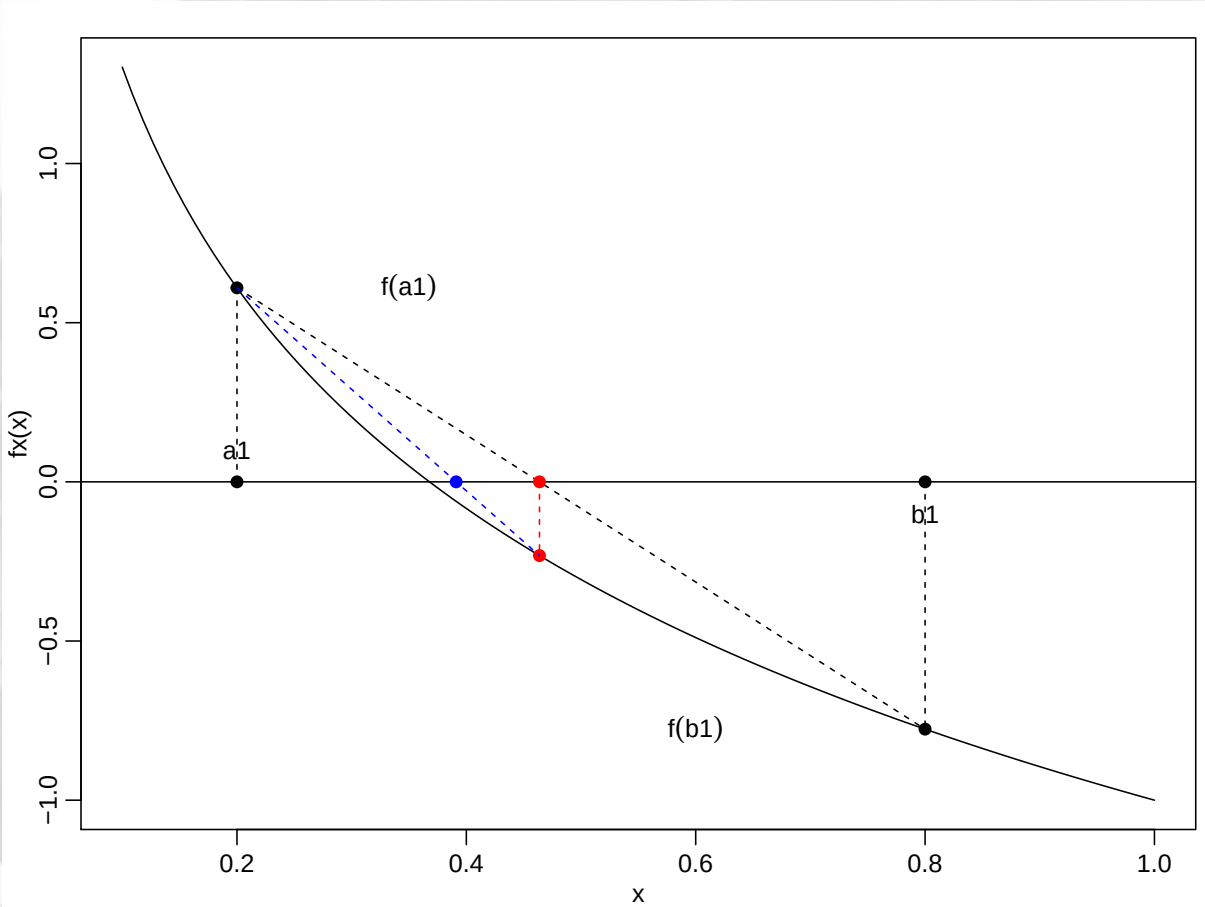
```
## [1] 0.3679016
```

```r
exp(-1) # Solução exata
```

```
## [1] 0.3678794
```

Método regula falsi

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



Algoritmo: Método regula falsi

1. Escolha os pontos a e b entre os quais existe uma solução.
2. Calcule a primeira estimativa: $x^{(i)} = \frac{af(b)-bf(a)}{f(b)-f(a)}$.
3. Determine se a solução está entre a e x^i, ou entre $x^{(i)}$ e b.
   - 3.1 Se $f(a)f(x^{(i)}) < 0$, a solução está entre a e $x^{(i)}$.
   - 3.2 Se $f(a)f(x^{(i)}) > 0$, a solução está entre $x^{(i)}$ e b.
4. Selecione o subintervalo que contém a solução como o novo intervalo $[a, b]$ e volte ao passo 2.
5. 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)
}
```

Encontrando a raiz de $f(x) = -\ln(x) - 1 = 0$

