# Métodos numéricos

Prof. Wagner H. Bonat

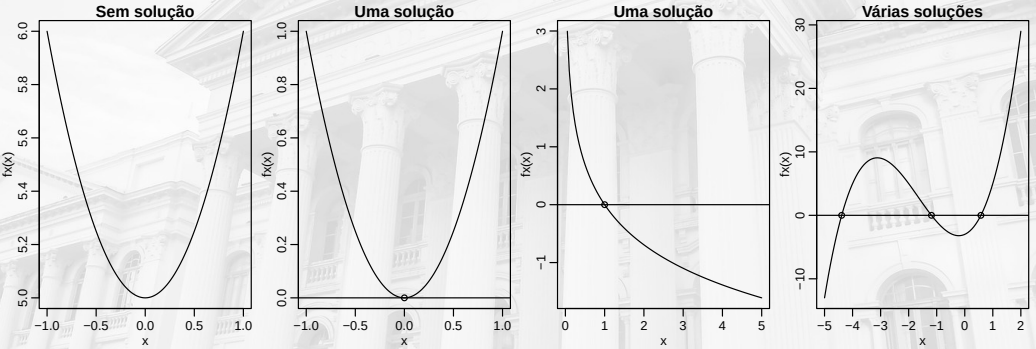
Laboratório de Estatística e Geoinformação
Departamento de Estatística
Universidade Federal do Paraná



# 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.




- 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.

# 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

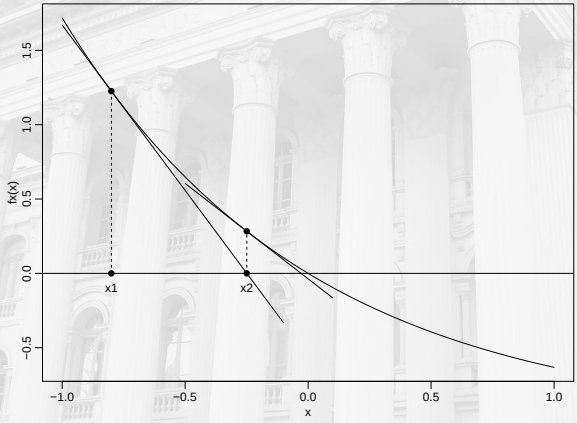
$$\left| \frac{b-a}{2} \right|.$$

  4. Erro relativo estimado:

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

# Método de Newton

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



# 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

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

- 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

- 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

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

```
ftheta <- function(theta){
  dd <- 2*length(y)*(log(theta.hat/theta) + mean(y)*(theta - theta.hat))
  return(dd - 3.84)
}
set.seed(123)
y <- rexp(20, rate = 1)
theta.hat <- 1/mean(y)
# Derivada da função a ser resolvida
fprime <- function(theta){2*length(y)*(mean(y) - 1/theta)}
```

# Aplicação: Método de Newton

```
# 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
```

# Sistemas 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

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

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

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 $\mathbf{f}(x)$.

# 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

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

# Aplicação: Método de Newton

```
# 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
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 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.
- Escolha do $\alpha$ 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
```

# Método Gradiente descendente

- O método estende naturalmente para sistema de equações não-lineares.
- Escolha um vetor $x_1$ como inicial.
- Para $i = 1,2,\ldots$ até que o erro seja menor que um valor especificado, calcule

