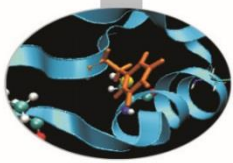
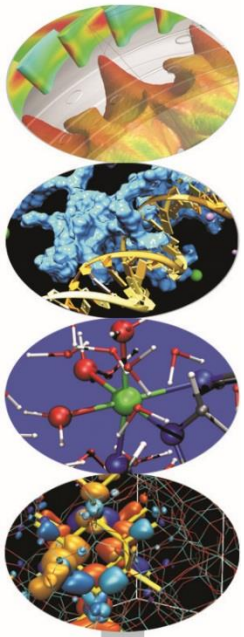
CINECA SCAI
SuperComputing Applications and Innovation



Introducion to R and parallel libraries



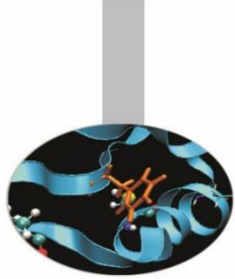
Giorgio Pedrazzi, CINECA-SCAI
Antonio Macaluso, CINECA
Luca Jacopo Avaldi,CINECA

School of Data Analytics and Visualisation

Bologna, 22/06/2016

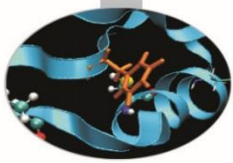
CINECA

Outline



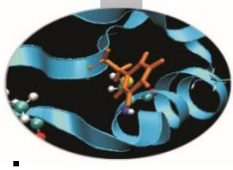
- Overview
  - What is R
  - R Console Input and Evaluation
- Data types
  - R Objects and Attributes
  - Vectors and Lists
  - Matrices
  - Data Frames
- Reading data
  - Reading Tabular Data
  - Reading Large Tables
- Subsetting
  - Lists
  - Matrices
- Grouping, loops and conditional execution
  - Repetitive execution
  - Loops
  - Function and operators
- Parallel libraries
  - Parallel
  - foreach
- R on PICO

What is R



- R is “GNU S” — A language and environment for data manipulation, calculation and graphical display.
  - a suite of operators for calculations on arrays
  - a large, coherent, integrated collection of intermediate tools for interactive data analysis,
  - graphical facilities for data analysis and display
  - a well developed programming language which includes conditionals, loops, user defined recursive functions and input and output facilities.
- The core of R is an interpreted computer language.
  - It allows branching and looping as well as modular programming using functions.
  - Most of the user-visible functions in R are written in R, calling upon a smaller set of internal primitives.
  - It is possible for the user to interface to procedures written in C, C++ or FORTRAN languages for efficiency, and also to write additional primitives.

R Console and evaluation



- At the R prompt we type expressions. The <- symbol is the assignment operator.

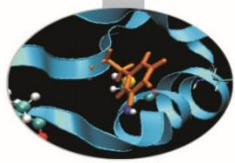
```
> x <- 5
> print(x)
[1] 5
> x
[1] 5
```

- The grammar of the language determines whether an expression is complete or not.

```
> x <- ## Incomplete expression
```

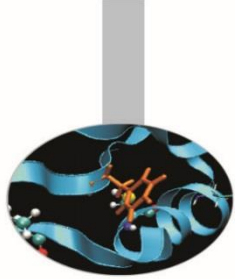
- The # character indicates a comment.
- When a complete expression is entered at the prompt, it is evaluated and the result of the evaluated expression is returned. The result may be auto-printed.

Objects



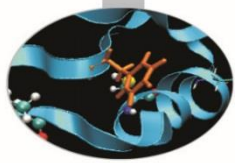
- R has six basic or “atomic” classes of objects
  - character
  - numeric (real numbers)
  - integer
  - complex
  - factor
  - logical (True/False)
- The most basic object is a vector
  - A vector can only contain objects of the same class
  - BUT: The one exception is a *list, which is represented as a vector but can contain objects of* different classes (indeed, that’s usually why we use them)
- Empty vectors can be created with the vector() function.

Attributes



- R objects can have attributes
  - names, rownames, colnames, dimnames
  - dimensions (e.g. matrices, arrays)
  - class
  - length
  - other user-defined attributes/metadata
- Attributes of an object can be accessed using the attributes() function.

Vectors



- The c() function can be used to create vectors of objects.