```r
# Implementando a função
fx <- function(x){-log(x) - 1}
# Resolvendo numericamente
resul <- regula_falsi(fx = fx, a = 0.1, b = 1)
resul$Tentativas
```

```
##  [1] 0.6091350 0.4670390 0.4102039 0.3862710 0.3759367 0.3714222 0.3694397 0.3685671
##  [9] 0.3681826 0.3680131 0.3679384
```

```r
resul$Limites[1:9,]
```

```
##       [,1]      [,2]
##  [1,]  0.1 0.6091350
##  [2,]  0.1 0.4670390
##  [3,]  0.1 0.4102039
##  [4,]  0.1 0.3862710
##  [5,]  0.1 0.3759367
##  [6,]  0.1 0.3714222
##  [7,]  0.1 0.3694397
##  [8,]  0.1 0.3685671
##  [9,]  0.1 0.3681826
```

```r
resul$Raiz # Solução aproximada
```

```
## [1] 0.3680131
```

```r
exp(-1) # Solução exata
```

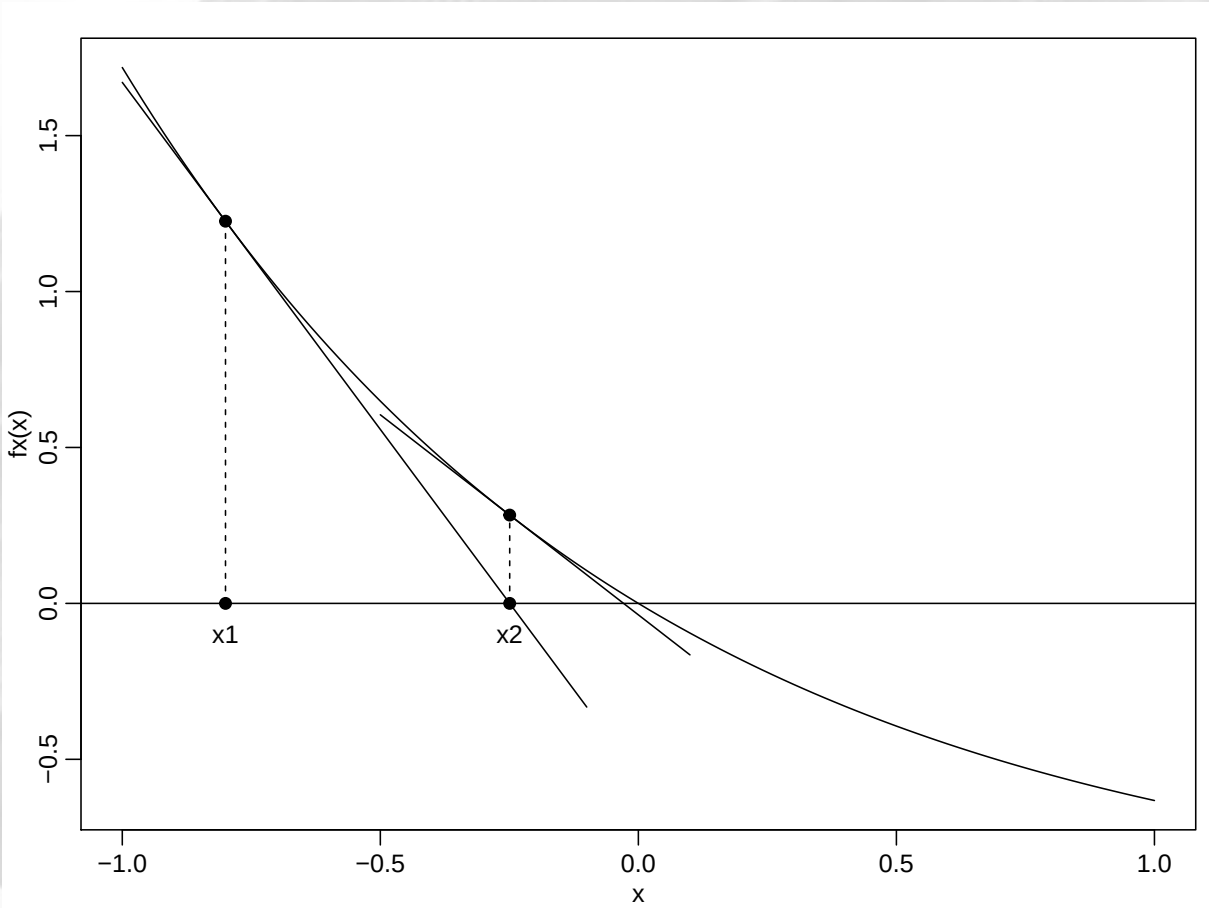
```
## [1] 0.3678794
```

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.

Método de Newton

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



Algoritmo: Método de Newton

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

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

- 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

- Resolva $f(x) = -\ln(x) - 1$.
- Derivada $f'(x) = -1/x$.