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

- 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
  }
```

# 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 $\alpha$ e da o passo usando o $\alpha$ que torna $F(x)$ maior/menor.
- O tamanho do passo pode ser adaptativo.

# 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.

# Método Trapezoidal

# Integração numérica: 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

# Integração numérica: 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:

$$p(x) = \alpha + \beta(x - x_1) + \lambda(x - x_1)(x - x_2) \tag{1}$$

  onde $\alpha$, $\beta$ e $\lambda$ 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 $p(x_3) = f(x_3)$.

# Integração numérica: 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 1 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

- Método trapezoidal e Simpson são muito simples.
- Aproximam o integrando por um polinômio de fácil integração.
- Resolvem a integral aproximada.
- Pontos são igualmente espaçados.
- Simples e intuitivos, porém de difícil generalização.
- 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:

$$\int_a^b f(x)dx \approx \sum_{i=1}^{n} C_i f(x_i), \tag{2}$$

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

- Exemplo 1: Para $n = 2$ a Eq. 2 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. 2 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. 2 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.
- 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. 2 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

$$\int_{-1}^{1} f(x)dx \approx C_1 f(x_1) + C_2 f(x_2). \tag{3}$$

# Quadratura de Gauss-Legendre

- As quatro constantes $C_1, C_2, x_1$ e $x_2$ são determinadas fazendo Eq. 3 exata quando aplicada aos quatro casos:

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

- 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

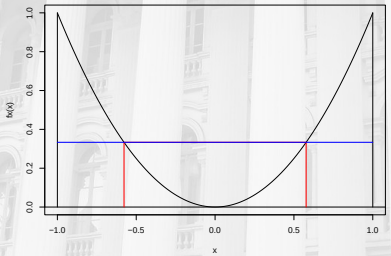
$$C_1 = 1, \quad C_2 = 1, \quad x_1 = -\frac{1}{\sqrt{3}} \quad \text{e} \quad x_2 = \frac{1}{\sqrt{3}}.$$

# Exemplo: Quadratura de Gauss-Legendre

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

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

- 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

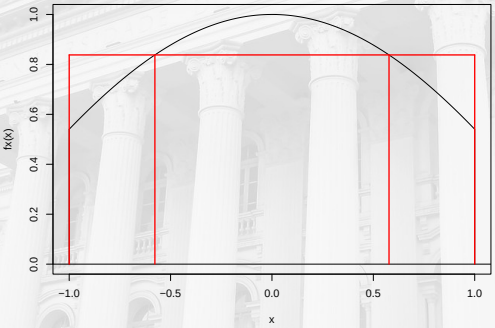
$$\int_{-1}^{1} cos(x)dx = sen(x)|_{-1}^{1} = sen(1) - sen(-1) = 1.682841.$$

- 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-Hermite

- Gauss-Hermite resolve integrais do tipo:

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

- Integral é aproximada por uma soma ponderada.

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

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

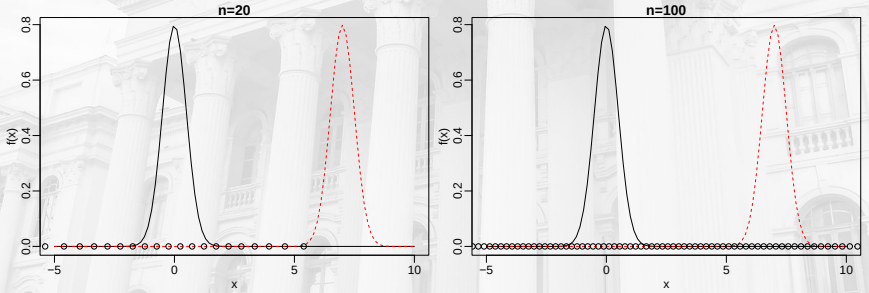
```
# n = 2
gauss_hermite(integrando = dnorm, n.pontos = 2)
## [1] 0.9079431

# 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 baseado no polinômio escolhido, ignorando a função a ser integrada.
  2. Número de pontos necessários para a integração cresce como uma potência da dimensão da integral. 20 pontos em uma dimensão demanda $20^2 = 400$ pontos em duas dimensões.




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

# Quadratura Gaussiana Adaptativa

# 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

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

- Novos pontos de integração adaptados serão dados por

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

com correspondentes pesos,

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

# 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,

$$I = \int_D f(x)dx \tag{4}$$

- Seja $p(x)$ uma fdp cujo domínio coincide com $D$.
- Então, a integral em Eq. 4 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] 0.9995921
```

# Funções residentes 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 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.

# Programação não-linear

# Métodos de otimização 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 traz de cada tipo de algoritmo.
- Existe uma infinidade de variações e implementações.

# Método Golden-Search ou Brent

# Programação não-linear: Problemas unidimensionais

- Golden Section Search é o mais popular e muito eficiente.
- Algoritmo
  1. Define 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 $f(x_1) < f(x_2)$ continua 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.

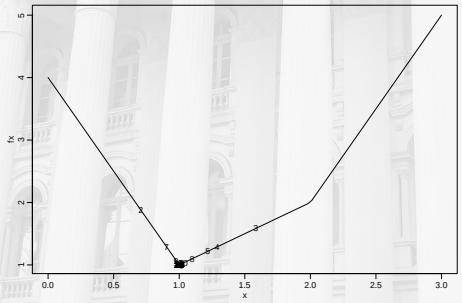
```
xx <- c()
fx <- function(x) {
  out <- abs(x-2) + 2*abs(x-1)
  xx <<- c(xx, x)
  return(out)
}
out <- optimize(f = fx, interval = c(-3,3))
out
```

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

# Exemplo: Otimização unidimensional

- Traço do algoritmo.

