Métodos de otimização

Prof. Wagner Hugo Bonat

Bacharelado em Estatística
Universidade Federal do Paraná

15 de outubro de 2018

Conteúdo

Conteúdo

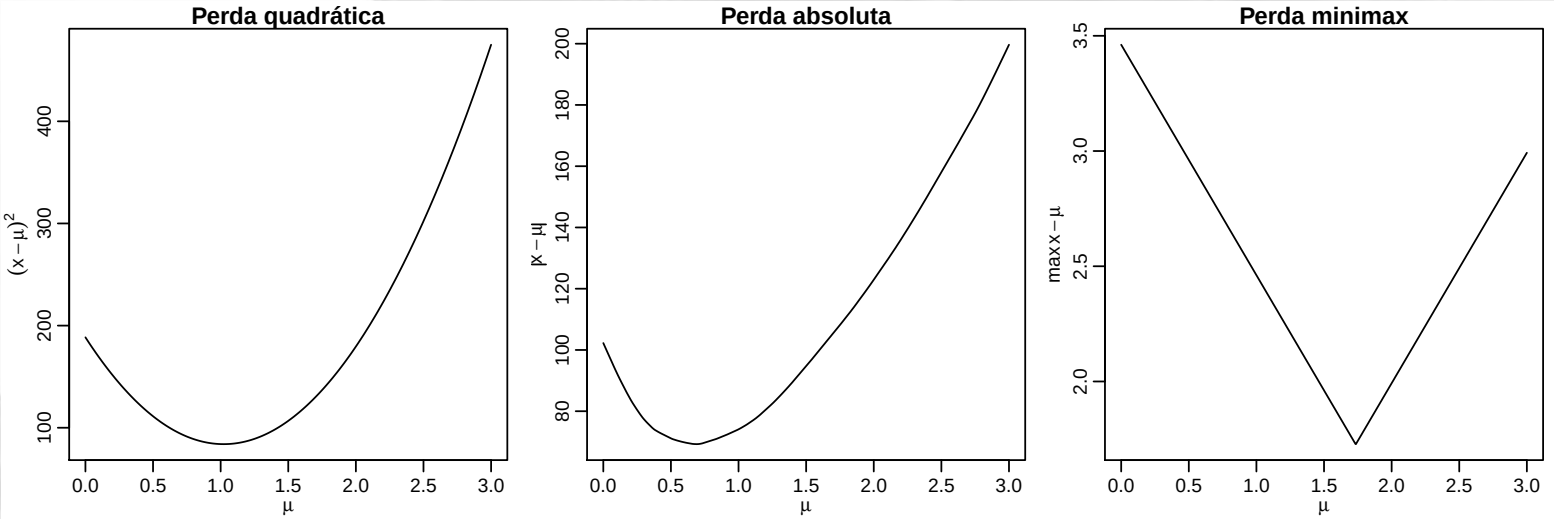
Métodos de otimização

Otimização

1. Otimização usa um modelo matemático rigoroso para determinar a solução mais eficiente para um dado problema.
2. Precisamos identificar um objetivo.
   - 2.1 Criar uma medida que mensure a performance. Ex. rendimento, tempo, custo, etc.
   - 2.2 Em geral, qualquer quantidade ou combinação de quantidades representada por um simples número.
3. Funções perda usuais.
   - 3.1 Perda quadrática: $\sum_{i=1}^{n}(y_i - \mu)^2$.
   - 3.2 Perda absoluta: $\sum_{i=1}^{n}|y_i - \mu|$.
   - 3.3 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

- Grupos comuns:
  1. 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 $\boldsymbol{x} \geq 0$.
  2. 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 $\boldsymbol{x} \geq 0$.
  3. 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*.
  URL: https://cran.r-project.org/web/views/Optimization.html

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:
  1. Perda quadrática: $\sum_{i=1}^{n}(y_i - \mu)^2$.
  2. Perda absoluta: $\sum_{i=1}^{n}|y_i - \mu|$.
  3. 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 objetivos.
  1. Perda quadrática

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

  2. Perda absoluta

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

  3. 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)
```

- 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
```

