

Parte 1 - Introdução ao R

- R é uma aplicação de distribuição gratuita (<http://cran.r-project.org/>). É um ambiente no qual se pode efectuar análises estatísticas e produzir gráficos e é também uma linguagem de programação.
- R é uma linguagem que manipula determinadas estruturas de dados, designadas objectos. Ao contrário de outras linguagens de programação de nível inferior (FORTRAN, C e Pascal, por exemplo), o R não acede directamente à memória do computador.
- R funciona fundamentalmente pelo modelo “pergunta-resposta”:
 - Os comandos introduzem-se a seguir à prompt (`>`) e são executados após pressionar *Enter* ↵
 - Edição de comandos:
 - Repetir comandos ou navegar entre comandos ↑ ou ↓
 - Percorrer a linha de comandos ← ou →
 - Colocar o cursor no início / fim da linha de comandos
 - Home / End

Comandos elementares:

Expressões aritméticas - são calculadas, o seu valor é impresso mas não é guardado.

```
> 2+3/4*7^2
[1] 38.75
> exp(-2)/log(sqrt(2))
[1] 0.3904951
```

Atribuições - calcula o valor de uma expressão e guarda-o numa variável mas o resultado não é impresso.

> x <- 2

↑ ↙
variável Operador de atribuição

```
> x
[1] 2
```

```
> x <- 2+3/4*7^2
> x
[1] 38.75
```

Nota: os nomes das variáveis podem conter letras (maiúsculas são diferentes de minúsculas), algarismos e pontos. Recomenda-se que os nomes das variáveis comecem por uma letra. Alguns nomes são utilizados pelo sistema, pelo que devem ser evitados (por ex., `c`, `q`, `t`, `C`, `D`, `F`, `I`, `T`, `diff`, `df`, `pt`).

Ex: precipitacao.mar01

Ajuda:

- Para obter informação sobre uma função específica, por exemplo a função `quantile()`, o comando é
> **help(quantile)**
> **?quantile**
- Para pesquisar uma sequência de caracteres no help do R o comando é `help.search()`, por exemplo
> **help.search("mean")**
- Para navegar nas páginas de ajuda do R utilizando um *browser* de internet:

menu Help → Html help ou > **help.start()**

1. Objectos

As entidades que o R cria e manipula designam-se Objectos.

Para mostrar os objectos disponíveis numa sessão

> **ls()** ou > **objects()**

Para remover objectos

> **rm(objecto1, objecto2, ...)**

A colecção de objectos disponíveis designa-se *workspace*. No fim de uma sessão pode-se guardar os objectos no ficheiro “.RData” na pasta de trabalho. Estes objectos podem ser carregados na sessão seguinte.

Especificação da pasta de trabalho:

menu File → Change dir...

Para importar um ficheiro de objectos (por exemplo o ficheiro “.RData”):

menu File → Load Workspace...

Exemplo:

1) importar o ficheiro

\\prunus\home\cadeiras\compsi\Programacao\precipitaISA.RData

2) ver os objectos **ls ()**

Alguns objectos

- *Vector*
- *Matrix*
- *List*
- *Data frame*
- *Function*

Nota: Deve distinguir-se os objectos do R *vector* e *matrix* das entidades matemáticas vector e matriz. Por exemplo, enquanto que no R é possível somar *vectors* com diferente número de componentes, na matemática esta operação só está definida para vectores com o mesmo número de componentes.

1.1. Vector

É uma colecção ordenada de elementos do mesmo tipo (valores numéricos, lógicos, alfanuméricos...).

Nota: os símbolos especiais NA e NaN designam valor desconhecido e valor não numérico, respectivamente.

Exemplos:

```
> x <- c(5.4, -3.7, 11.2, 0.78, 21.6)
> x
[1] 5.40 -3.70 11.20 0.78 21.60
> y<-c(FALSE, TRUE, TRUE, TRUE, FALSE)
> nomes <- c("Ana","Paulo","Zé")
> nomes
[1] "Ana" "Paulo" "Zé"
> z <- c(5.4, -3.7, 11.2, NA, 21.6)
> sqrt(c(-1,1,2))
[1]      NaN 1.000000 1.414214
Warning message:
NaNs produced in: sqrt(c(-1, 1, 2))
```

Para atribuir nomes às componentes de um vector usa-se a função `names()`.

Exemplo:

```
> idades<-c(17,19,18)
> idades
[1] 17 19 18
> names(idades)<-nomes
> idades
  Ana Paulo   Zé
  17   19   18
```

Concatenação de vectores

`c()` é a função de concatenação. `c(v1, v2)` devolve a concatenação de `v1` e `v2`. A função pode ter qualquer número de argumentos.