  > x <- c(0.5, 0.6) ## numeric

  > x <- c(TRUE, FALSE) ## logical

  > x <- c("a", "b", "c") ## character

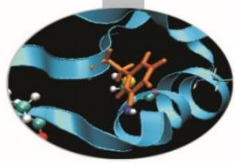
  > x <- 9:29 ## integer

- Using the vector() function

  x <- vector("numeric", length = 10)

- [1] 0 1 2 3 4 5 6

Coercion

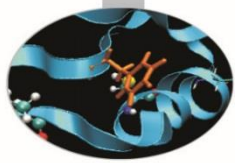


- When different objects are mixed in a vector, *coercion occurs so that every element in the vector is* of the same class.
- Objects can be explicitly coerced from one class to another using the as.* functions, if available.

```
> x <- 0:6
> class(x)
[1] "integer"
> as.numeric(x)
```

- Nonsensical coercion results in NAs.

Lists

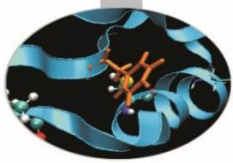


- Lists are a special type of vector that can contain elements of different classes. Lists are a very important data type in R and should be examined carefully.

```
x <- list(1, "a", TRUE, 1 + 4i)
```

- Lists can contain also other lists, matrices, vector or a combination of these
- Lots of the modeling functions (like t.test() for the t test or lm() for linear models) produce lists as their return values

Matrices



- Matrices are vectors with a *dimension attribute. The dimension attribute is itself an integer vector of* length 2 (nrow, ncol)

  >m <- matrix(nrow = 2, ncol = 3)

- Matrices are constructed *column-wise, so entries can be thought of starting in the "upper left" corner* and running down the columns.

  > m <- matrix(1:6, nrow = 2, ncol = 3)

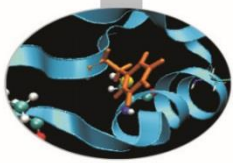
- Matrices can be created by *column-binding or row-binding with cbind() and rbind().*

  > x <- 1:3

  > y <- 10:12

  > cbind(x, y)

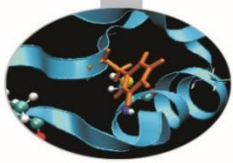
Factors



- Factors are used to represent categorical data. Factors can be unordered or ordered.
- Using factors with labels is *better than using integers because factors are self-describing*
- The order of the levels can be set using the levels argument to factor(). This can be important in linear modelling because the first level is used as the baseline level.

```
> x <- factor(c("yes", "yes", "no", "yes", "no"),
levels = c("yes", "no"))
```

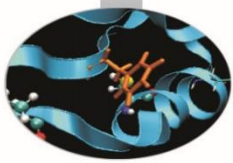
Missing values



- Missing values are denoted by NA or NaN for undefined mathematical operations.
- is.na() is used to test objects if they are NA
- is.nan() is used to test for NaN
- NA values have a class also, so there are integer NA, character NA, etc.
- A NaN value is also NA but the converse is not true

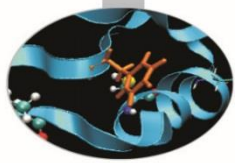
```
> x <- c(1, 2, NA, 10, 3)
> is.na(x)
[1] FALSE FALSE TRUE FALSE FALSE
```

Data Frames



- Used to store tabular data
- Can be considered as a special type of list where every element of the list has to have the same length
- Each element of the list can be thought of as a column and the length of each element of the list is the number of rows.
- Unlike matrices, they can store different classes of objects in each column
- Data frames also have special attributes called **names** and **row.names**
- Data frames are usually created by calling read.table() or read.csv()
- Can be converted to a matrix by calling data.matrix()

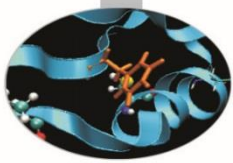
Reading and writing data



- There are a few principal functions reading data into R.
  - read.table, read.csv: tabular data
  - readLines: lines of a text file
  - source: R code files (inverse of dump)
  - dget: R code files (inverse of dput)
  - load: saved workspaces

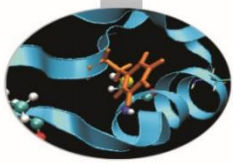
- There are analogous functions for writing data to files
  - write.table
  - writeLines
  - dump
  - dput
  - save

read.table (1)



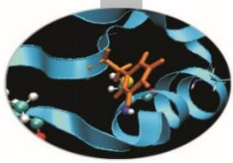
- **read.table** is one of the most used functions for reading data. The most important arguments are:
  - file, the name of a file, or a connection
  - header, logical indicating if the file has a header line
  - sep, a string indicating how the columns are separated
  - colClasses, a character vector indicating the class of each column in the dataset
  - nrows, the number of rows in the dataset
  - comment.char, a character string indicating the comment character
  - skip, the number of lines to skip from the beginning
  - stringsAsFactors, should character variables be coded as factors?

read.table (2)



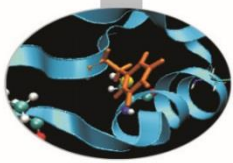
- For small and moderately sized datasets, the function may be called without specifying any other argument
  - data <- read.table("foo.txt")
- R will automatically:
  - skip lines that begin with a # (such setting may be modified)
  - figure out how many rows the file has
  - assign a type to each column of the table
- **read.csv** is identical to read.table, but the default separator is a comma.