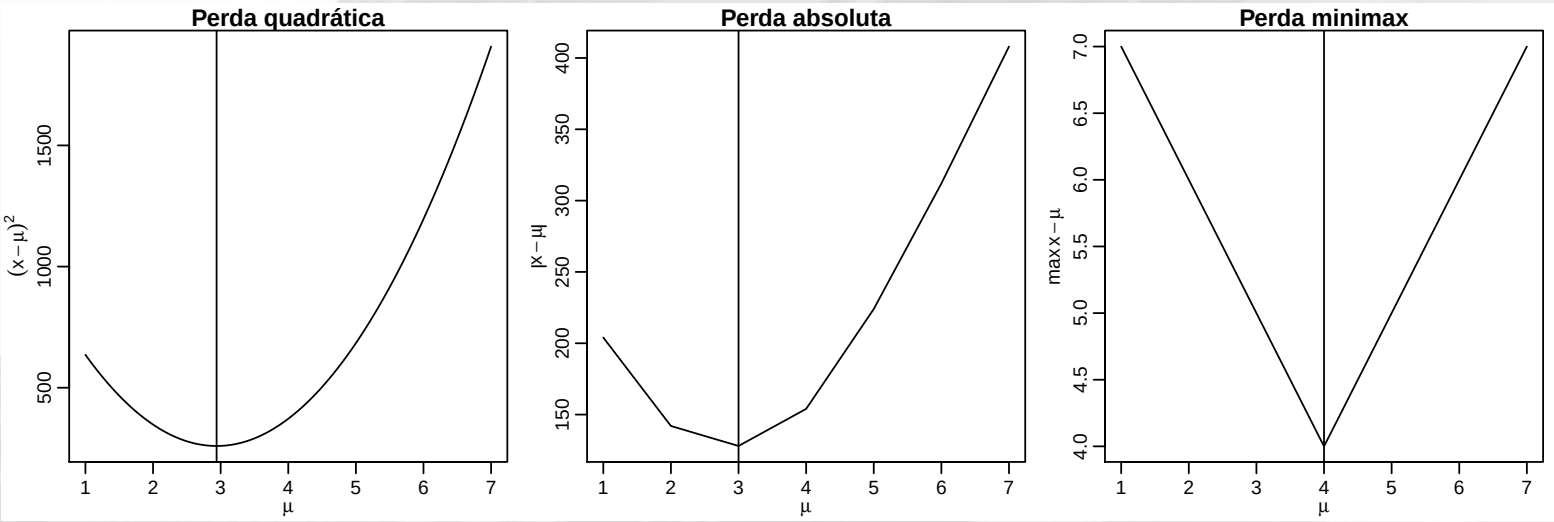
- Perda minimax

```
fit_minimax

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

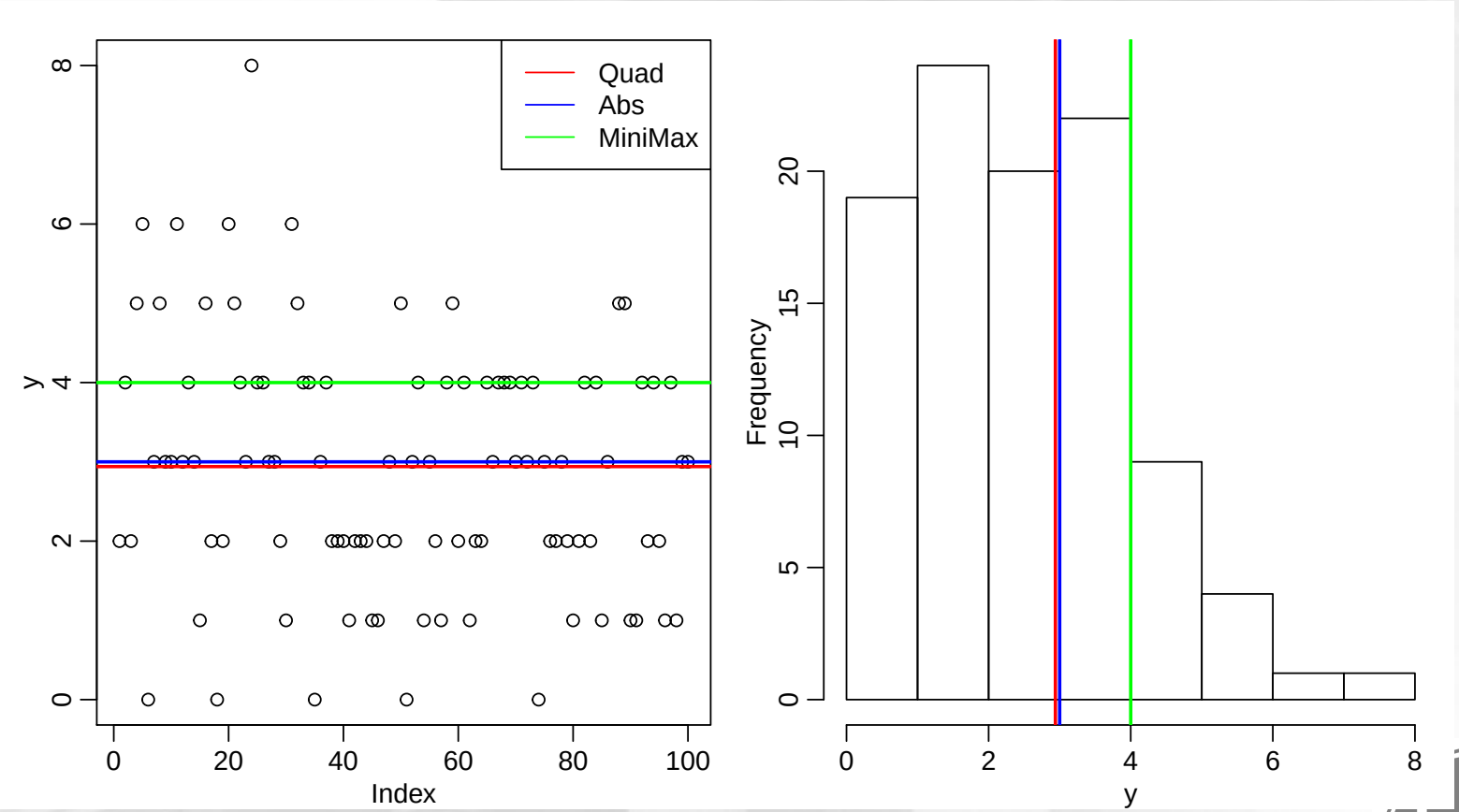
Otimizando funções perda: Redução de dados

- Graficamente, tem-se



Otimizando funções perda: Redução de dados

- Outra forma de visualizar.



Otimização numérica

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

Programação não-linear

- Os métodos são em geral categorizados baseado na dimensionalidade
  - Unidimensional: Golden Section search.
  - Multidimensional.