Exemplos:

```
> c(x, 0, x)
[1] 5.40 -3.70 11.20 0.78 21.60 0.00
5.40 -3.70 11.20 0.78 21.60

> nomes<-c(nomes, "Manel")
> nomes
[1] "Ana" "Paulo" "Zé" "Manel"
```

Operações com vectores numéricos

- operadores aritméticos: +, -, *, /, ^, %% (resto da divisão inteira), %/% (quociente da divisão inteira)

Exemplos:

```
> v1<-c(5, 7)
> v2<-c(10, 11, 12, 13)
> 1/v1
[1] 0.2000000 0.1428571
> v1+v2 #o vector de menor dimensão é
repetido até ficar com o mesmo número de
elementos do maior
[1] 15 18 17 20
> 6%%3 #resto da divisão inteira de 6 por 3
[1] 0
> 6%%4
[1] 2
> 6%/%3 #quociente da divisão inteira de 6
por 3
[1] 2
> 6%/%4
[1] 1
```

- funções aritméticas

`log`, `exp`, `sin`, `cos`, `tan`, `atan`, `sqrt`, `abs`, ...

`length` devolve o número de elementos de um vector

`max/min` devolve o maior/menor elemento de um vector

`range` devolve o vector `c(min, max)`

`sum` devolve a soma dos elementos de um vector

`prod` devolve o produto dos elementos de um vector

`mean` devolve a média dos elementos de um vector

`var` devolve a variância dos elementos de um vector; `var(x)` é igual a `sum((x-mean(x))^2)/(length(x)-1)`

`sd` devolve o desvio padrão dos elementos de um vector; `sd(x)` é igual a `sqrt(var(x))`

As funções `max`, `min`, `range`, `sum`, `prod`, `mean`, `var`, `sd` podem ter vários argumentos. A sua forma pode ser

`função(x, na.rm=FALSE)`

em que `na.rm` é uma variável lógica que indica se os valores desconhecidos devem ser ou não ignorados. Caso existam valores NA em algum dos argumentos, se `na.rm=FALSE` (por omissão) a função devolve NA ou uma mensagem de erro, se `na.rm=TRUE` os valores NA são ignorados.

Exemplos:

```
> z
```

```
[1]  5.4 -3.7 11.2  NA 21.6
```

```
> mean(z)
```

```
[1] NA
```

```
> mean(z, na.rm=TRUE)
```

```
[1] 8.625
```

Exercício 1:

a) calcular a média, a variância e o desvio padrão dos valores da precipitação no mês de Março de 2001 (vector

`precipitacao.mar01`) e atribuir os resultados ao vector `indicadores.mar01`.

b) atribuir os nomes “media”, “var” e “dp” às componentes do vector `indicadores.mar01`.

Geração de sequências regulares

O R permite gerar sequências de valores numéricos

```
> 1:15      # é o vector c(1,2,...,14,15)
[1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
> 2*1:15
[1]  2  4  6  8 10 12 14 16 18 20 22 24 26 28 30
> (2*1):15
[1]  2  3  4  5  6  7  8  9 10 11 12 13 14 15
> seq(1,15)
[1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
> seq(to=15,from=1)
[1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
> seq(-5,5,by=0.5)
[1] -5.0 -4.5 -4.0 -3.5 -3.0 -2.5 -2.0 -1.5 -1.0 -0.5
[11]  0.0  0.5  1.0  1.5  2.0  2.5  3.0  3.5  4.0  4.5
[21]  5.0
> seq(length=21, from=-5, by=0.5)
[1] -5.0 -4.5 -4.0 -3.5 -3.0 -2.5 -2.0 -1.5 -1.0 -0.5
[11]  0.0  0.5  1.0  1.5  2.0  2.5  3.0  3.5  4.0  4.5
[21]  5.0
> rep(1,10)
[1] 1 1 1 1 1 1 1 1 1 1
```

Exercício 2:

a) criar o vector `mult3` com os múltiplos de 3 inferiores a 40.

b) criar o vector `indices` com os valores 1,2,3,... e com o mesmo número de componentes do vector `mult3`.

Operações com vectores lógicos

- operadores relacionais: `<`, `<=`, `>`, `>=`, `==`, `!=`

Exemplos:

```
> x
[1]  5.40 -3.70 11.20  0.78 21.60
> x>13
[1] FALSE FALSE FALSE FALSE  TRUE
• operadores booleanos:
  & conjunção
  | disjunção
  ! negação
```

Exemplos:

```
> x
[1]  5.40 -3.70 11.20  0.78 21.60
> x>13 | x<5
[1] FALSE TRUE  FALSE TRUE  TRUE
> z
[1]  5.4 -3.7 11.2  NA 21.6
> !is.na(z)
[1]  TRUE  TRUE  TRUE FALSE  TRUE
```

Operações com vectores alfanuméricos

- concatenação de um número qualquer de argumentos em valores alfanuméricos:

```
paste(..., sep = " ", collapse = NULL)
```