```
par(mfrow = c(1,1), mar=c(2.6, 3, 1.2, 0.5), mgp = c(1.6, 0.6, 0))
fx <- function(x) abs(x-2) + 2*abs(x-1)
plot(fx, 0, 3)
for(i in 1:length(xx)) {
  text(x = xx[i], y = fx(xx[i]), label = i)
}
```

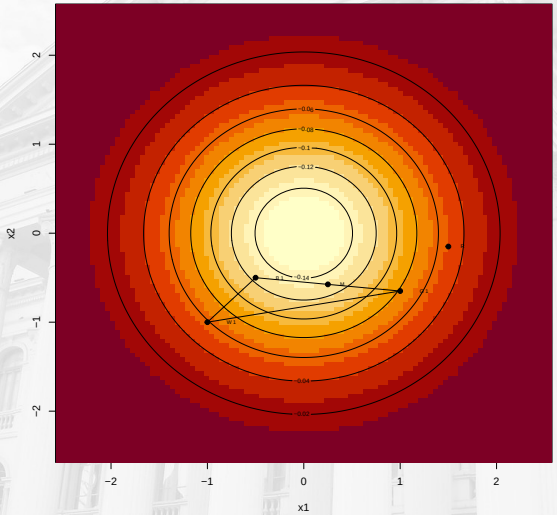


# Método de Nelder-Mead

# 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):

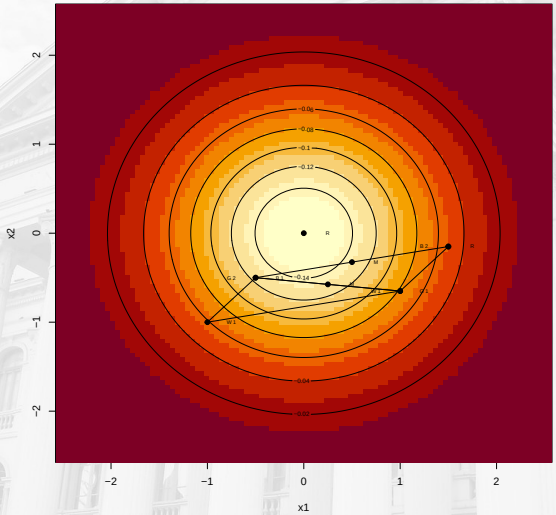
$$M = \frac{B + G}{2} = \left( \frac{x_1 + x_2}{2}, \frac{y_1 + y_2}{2} \right).$$

- 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

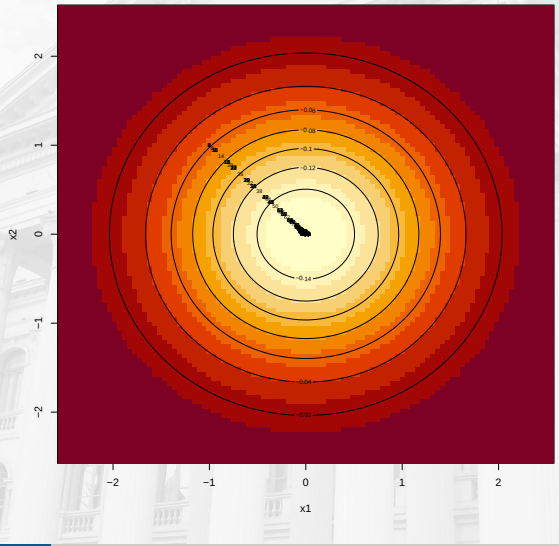


# Métodos baseados em gradiente

# 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 $\alpha_n$ na direção de $-f'(x)$.
  3. Tamanho do passo $\alpha_n$ pode ser fixo ou variável.
  4. Repita até $f'(x^i) \approx 0$ seja satisfeito.

# Ilustração: Métodos baseado em gradiente



# Métodos baseados em hessiano

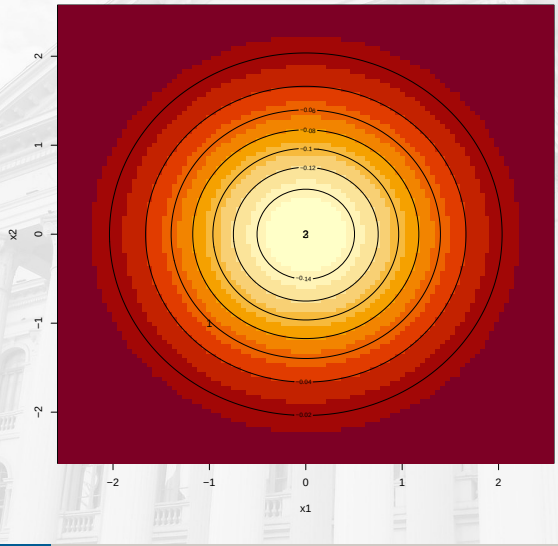
# Métodos baseado em hessiano

- 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

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

  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 (Newton)



# Métodos Quasi-Newton

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

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

  onde $\mathbf{H}_i$ é alguma aproximação para o inverso do Hessiano.
- $\alpha_i$ 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 $\mathbf{H}_{i+1}$ o algoritmo impõe que

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

- Algoritmo DFP

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

# Métodos Quasi-Newton

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

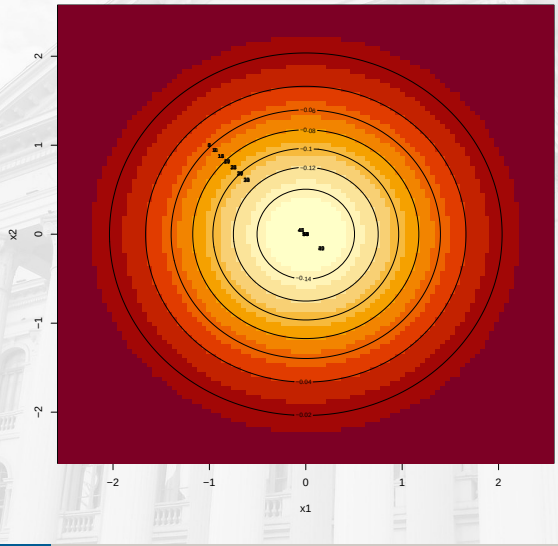