- Caso multidimensional, tem-se pelo menos quatro tipos de algoritmos
  - Não baseados em gradiente: Nelder-Mead;
  - Baseados em gradiente: Gradiente descendente e variações;
  - Baseados em hessiano: Newton e quasi-Newton (BFGS);
  - 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.
- Existem uma infinidades de variações e implementações.

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.

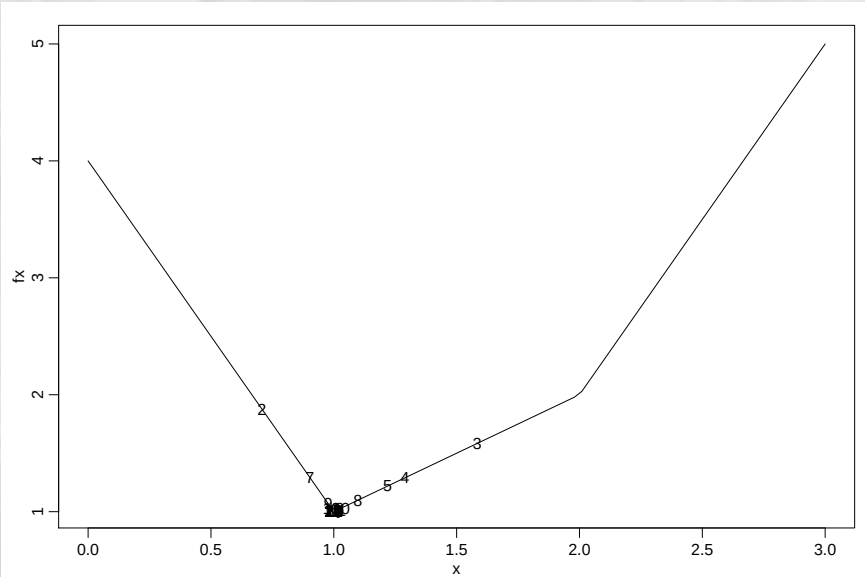
```
xx <- c()
fx <- function(x) {
  out <- abs(x-2) + 2*abs(x-1)
  xx <<- c(xx, x)
  return(out)
}
out <- optimize(f = fx, interval = c(-3,3))
out