Exemplos:

```
> paste("Today is", date())
[1] "Today is Wed Oct 20 12:09:46 2004"
> paste(c("Ana","Maria","Silva"),collapse=" ")
[1] "Ana Maria Silva"
> paste(1:8)
[1] "1" "2" "3" "4" "5" "6" "7" "8"
```

```

> paste("A", 1:6)
[1] "A 1" "A 2" "A 3" "A 4" "A 5" "A 6"
> paste("A", 1:6, sep = "")
[1] "A1" "A2" "A3" "A4" "A5" "A6"
> paste("A", 1:6, sep = "",collapse=" ")
[1] "A1 A2 A3 A4 A5 A6"

```

Indexação e selecção de componentes de vectores

Pode-se seleccionar subconjuntos de elementos de um vector colocando à frente do seu nome um vector de índices entre []. Este vector de índices pode ser de vários tipos:

- vector lógico

são seleccionadas as componentes correspondentes aos valores TRUE no vector de índices, por exemplo

```

> z
[1] 5.4 -3.7 11.2 NA 21.6
> z[z>0]
[1] 5.4 11.2 NA 21.6
> z[!is.na(z)]
[1] 5.4 -3.7 11.2 21.6
> letras<-c("a","b","c","d","e")
> letras[(!is.na(z)) & z>0]
[1] "a" "c" "e"

```

- vector de números inteiros positivos

são seleccionadas as componentes cujo índice pertence ao vector, por exemplo

```

> x
[1] 5.40 -3.70 11.20 0.78 21.60
> x[2]      #ou x[c(2)]
[1] -3.7
> z

```

```
[1] 5.4 -3.7 11.2 NA 21.6
> z[c(1,2,4)]
[1] 5.4 -3.7 NA
```

- Simétrico de um vector de números inteiros positivos
são excluídas as componentes cujo índice pertence ao vector, por exemplo

```
> x[-2]
[1] 5.40 11.20 0.78 21.60
> z[-c(1,2,4)]
[1] 11.2 21.6
```

Exercício 3:

- a) criar o vector `dias` com os valores “dia 1”, “dia 2”, “dia 3”,... e com o mesmo número de componentes do vector `precipitacao.mar01`.
- b) usar o vector `dias` para atribuir nomes às componentes do vector `precipitacao.mar01`.
- c) criar o vector lógico `com.precipitacao` que indica as componentes do vector `precipitacao.mar01` com valores positivos.
- d) visualizar os dias em que choveu
- e) determinar o número de dias em que não choveu
- f) criar um vector por eliminação das componentes NA do vector `precipitacao.mar01`.

1.2. Matrix

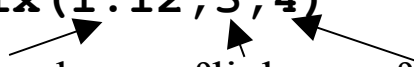
É uma colecção de dados (todos do mesmo tipo) referenciados por dois índices. É uma generalização para duas dimensões de um *vector*.

Nota: quando a dimensão é superior a dois, a colecção é designada *array*. Portanto uma *matrix* é um *array* de dimensão 2.

Uma matriz é definida por um número de linhas (n), um número de colunas (m) e um conjunto de ($n \times m$) valores.

Pode-se definir um objecto do tipo `matrix` através da função `matrix()`, por exemplo

```
> M<-matrix(1:12,3,4)
```



```
> M
```

	[, 1]	[, 2]	[, 3]	[, 4]
[1,]	1	4	7	10
[2,]	2	5	8	11
[3,]	3	6	9	12

Os valores são colocados na *matrix* por coluna

Um objecto do tipo *matrix* tem associado um vector de dimensões com duas componentes, nº de linhas e nº de colunas. A função `dim()` permite ver e definir a dimensão. Por exemplo,

```
> dim(M) # ver a dimensão de M
[1] 3 4
> A<-c(3,2,-4,0,-1,8) # A é um vector
> A
[1] 3 2 -4 0 -1 8
> dim(A)<-c(3,2) # transforma A numa
matrix com 3 linhas e 2 colunas
> A
```

	[, 1]	[, 2]
[1,]	3	0
[2,]	2	-1
[3,]	-4	8

A função `as.vector` devolve um *vector* com os elementos da *matrix*, por exemplo

```
> as.vector(A)
[1] 3 2 -4 0 -1 8
```

Os valores são empilhados por coluna

Indexação e selecção de componentes de matrizes

- $A[i, j]$ é o elemento de A que está na linha i e na coluna j ,

```
> A[3,1]
```

```
[1] -4
```

- $A[i,]$ é a linha i e $A[, j]$ é a coluna j da matriz A . Estes objectos são do tipo *vector*.

```
> A[2,]
```

```
[1] 2 -1
```

```
> A[,1]
```

```
[1] 3 2 -4
```

- Para seleccionar parte de uma linha/coluna indicam-se os índices correspondentes

```
> A[c(1,3),1] # igual a A[-2,1]
```

```
[1] 3 -4
```

