Programação em R - Exercícios - Lista 2

Carlos Cinelli

Julho, 2016

1. Considere os seguintes objetos:

```
set.seed(321)

mat <- matrix(rchisq(100, df = 2), ncol = 10)

df <- data.frame(grupo = rep(c("a","b"), 5),
                 col1 = rbinom(10, 1, 0.5),
                 col2 = rcauchy(10))

lista <- lapply(1:10, function(x) matrix(rnorm(100), ncol = 10))
```

- Em `mat`, calcule as somas e as médias, por linhas e por columas, utilizando `apply` e compare o resultado com as funções `rowSums`, `colSums`, `rowMeans` e `colMeans`. **Extra**: em `mat`, calcule a média móvel com janela de 5 observações, por linha e por coluna, utilizando a função `rollapply` do pacote `zoo` (não vimos essa função em aula, olhe a ajuda em `?rollapply`).
- Em `df`, utilize `lapply` somente nas colunas numéricas para calcular a função `sum(x^2)/length(x)`. Faça isso tanto definindo a função fora do apply, quanto definindo a função dentro do próprio `lapply` como uma fução anônima.
- Em `lista` utilize `lapply` para calcular os determinantes, máximo e mínimo (ao mesmo tempo) de cada matriz dentro da lista.

2. Filtrando e aplicando funções em data.frames:

```
set.seed(10)

df <- data.frame(x = rnorm(100),
                 y = sample(letters, 100, replace =  TRUE),
                 z = sample(LETTERS, 100, replace = TRUE),
                 a = rnorm(100))
```

- Crie um vetor chamado `numericas` usando o `sapply()` para descobrir quais colunas do `data.frame` são numéricas. Utilize o vetor `numericas` para selecionar apenas as colunas numéricas do `data.frame` e use novamente o `sapply()` para calcular os desvios-padrão dessas colunas.
- Agora use a mesma lógica da questão anterior para filtrar apenas os vetores do tipo factor e aplicar a função `table()` nessas colunas.
- Repita os exercícios anteriores usando `Filter()` no primeiro passo (isto é, no passo de selecionar apenas certas colunas).

3. Vamos criar nossas próprias funções para aplicar em `data.frame`s. Leia a base de dados `wi.venda.rds` com o comando `dados <- readRDS("Dados/wi.venda.rds")`. Considerando esta base, faça:

- Crie uma função que receba um vetor e calcule diversas estatísticas descritivas, como a média, média aparada, mediana, variância, desvio-padrão, mínimo, máximo, alguns quantis (olhe a função `quantile()`) e outras que você quiser. **Extra**: faça com que a função retorne NA se o vetor não for numérico e emita um warning para avisar o usuário (veja a ajuda da função `warning()`).
- Após criar sua função, use `sapply()` ou `lapply()` em conjunto com `Filter()` para aplicar sua função às colunas numéricas da nossa base de dados.

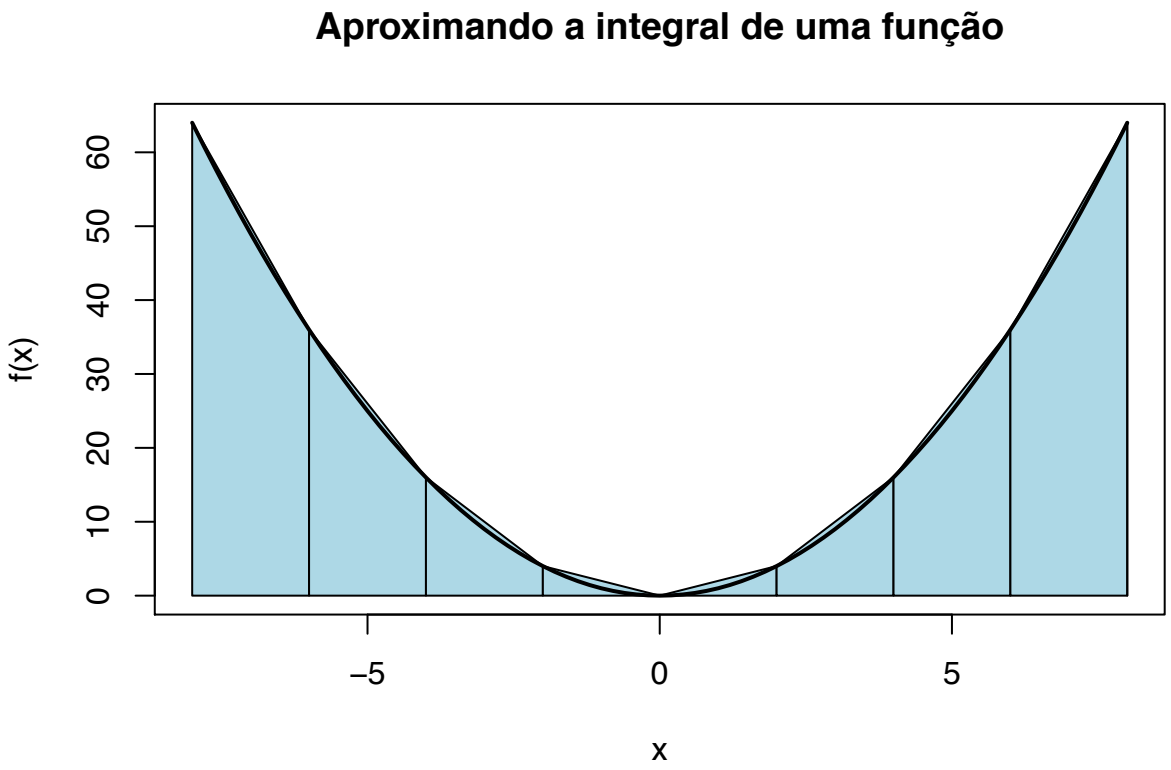
- Agora crie dois `data.frame`s separados `dados_asa_sul` e `dados_asa_norte` com os dados somente da Asa Sul e da Asa Norte, respectivamente. Aplique sua função nesses `data.frame`s e compare os resultados dos dois bairros.