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

Exemplo: Otimização unidimensional

- Traço do algoritmo.

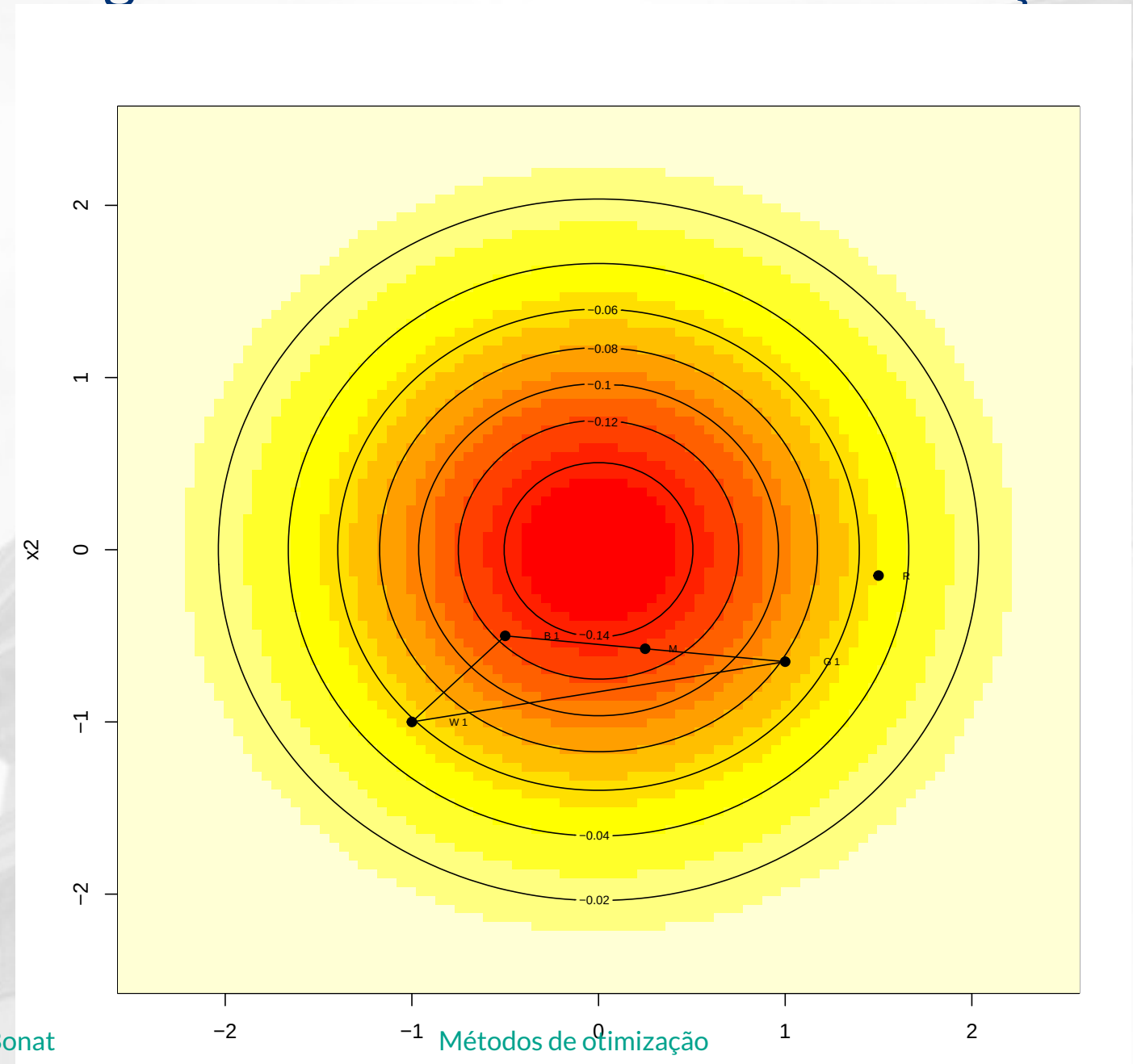
```
par(mfrow = c(1,1), mar=c(2.6, 3, 1.2, 0.5), mgp = c(1.6, 0.6, 0))
fx <- function(x) abs(x-2) + 2*abs(x-1)
plot(fx, 0, 3)