- Se a selecção incluir mais do que uma linha e mais do que uma coluna, o resultado é uma submatriz. Por exemplo:

```
> U
```

	[, 1]	[, 2]	[, 3]	[, 4]
[1,]	50	70	80	25
[2,]	73	36	38	37
[3,]	13	76	42	12

```
> U[2:3,c(1,3,4)]
```

	[, 1]	[, 2]	[, 3]
[1,]	73	38	37
[2,]	13	42	12

Operações aritméticas

- Apenas objectos do tipo *matrix*

As operações $+$, $-$, $*$, $/$, $^$, $\%\%$, $\%/ \%$ são realizadas **elemento a elemento**.

Todos os objectos do tipo *matrix* envolvidos na expressão devem ter o mesmo atributo `dim` (isto é, o mesmo número de linhas e de colunas).

Exemplo:

```
> B<-c(1:6)
```

```
> B
```

```
[1] 1 2 3 4 5 6
```

```
> dim(B)<-c(3,2)
```

```
> B
```

```
      [,1] [,2]
[1,]     1     4
[2,]     2     5
[3,]     3     6
```

```
> A
```

```
      [,1] [,2]
[1,]     3     0
[2,]     2    -1
[3,]    -4     8
```

```
> A*B
```

```
      [,1] [,2]
[1,]     3     0
[2,]     4    -5
[3,]   -12    48
```

Atenção: esta operação não é a habitual multiplicação de matrizes definida em Álgebra Linear.

- Mistura de objectos do tipo *matrix* e do tipo *vector*
 - i) Todos os objectos do tipo *matrix* devem ter a mesma dimensão.
 - ii) O número de elementos de cada *vector* deve ser inferior ou igual ao número de elementos de cada *matrix*.
 - iii) Cada *vector* é repetido até ter o mesmo número de elementos de cada *matrix*.

- iv) As operações são realizadas **elemento a elemento** e o resultado tem o atributo `dim` igual ao de cada *matrix*.

Exemplos:

```
> b<-c(1,2)
> 2*A*b # o valor 2 é repetido 6 vezes e o
vector b é repetido 3 vezes
```

```
      [,1] [,2]
[1,]     6     0
[2,]     8    -2
[3,]    -8    32
```

6=2*A[1,1]*1
8=2*A[2,1]*2
-8=2*A[3,1]*1
0=2*A[1,2]*2
-2=2*A[2,2]*1
32=2*A[3,2]*2

Operações com matrizes

O R permite efectuar operações sobre matrizes (entidades matemáticas) através da utilização de dados do tipo *matrix*.

- Matriz transposta: `t(x)` em que x é uma *matrix*

```
> B
      [,1] [,2]
[1,]     1     4
[2,]     2     5
[3,]     3     6
```

```
> C<-t(B)
```

```
> C
      [,1] [,2] [,3]
[1,]     1     2     3
[2,]     4     5     6
```

- Produto de matrizes: `A%*%B` em que A e B são do tipo *matrix* com dimensões da forma `dim(A)=c(n,k)` e `dim(B)=c(k,m)` com n , k e m inteiros positivos. O

resultado (a matriz produto AB) é uma *matrix* com dimensões $c(n, m)$.

Exemplo:

```
> A
      [,1] [,2]
[1,]    3    0
[2,]    2   -1
[3,]   -4    8
> C
      [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
> A %*% C
      [,1] [,2] [,3]
[1,]    3    6    9
[2,]   -2   -1    0
[3,]   28   32   36
> b<-c(1,0)
> A %*% b #o vector b é promovido a matrix
com dimensão 2x1
      [,1]
[1,]    3
[2,]    2
[3,]   -4
> b<-c(1,1,0)
> b %*% A #o vector b é promovido a matrix
com dimensão 1x3
      [,1] [,2]
[1,]    5   -1
```


- Sistemas de equações lineares da forma $Ax=b$

Se A representar uma matriz $n \times n$ invertível e b um vector com n elementos, o comando

```
> solve (A, b)
```

devolve a solução (única) do sistema $Ax=b$.

Exemplo:

```
> M<-matrix(c(1,0,1,2,1,2,1,0,-1),3,3)
```

```
> M
```

```
      [,1] [,2] [,3]
[1,]    1    2    1
[2,]    0    1    0
[3,]    1    2   -1
```

```
> b
```

```
[1] 1 1 0
```

```
> solve(M,b)
```

```
[1] -1.5  1.0  0.5
```

A matriz inversa de A pode ser obtida através do comando

```
> solve (A)
```

Exemplo:

```
> Minv<-solve (M)
```

```
> Minv%*%M
```

```
      [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    1    0
[3,]    0    0    1
```

1.3. List

É uma colecção ordenada de elementos que **podem ser de tipos diferentes**. As componentes de uma lista podem ser vectores numéricos, vectores lógicos, matrizes, listas, funções,...