$$\mathbf{H}_{i+1} = \mathbf{H}_i - \frac{\delta_i \gamma_i^\top \mathbf{H}_i + \mathbf{H}_i \gamma_i \delta_i^\top}{\delta_i^\top \gamma_i} + \left(1 + \frac{\gamma_i^\top \mathbf{H}_i \gamma_i}{\delta_i^\top \gamma_i}\right) \left(\frac{\delta_i \delta_i^\top}{\delta_i^\top \gamma_i}\right).$$

- Implementações modernas do BFGS usando *wolfe line search* para encontrar $\alpha_i$.
- Considere $\psi(\alpha) = f(x^{(i)} - \alpha \mathbf{H}_i f'(x^{(i)}))$, encontre $\alpha_i$ 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 $\mu$ e $\eta$ são constantes com $0 < \mu \leq \eta < 1$.

# Ilustração: Métodos baseado em hessiano (BFGS)



# Métodos baseados em simulação

# Métodos baseado 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

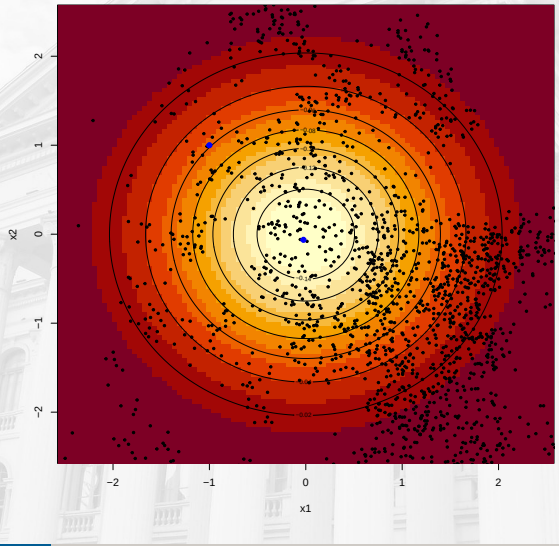
- Para decidir se um ponto $x_2$ quando $f(x_2) < f(x_1)$ será aceito, usa-se uma probabilidade de aceitação

$$a = \exp\left(f(x_2) - f(x_1)\right)/T,$$

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.

# Recomendações

# 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

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

# Escolhendo o melhor método

- Comparando os diversos algoritmos descritos.

```
require(optimx)
```

```
## Loading required package: 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 0.005
## Nelder-Mead      NA        0 TRUE TRUE 0.004
## CG               NA        0 TRUE TRUE 0.019
```

# Algumas recomendações

- Otimização trata todos os parâmetros da mesma forma.
- Cuidado com parâmetros em escalas muito diferentes.
- Cuidado com parâmetros restritos.
- Recomendação: Torne todos os parâmetros irrestritos ou faça sua função a prova de erros.
- Use o máximo possível de resultados analiticos.
- Tutorial V.