for(i in 1:length(xx)) {
  text(x = xx[i], y = fx(xx[i]), label = i)
}
```



Método de Nelder-Mead (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):

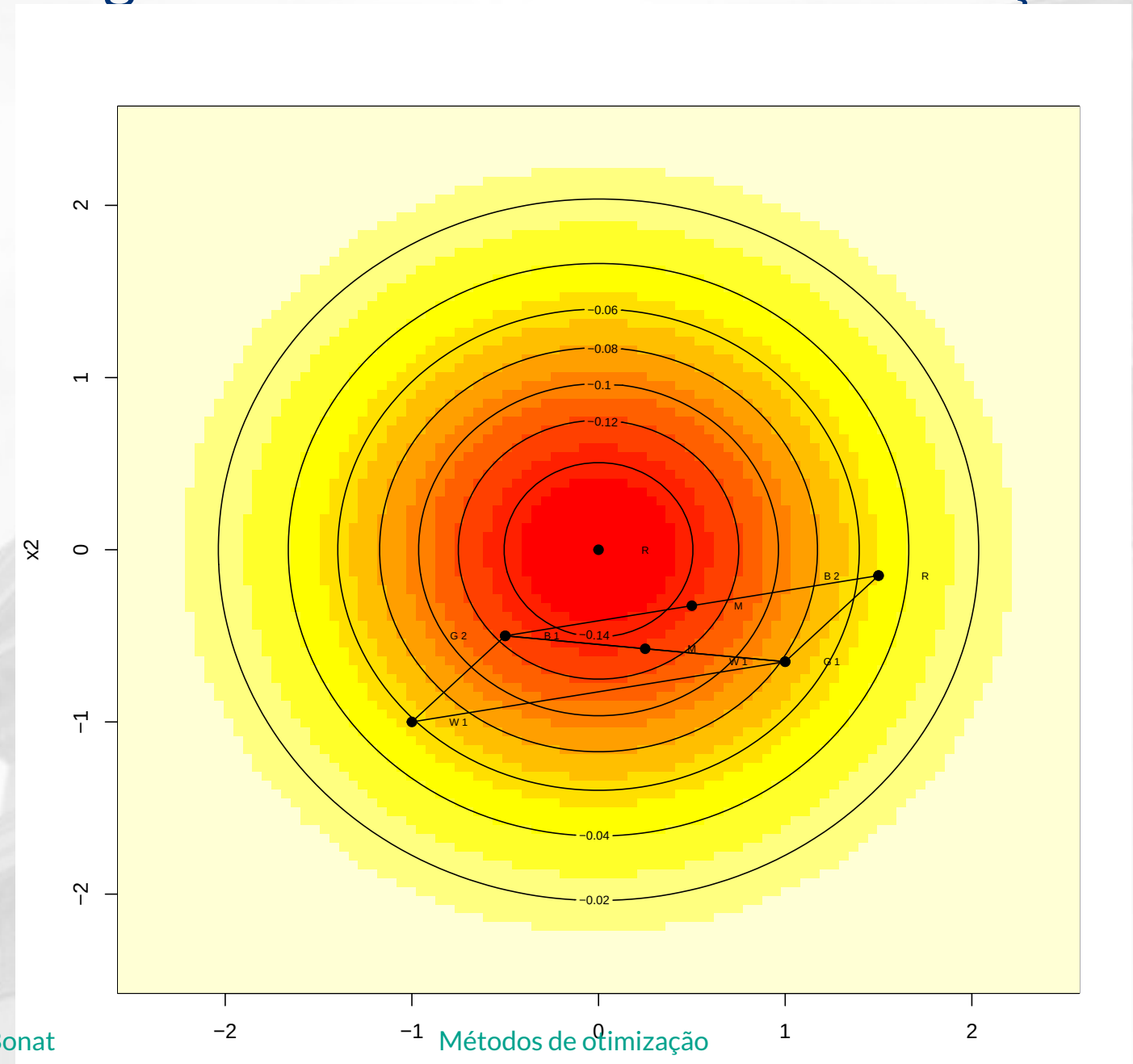
$$M = \frac{B+G}{2} = \left(\frac{x_1+x_2}{2}, \frac{y_1+y_2}{2}\right).$$