Exemplo de uma lista:

```
joao<-list(nome="joao", emprego="jardineiro",  
           n.filhos=3, idades=c(2,4,7))
```

Seleccção de componentes de uma lista

As componentes de uma lista são numeradas e podem ser referidas pelo seu número; se tiverem nome, também podem ser referidas pelo nome. Por exemplo,

```
> joao[[2]]  
[1] "jardineiro"  
> joao$emprego  
[1] "jardineiro"
```

Para referir o primeiro elemento do vector *idades*:

```
> joao[[4]][1]  
[1] 2  
> joao$idades[1]  
[1] 2
```

1.4. Data frames

Uma *data frame* pode ser vista como uma *matrix* em que as colunas podem ser de diferentes tipos.

Exemplo:

```
> NotasQ
      [,1] [,2] [,3] [,4]
[1,]  0.1  0.3  0.1  0.4
[2,]  0.0  0.1  0.3  0.1
[3,]  0.5  0.4  0.1  0.4
> Nomes<-c("Ana", "Paulo", "Zé")
> freq<-c(T, F, T)
> df.notas<-data.frame(Nomes, freq, Q=NotasQ)
> df.notas
  Nomes  freq Q.1 Q.2 Q.3 Q.4
1   Ana  TRUE 0.1 0.3 0.1 0.4
2 Paulo FALSE 0.0 0.1 0.3 0.1
3    Zé  TRUE 0.5 0.4 0.1 0.4
```

As *data frames* são casos especiais de listas (com formato de tabela). Em particular as componentes do tipo *vector* devem ter a mesma dimensão e as componentes de tipo *matrix* devem ter o mesmo número de linhas. Cada coluna de uma *matrix* e de uma *data frame*, e cada componente de uma lista origina uma componente (coluna) da *data frame*.

Os elementos de uma *data frame* podem ser referidos indicando a linha e a coluna,

```
> df.notas[2,4]
[1] 0.1
```

A notação `nomeDataFrame$nomeComponente` também pode ser usada,

```
> df.notas$Q.2
[1] 0.3 0.1 0.4    é equivalente a
> df.notas[[4]]
```

Leitura e escrita em ficheiros

- Leitura de dados

Uma tabela de valores pode ser lida para uma *data frame*, através da função

```
read.table(ficheiro, header=FALSE, sep=" ", as.is=FALSE)
```

em que

`ficheiro` é o nome do ficheiro de dados; se este não se encontrar na pasta de trabalho, é necessário indicar o endereço completo;

`header` é uma variável lógica que indica se o ficheiro tem os nomes das variáveis na primeira linha. Por omissão o valor é determinado a partir do formato do ficheiro: `header` toma o valor TRUE se e só se a primeira linha contém menos um campo do que o número de colunas;

`sep` é o carácter que separa os valores em cada linha; por omissão é " " que indica que os valores estão separados por um ou mais espaços em branco.

`as.is` é uma variável lógica. Se o seu valor é TRUE a leitura mantém o tipo original dos dados.

Exemplo:

```
> rga<-read.table("//prunus/home/cadeiras/compsi  
+ /Programacao/RGA99UT.csv", header=T, as.is=T, sep=" ")
```

```
> rga[1:2, 1:5]
```

		UT	EDM	TM	BL	BI
1	Cereais para grão	44914	57288	51856	41645	
2	Leguminosas secas para grão	5274	1153	6390	1641	

- Edição de dados

A função `edit()`, quando aplicada sobre uma *data frame* ou uma *matrix*, permite visualizar e editar os dados num ambiente idêntico a uma folha de cálculo.

Exemplo:

```
> rga.novo<-edit(rga)
```

Edita a *data frame* rga e guarda as alterações na *data frame* rga.novo.

Para criar e introduzir dados numa nova *data frame* num ambiente de folha de cálculo:

```
> Nome<-edit(data.frame())
```

- Escrita de dados

Para guardar num ficheiro o conteúdo da *data frame* x ou *matrix* x usa-se a função

```
write.table(x, file="ficheiro", sep=" ", na="NA",  
            row.names=T, col.names=T)
```

em que

ficheiro é o nome do ficheiro (incluindo o caminho);

row.names, col.names podem ser variáveis lógicas que indicam se o ficheiro contém os nomes das linhas/colunas de x ou podem ser vectores alfanuméricos com os nomes a atribuir às linhas/colunas no ficheiro;

sep é o carácter que separa os valores em cada linha; por omissão é " " que indica que os valores estão separados por um espaço em branco;

na indica a representação no ficheiro dos NA's em x.

```
> df.notas
```

	Nomes	freq	Q.1	Q.2	Q.3	Q.4
1	Ana	TRUE	0.1	0.3	0.1	0.4
2	Paulo	FALSE	0.0	0.1	0.3	0.1
3	Zé	TRUE	0.5	0.4	0.1	0.4

```
> write.table(df.notas, "notas.txt", row.names=F, sep="\t")
```