EXTRA - Exercitando a implementação de algoritmos

4. Criando algumas funções:

- Seja n um número inteiro. É possível linearizar o quadrado de n por meio do sequinte algoritmo: $n^2 = 1 + 3 + ... + 2n - 1$. Crie uma função de argumento `n` que gere o quadrado de `n` por meio do algoritmo mencionado. Compare seu resultado com `n^2`. Agora faça com que a função verifique se o argumento `n` é um número redondo e, se não for, faça com que retorne um erro (Dica: veja a ajuda da função `stop()`).
- Crie uma função, usando um loop, que encontre o produtório de um vetor. Compare o resultado de sua função com a função `prod()` do R. **Extra**: agora dê uma olhada na ajuda da função `Reduce()`. Reescreva sua função sem usar loop, usando apenas o `Reduce()`.
- Crie uma função, usando um loop, que encontre o somatório acumulado de um vetor. Compare o resultado de sua função com a função `cumsum()` do R.
- A sequência de fibonacci é definida por: $fib_1 = 0$, $fib_2 = 1$, $fib_3 = 1$, $fib_3 = 2$, $fib_i = fib_{i-1} + fib_{i-2}, \forall i \geq 3$. Crie uma função `fib(n)`, usando um loop, que calcule o enésimo termo da sequência de fibonacci. Faça com que função retorne um erro se o argumento passado não for um inteiro. (Dica: para retornar uma mensagem de erro use a função `stop()`).

5. Aproximando o valor de uma integral: considere uma função qualquer `f(x)`. Suponha que você queira calcular a área embaixo desta função (a integral). É possível aproximar esta área somando pequenos trapézios, como ilustrado na figura a seguir:



Com base nisso, crie uma função no R chamada `integral(a, b, k, f)` que:

- crie uma sequência `x` que vai de a até b em k intervalos equidistantes de tamanho $\frac{b-a}{k}$ (Dica: use a função `seq(a, b, by = (b-a)/k)`).

- crie um vetor `fx` com o resultado da função `f()` em cada ponto da sequencia x criada anteriormente (lembre que as funções do R são vetorizadas!).
- Calcule as áreas dos trapézios embaixo da curva `f()` e que depois some todas as áreas. A área de um trapézio é dada por: $A_i = \frac{f(x_i)+f(x_{i+1})}{2}\frac{(b-a)}{k}$ e a soma das áreas por $A = \sum_i A_i$.
- O resultado da função que acabamos de criar é uma aproximação numérica da integral de $f(x)$ definida no intervalo $x \in (a, b)$, $\int_a^b f(x)dx$. Compare o resultado da sua função coma função `integrate(f, a, b)` do R.

EXTRA - Recursão

Não ensinamos em aula, mas o R também aceita funções recursivas, isto é, funções que podem chamar a si próprias. Um fato importante em definições recursivas é não permitir uma recursão infinita, colocando uma condição de finalização. Por exemplo, a função abaixo seria um implementação recursiva de um fatorial:

```
fatorial <- function(n){
  # condição para evitar recursão infinita
  if(n==1) return(1)
  # recursão
  return(n*fatorial(n-1))
}
fatorial(5) # testando
factorial(5)
```

Com base nisso, tente definir uma função que calcule o enésimo termo de fibonacci usando recursão. Compare o resultado da função recursiva com a função usando *loop*. Qual é mais lenta? O que está acontecendo? Há como resolver isso?