- 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.

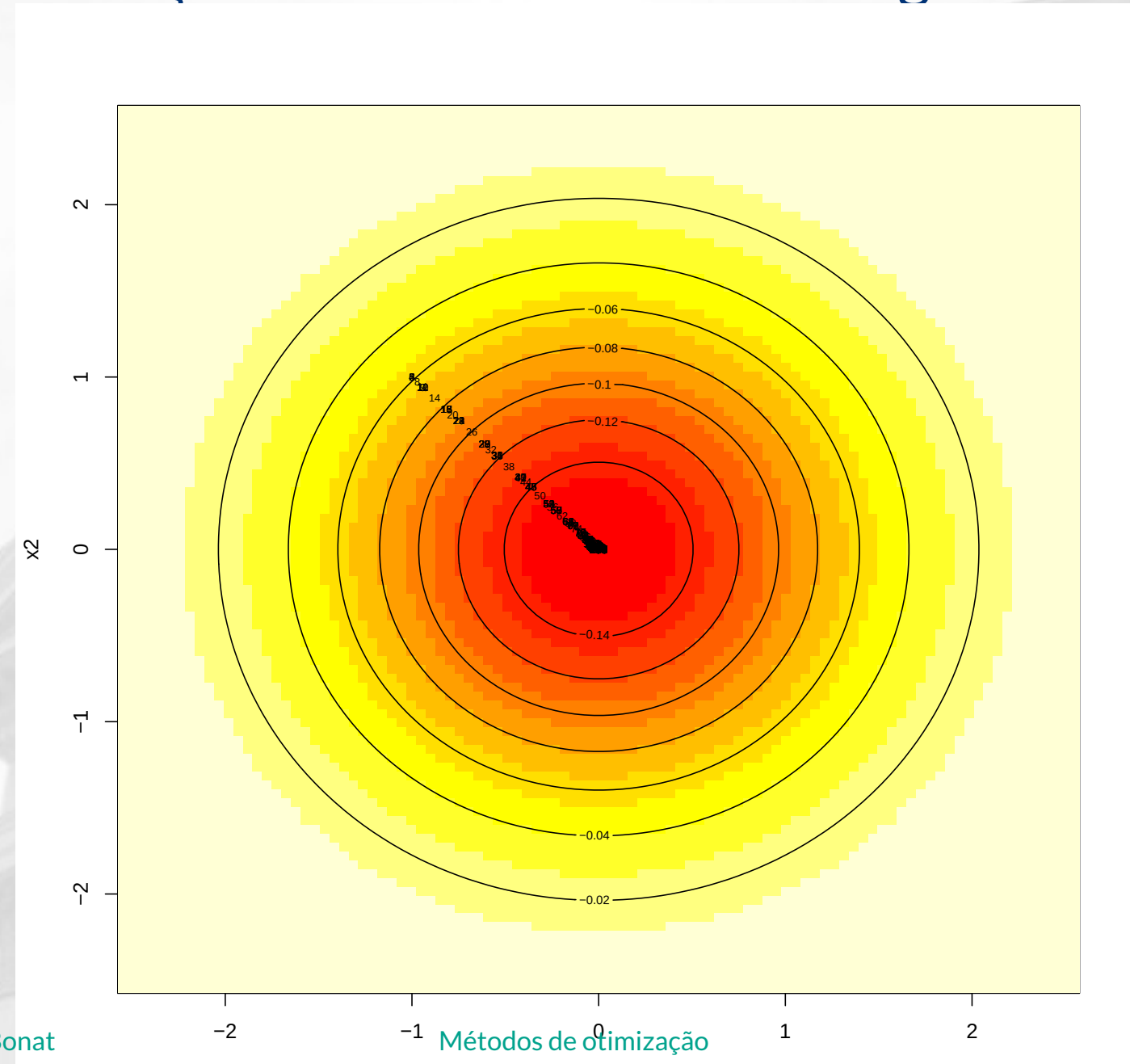
Algoritmo de Nelder-Mead: Ilustração



Métodos 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



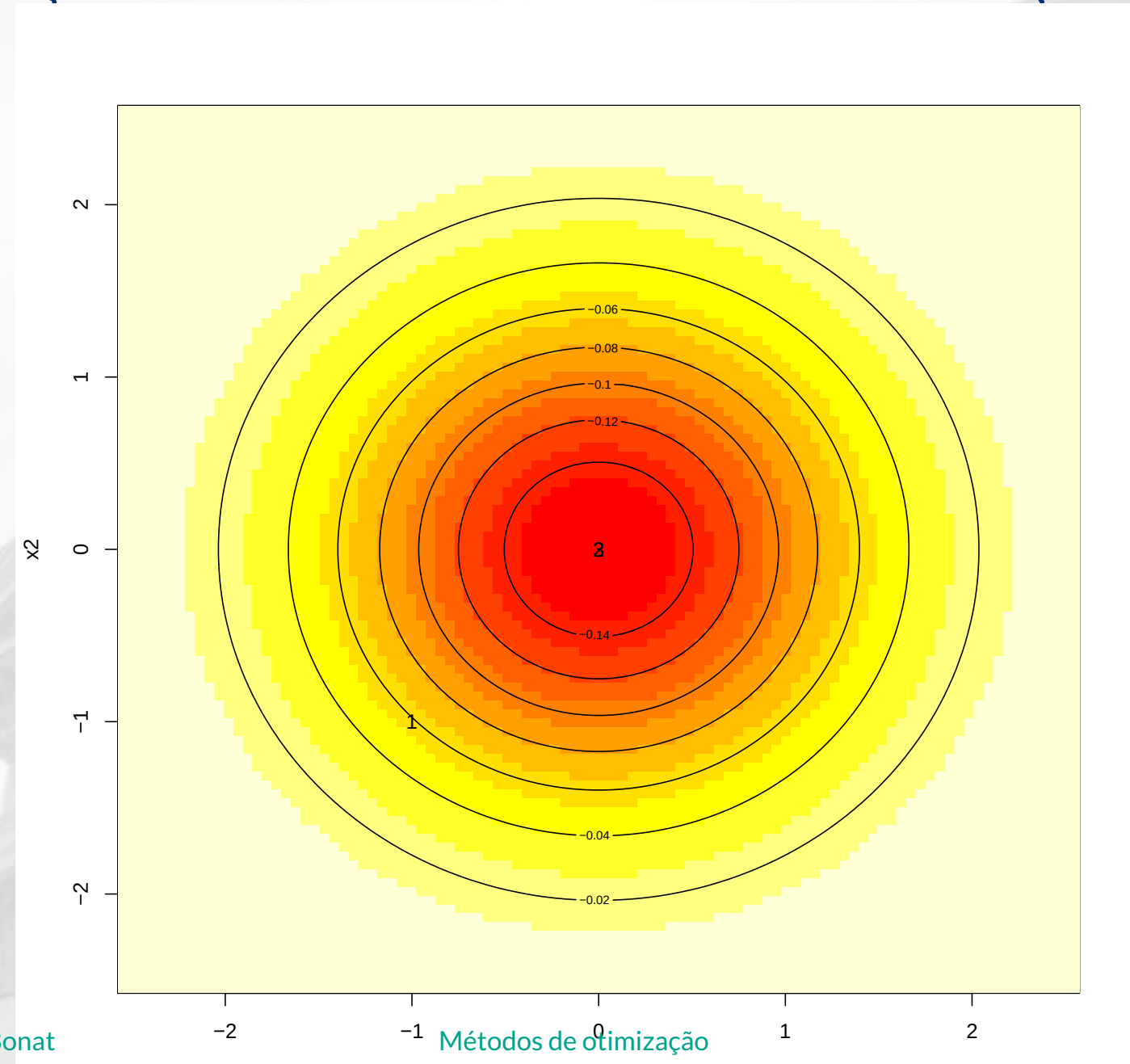
Métodos baseado em hessiano

- 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.
- 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 $\mathbf{H}_{i+1}$ 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