Exercício 4:

a) Importar o ficheiro de objectos

`\\prunus\home\cadeiras\compsi\Programacao\Exercicio4.RData`

(Nota: o vector `Notas.FC`, contém a cotação obtida por 20 alunos em 10 perguntas de um exercício de folha de cálculo)

b) Transformar o vector `Notas.FC` numa matriz de 20 linhas por 10 colunas, colocando os valores por coluna.

c) Atribuir os nomes `aa20001`, ... , `aa20020` às linhas de `Notas.FC` e os nomes `P1`, ... , `P10` às colunas de `Notas.FC`. (sugestão: utilize a função `dimnames`; `dimnames` de uma matriz `x` é uma lista com duas componentes, um vector com os nomes das linhas de `x` e outro com os nomes das colunas de `x`)

d) Determinar:

- i) a nota máxima do aluno `aa20003`;
- ii) a(s) pergunta(s) em que a nota anterior foi obtida;
- iii) a média das notas na pergunta `P4`;
- iv) a média das notas nas perguntas `P9` e `P10`;
- v) o username dos alunos que obtiveram a cotação máxima (0.4) em `P6`.

e) Construir o vector `total.FC` com as notas finais dos 20 alunos. (Sugestão: utilize a função `apply`)

f) Construir a lista `final.FC` com 3 componentes: as notas finais, a média das notas finais, o username do(s) aluno(s) com a melhor nota.

g) Guardar no ficheiro “`notasFC.txt`” o conteúdo da matriz `Notas.FC`, sem os nomes das colunas e utilizando a vírgula como separador.

2. Estruturas de controlo

Tal como outras linguagens de programação, o R tem estruturas que permitem a execução condicional e a execução repetida de instruções.

2.1. Execução condicional: `if`

O comando `if` apresenta duas formas

```
if(cond) expr  
if(cond) expr_1 else expr_2
```

em que

`cond` é uma condição, isto é, uma expressão cujo resultado é um valor lógico;

`expr`, `expr_1` é a instrução (ou instruções) a ser executada quando `cond` é TRUE;

`expr_2` é a instrução (ou instruções) a ser executada quando `cond` é FALSE.

Nota: Caso `expr_1` ou `expr_2` seja um conjunto de 2 ou mais instruções, estas devem ser colocadas entre chavetas.

Exemplos:

```
> x <- 1  
> if (x<0) -x else x  
[1] 1  
> x <- -3  
> if (x<0) -x else x  
[1] 3
```

2.2. Execução repetida: **for** e **while**

O comando `for` tem a seguinte forma

```
for (var in vect) expr
```

em que

`var` é o nome da variável que controla o ciclo;

`vect` é um vector ou uma expressão cujo resultado é um vector
(habitualmente é uma sequência, por ex. `1:20`);

`expr` é a instrução (ou conjunto de instruções entre chavetas) que
é repetidamente executada à medida que `var` toma
sequencialmente os valores de `vect`.

Exemplo:

```
>for(i in 1:5) print(1:i)
```

A estrutura `while` permite também a execução repetida de uma
ou mais instruções (`expr`) enquanto o valor lógico de uma
condição (`cond`) for verdadeiro,

```
while(cond) expr
```

Exemplo:

```
> i<-3  
> while(i<30){print(i)  
+       i<-i+3}
```

As estruturas de controlo referidas são na maior parte dos casos
utilizadas em conjunto com as funções.

3. Funções definidas pelo utilizador

Os objectos do tipo *function* são objectos de R que implementam funções, e que podem ser usados em expressões, em instruções e na implementação de outras funções.

Uma função é definida por

```
f<-function(arg_1,arg_2,...){corpo da função}
```

em que

f é o nome da função;

arg_1, arg_2, ... são os argumentos da função (podem ser dos tipos *vector, matrix, list, data frame* e *function*).

O corpo da função é uma sequência de instruções em R, que incluem os argumentos *arg_1, arg_2, ...* da função. As instruções do corpo da função são executadas sequencialmente e o valor da função é o resultado da última instrução. Em alternativa, o valor devolvido pela função pode ser identificado na instrução `return(z)`, em que *z* é o objecto que contém o valor da função.

Exemplo:

```
> logistica<-function(x){1/(1+exp(-x))}
```

Uma função pode ser invocada em qualquer expressão, através de `f(valor_arg_1,valor_arg_2,...)`.

```
> logistica(5)  
[1] 0.9933071
```

Outro exemplo:

```
> val.abs<-function(n){if (n<0) -n else n}
```

```
> val.abs(3)
[1] 3
> val.abs(-3)
[1] 3
```

As atribuições de valores a variáveis no interior da função são locais e temporárias. Esses valores são perdidos depois da execução da função estar concluída.