```
# Função a ser resolvida
fx <- function(x){-log(x) - 1}
# Derivada da função a ser resolvida
fprime <- function(x){-1/x}
# Solução numerica
sol_new <- newton(fx = fx, f_prime = fprime, x1 = 0.5)
sol_biss <- bissecao(fx = fx, a = 0, b = 1)
sol_reg <- regula_falsi(fx = fx, a = 0.1, b = 1)
# Método de Newton
sol_new[length(sol_new)]

## [1] 0.3678794

# Método bisseção
sol_biss$Raiz

## [1] 0.3678589

# Método regula falsi
sol_reg$Raiz

## [1] 0.3680131

exp(-1) # Solução exata

## [1] 0.3678794
```

Método Gradiente Descendente

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

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

- $\alpha > 0$ é um parâmetro de *tuning* usado para controlar o tamanho do passo.
- Vamos voltar a trabalhar com o Gradiente descendente em métodos de otimização.

Algoritmo: Método Gradiente descendente

1. Escolha um ponto x_1 como inicial.
2. 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(sol[i+1] < 0) {sol[i+1] = 1e-04}
    if(abs(fx(sol[i+1])) < tol) break
  }
  return(sol)
}
```

Aplicação: Método Gradiente descendente

- Resolva $f(x) = -\ln(x) - 1$.

```
# Função a ser resolvida
fx <- function(x){-log(x) - 1}
# Solução numerica
sol_grad <- grad_des(fx = fx, alpha = 0.2, x1 = 1)
sol_grad[length(sol_grad)]

## [1] 0.3679003

# Exata
exp(-1)

## [1] 0.3678794
```

- Escolha do α é fundamental para atingir convergência.
- Busca em gride pode ser uma opção razoável.

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.
- A idéia 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

1. Escolha um vetor $\boldsymbol{x}_1$ como inicial.
2. 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: 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

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

Aplicação: Método de Newton

```r
# 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)
}
# 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

1. O método estende naturalmente para sistema de equações não-lineares.
2. Escolha um vetor $\boldsymbol{x}_1$ como inicial.
3. 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

```
fx2 <- function(x) { fx(abs(x)) }
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(solucao[i+1,])
    if( sum(abs(solucao[i+1,] - solucao[i,])) < tol) break
  }
  return(sol)
}
```

Aplicação: Método Gradiente descendente

- Resolva

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

```
sol_grad <- grad_des(fx = fx2, x1 = c(2.5, 2), alpha = 0.025, max_iter = 20)
tail(sol_grad)
```

```
##           [,1]     [,2]
##  [5,] 3.154278 2.362746
##  [6,] 3.034792 2.385386
##  [7,] 3.031159 2.385865
##  [8,] 3.031155 2.385866
##  [9,]       NA       NA
## [10,]       NA       NA
```

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

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

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.

Exercício: Regressão robusta

- Considere um conjunto de observações da variável de interesse y_i, para $i = 1, \ldots, n$.
- Considere uma variável explicativa x_i.
- Relacione y_i e x_i por uma reta, tal que

$$y_i = \beta_0 + \beta_1 x_i.$$

- Encontre β_0 e β_1 tal que

$$\sum_{i=1}^{n} |y_i - \beta_0 - \beta_1 x_i|,$$

seja o menor possível.