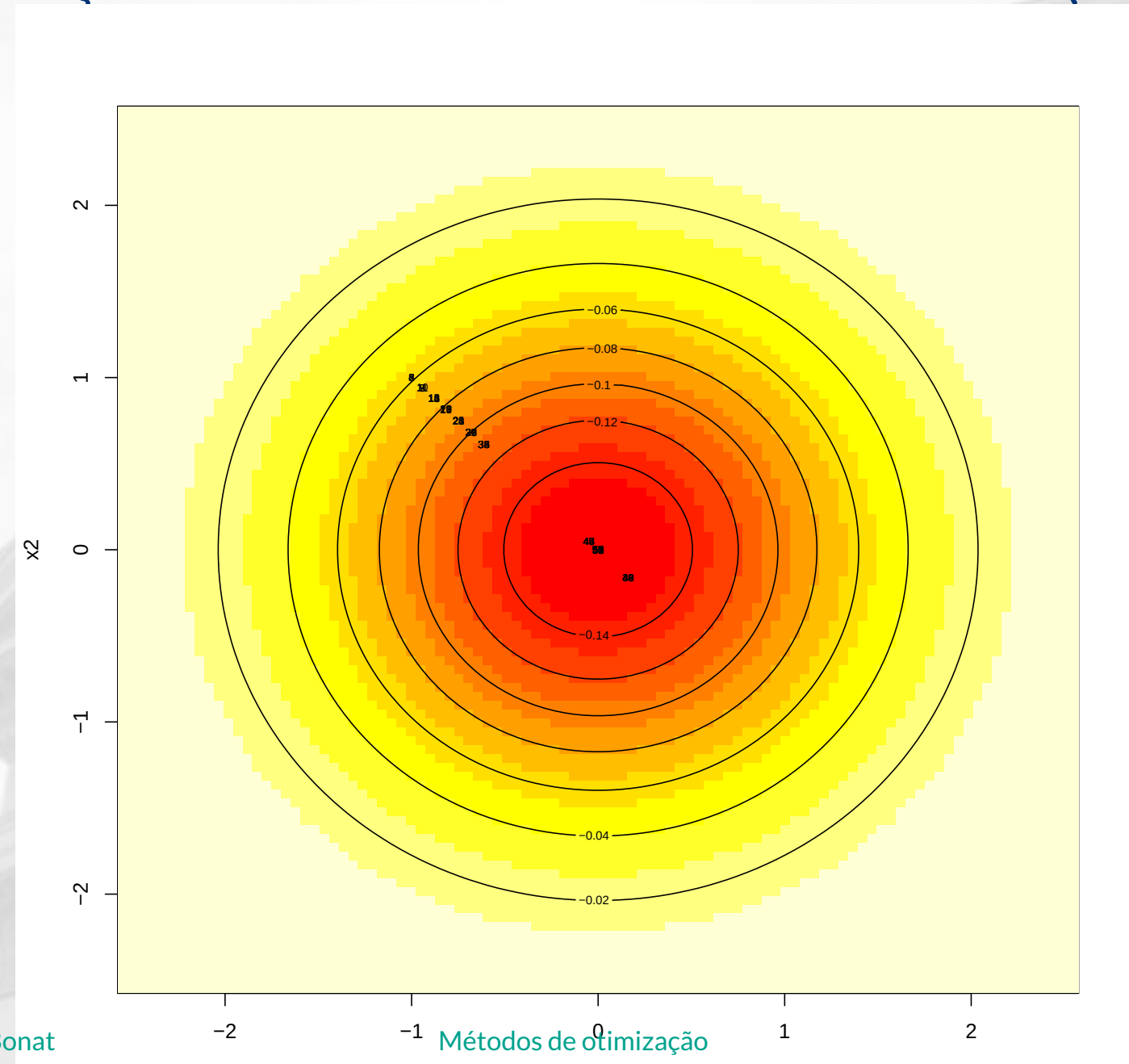
$$\mathbf{H}_{i+1} = \mathbf{H}_i - \frac{\delta_i \gamma_i^\top \mathbf{H}_i + \mathbf{H}_i \gamma_i \delta_i^\top}{\delta_i^\top \gamma_i} + \left(1 + \frac{\gamma_i^\top \mathbf{H}_i \gamma_i}{\delta_i^\top \gamma_i}\right)\left(\frac{\delta_i \delta_i^\top}{\delta_i^\top \gamma_i}\right).$$

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



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 um atingir algum critério de convergência ou número máximo de iterações.