Edição de funções

Uma função pode ser definida em R numa única linha de comandos: nesse caso as instruções no corpo da função são separadas por “;”.

Por exemplo, a seguinte função devolve o número de ocorrências do valor *k* no vector *x*.

```
> conta<-function(x,k){n<-length(x);c<-0;
+ for (i in 1:n){if (x[i]==k) c<-c+1};
+ return(c) }
```

É geralmente mais prático, no entanto, usar um editor para escrever o corpo da função, e organizar as expressões por forma a que a definição da função seja mais facilmente compreensível.

O editor é invocado por `fix`. `fix` é usado para criar ou modificar uma função. Exemplo:

```
> fix(f)           #f é o nome da função
```

Se o corpo da função tem erros de sintaxe quando o editor é “fechado”, R devolve uma mensagem de erro e permite que a última versão da função seja recuperada com `edit()` para que os

erros possam ser corrigidos. Nota: caso se use `fix` de novo em vez de `edit()`, a última versão da função é “perdida”.

```
> fix(f)
```

```
Error in edit(name, file, editor) : An error  
occurred on line 3
```

```
use a command like
```

```
  x <- edit()
```

```
to recover
```

```
> f <- edit()
```

Para visualizar a definição da função faz-se, como é habitual,

```
> f
```

Nomes dos argumentos e valores por omissão

Numa invocação da função os valores dos argumentos podem ser ordenados de qualquer forma, desde que se indique qual o nome do argumento respectivo. Caso se preserve a ordem dos argumentos da definição da função, não é necessário indicar os nomes dos argumentos na invocação da função.

Exemplo:

```
> conta
```

```
function(x, k) {  
  n<-length(x)  
  c<-0  
  for (i in 1:n)  
    if (x[i]==k) c<-c+1  
  return(c)  
}
```

```
> conta(k=5,x=c(3,-2,5,4,5,-10,11,5,7))
[1] 3
> conta(c(3,-2,5,4,5,-10,11,5,7),5)
[1] 3
```

Podem ser atribuídos valores por omissão para os argumentos da função. Caso o valor do argumento não seja indicado na invocação da função, o valor por omissão é utilizado.

```
f<-function(arg_1,arg_2=valor,...){corpo da função}
```

Exemplo: função que itera outra função (f) sobre um determinado valor (x) um certo número de vezes ($n.iteracoes$). Neste exemplo é atribuído 10 como valor por omissão para o argumento $n.iteracoes$.

```
> itera
function (f,x,n.iteracoes=10)
{
  i <- 1
  while (i <= n.iteracoes)
  {
    x <- f(x)
    i <- i+1
  }
  return(x)
}
```

Em seguida, calcula-se a 10^a iteração da função logística (definida atrás) sobre 3:

```
> itera(logistica,3)
[1] 0.6590465
```

Abaixo, calcula-se a 2ª iteração da função `logistica` (definida atrás) sobre 3:

```
> itera(logistica,3,n.iteracoes=2)
[1] 0.7216326
```

Pode-se alterar a ordem dos argumentos:

```
> itera(x=3,f=logistica,n.iteracoes=2)
[1] 0.7216326
```

Exercício 5:

a) Escreva a função `un` que dado um número inteiro positivo n , devolve u_n definido por

$$u_n = \begin{cases} n^2 & \text{se } n < 10 \\ \log(n)/n & \text{se } n \geq 10 \end{cases} .$$

b) i) Escreva a função `int` que dado um número positivo x , devolve o maior inteiro menor ou igual a x .

ii) Escreva a função `q.perfeito` que dado um número inteiro positivo n , devolve `TRUE` se n é quadrado perfeito ou `FALSE` caso contrário. (Definição: n é um quadrado perfeito se e só se a raiz quadrada de n for um número inteiro. Sugestão: utilize a função `int` definida na alínea anterior.)

c) Escreva a função `prog.geom` que dado um número e entre 0 e $1/2$, devolve um vector com os primeiros k termos da progressão geométrica $u_n = (1/2)^n$, $n \in \mathbb{N}_0$, em que k é tal que

$$u_n - u_{n+1} \geq e, \forall n < k-1 \quad \text{e} \quad u_{k-1} - u_k < e .$$

4. Gráficos

Os comandos para criar gráficos dividem-se fundamentalmente em dois grupos:

- funções gráficas de alto nível que permitem criar um novo gráfico;
- funções gráficas de baixo nível que permitem acrescentar informação a um gráfico existente.

Além destas funções gráficas existe também a função `par` que permite controlar uma lista de parâmetros gráficos. A instrução `par()` devolve a lista dos valores dos parâmetros (ver também `?par`).

Funções gráficas de alto nível. Exemplos: `plot`, `barplot`, `pie`

A função gráfica `plot(f, a, b)`, com f uma função¹, traça o gráfico de f entre a e b (por omissão, os valores de a e b são 0 e 1, respectivamente).

Exemplo:

```
> plot(sin)
> plot(sin, -pi, pi)
```

A função `plot` pode também ser utilizada com um ou dois vectores numéricos, `plot(x)` ou `plot(x,y)`, produzindo, no primeiro caso, um diagrama dos valores do vector x versus o seu índice e, no segundo caso, um diagrama de pontos de y versus x .

Exemplo:

```
> plot(precipitacao.mar01)

> un<-function(n) {sin(n)/n}
> plot(un(1:20))

> x<-seq(2,20,2)
> plot(x, un(x))
```

¹ A função f tem de estar definida por forma a que $f(x)$ e x sejam vectores numéricos com o mesmo número, arbitrário, de componentes.

Com a função `plot` podem ainda ser utilizados outros argumentos. O argumento `type` controla o tipo de gráfico produzido. O seu valor pode ser "p" (pontos) (valor por omissão), "l" (linhas), "b" (pontos ligados por linhas), "h" (linhas verticais dos pontos ao eixo das abcissas), "s" e "S" (função em escada).

```
> plot(precipitacao.mar01, type="h")
```

O título, sub-título e as legendas dos eixos num gráfico podem ser criados/alterados através dos argumentos `main`, `sub`, `xlab` e `ylab`, respectivamente. Os extremos dos eixos podem ser definidos com os argumentos `xlim` e `ylim`, cujos valores são vectores de duas componentes.

```
> plot(precipitacao.mar01, type="h",  
+ main="Precipitação", sub="Março 2001",  
+ xlab="Dias", ylab="(mm)", ylim=c(0, 30))
```

Para além dos argumentos já referidos (`main`, `sub`, `xlab`, `ylab`, `xlim` e `ylim`), qualquer outro argumento da função `plot` pode ser utilizado como argumento da função `plot` para modificar as características do gráfico produzido.

A função gráfica `barplot` cria um gráfico de barras.

```
> p<-precipitacao.mar01  
> barplot(p, names.arg=1:length(p), ylab="mm")
```

A função gráfica `pie` cria gráficos do tipo “queijo”.

```
> rga<-read.table("RGA99UT.csv", sep=" ",  
+ as.is=T, header=T)
```

```
> pie(rga[,8], labels=rga[,1], cex=.7)
```

As funções gráficas de alto nível criam, por omissão, um novo gráfico, apagando, se necessário, aquele que se encontra na janela gráfica.

Em alguns casos, a utilização do argumento `add=TRUE` permite sobrepor diferentes gráficos na mesma janela gráfica.

Exemplo:

```
> plot(sin, -pi, pi)
> plot(cos, -pi, pi, add=T, col="red", lty=2)
```

Funções gráficas de baixo nível

Ao contrário das funções gráficas de alto nível, as funções gráficas de baixo nível permitem acrescentar informação a um gráfico existente na janela gráfica.

As funções `points(x, y)` e `lines(x, y)` permitem acrescentar, respectivamente, pontos e pontos ligados por linhas. A função `abline(a, b)` acrescenta uma recta de declive `b` e ordenada na origem `a`.

Exemplo: traçar o gráfico da função seno e da sua recta tangente no ponto $\pi/4$.

```
> plot(sin, -pi, pi)
> x0<-pi/4
> points(x0, sin(x0))
> b<-cos(x0)
> a<-sin(x0)-b*x0
> abline(a, b)
```

Para adicionar ao gráfico linhas verticais e horizontais utilizar

```
> abline(v=0)
```



```
> abline(h=0)
```

A função `legend` permite acrescentar uma legenda ao gráfico.

Exemplo:

```
> plot(sin, -pi, pi)
> plot(cos, -pi, pi, add=T, col="red", lty=2)
> legend(-3, 1, c("sin", "cos"),
+ col=c("black", "red"), lty=1:2)
```

5. Programas em R

Um programa em R é um ficheiro de texto com a extensão “.R”, que contém um conjunto de comandos e que pode ser executado com o comando `source("ficheiro")` ou através de
menu File → Source R code...

Tipicamente, um programa é útil quando se pretende repetir as mesmas tarefas várias vezes ou quando se quer divulgar a outros os comandos necessários para resolver um dado problema.

Um modo simples de criar um programa em R é abrir uma janela do editor de R (menu File → New script), ir escrevendo os diversos comandos e, no final, guardar o ficheiro de texto (menu File → Save). Caso se pretenda ver ou alterar um programa já existente

menu File → Open script...

Exemplo:

1) Correr o programa

```
\\prunus\home\cadeiras\compsi\Programacao\SolucaoI.R
```

2) Visualizar os comandos que constituem o programa anterior.