Métodos baseado em simulação: Simulating annealing

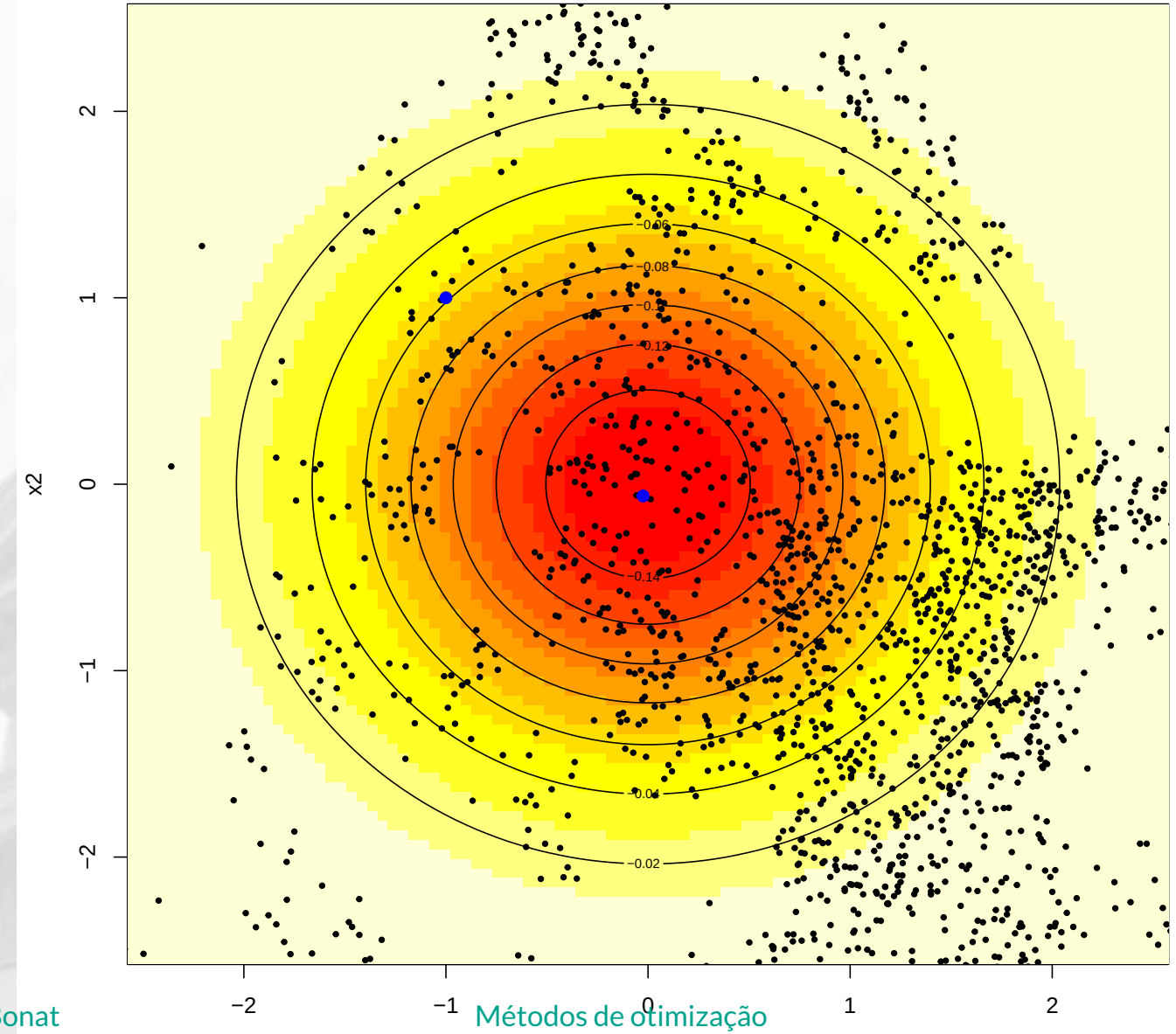
- Para decidir se um ponto x_2 quando $f(x_2) < f(x_1)$ será aceito, usa-se uma probabilidade de aceitação

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

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 probilidade 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 eficiente, 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 lentos e portanto caros computacionalmente.

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)}
```

- Comparando os diversos algoritmo descritos.

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

##                         p1            p2      value fevals gevals niter convcode kkt1 kkt2
## BFGS         -1.772901e-06  1.772901e-06 -0.1591549     13     11    NA         0 TRUE TRUE
## Nelder-Mead   1.134426e-04 -1.503306e-04 -0.1591549     55     NA    NA         0 TRUE TRUE
## CG           -8.423349e-06  8.423349e-06 -0.1591549     97     49    NA         0 TRUE TRUE
##              xtime
## BFGS         0.005
## Nelder-Mead 0.004
## CG           0.017
```

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.

Aplicação: Regressão logística

- Suponha que temos um conjunto de 200 usuários com as informações.
  - Salário em reais.
  - Anos de experiência.
  - Paga conta *premium* (SIM ou NÃO).
- Construa um modelo para avaliar se um usuário **novo** será ou não assinante de uma conta *premium*.