Reading larger datasets (1)



- With much larger datasets, the following precautions will make life easier and prevent R from choking.
  - Make a rough calculation of the memory required to store the dataset. If the dataset is larger than the available RAM, no further stops can be executed.
  - Set *comment.char* = "" if there are no commented lines in the file
  - Set quote = “”
  - Hint: **always read the help pages!**
  - Example: ?read.table
  - Once a R package is loaded, it will be possible to read the help page of every function contained in such package

Reading larger datasets (2)



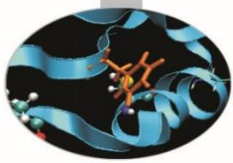
- Know your system
  - In general, when using R with larger datasets, it's useful to know a few things about the system used.
  - How much memory is available?
  - What other applications are in use?
  - Are there other users logged into the same system?
  - What operating system?
  - Is the OS 32 or 64 bit?
- Calculating Memory Requirements
  - Having a data frame with 1,500,000 rows and 120 columns, with numeric data. Roughly, how much memory is required to store this data frame?

    1,500,000 × 120 × 8 bytes/numeric = 1440000000 bytes=

    = 1440000000 /2^20 bytes/MB = 1,373.29 MB=

    = 1.34 GB

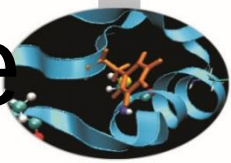
Subsetting



- Some operators can be used to extract subsets of R objects.
  - **[** always returns an object of the same class as the original: it can be used to select more than one element (there is one exception)
  - **[[** is used to extract elements of a list or a data frame; it can only be used to extract a single element and the class of the returned object will not necessarily be a list or a data frame
  - **$** is used to extract elements of a list (or data frame) by name:

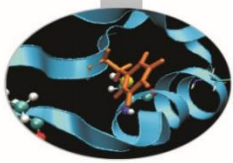
```
> x <- c("a", "b", "c", "c", "d", "a")
> x[1]
[1] "a"
> x[2]
[1] "b"
> x[1:4]
[1] "a" "b" "c" "c"
```

Subsetting a list: an example



```
> x <- list(foo = 1:4, bar = 0.6)
> x[1]
$foo
[1] 1 2 3 4
> x[[1]]
[1] 1 2 3 4
> x$bar
[1] 0.6
> x[["bar"]]
[1] 0.6
> x["bar"]
$bar
[1] 0.6
```

Subsetting a matrix



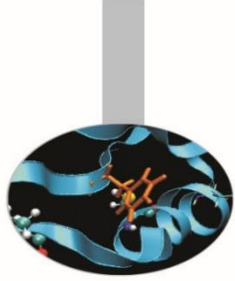
- Matrices can be subsetted in the usual way with *(i,j) type indices.*

```
> x <- matrix(data=1:6, nrow=2, ncol=3)
> x[1, 2]
[1] 3
> x[2, 1]
[1] 2
```

- If the index is missing, the entire row (column) will be selected

```
> x[1, ]
[1] 1 3 5
> x[, 2]
[1] 3 4
```

Removing missing values

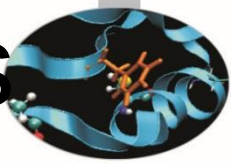


- Missing values (NAs) can be easily removed

```
> x <- c(1, 2, NA, 4, NA, 5)
> bad <- is.na(x)
> x[!bad]
[1] 1 2 4 5

> x <- c(1, 2, NA, 4, NA, 5)
> na.omit(x)
[1] 1 2 4 5
```

Grouping and if() statements



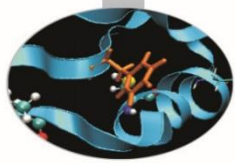
- Grouped expressions
  - R is an expression language in the sense that its only command type is a function or expression which returns a result.
  - Commands may be grouped together in braces, {expr 1, . . ., expr m}, in which case the value of the group is the result of the last expression in the group evaluated.
- **if()** statement
  - The language has available a conditional construction of the form

    if (expr 1) {expr 2}

    else {expr 3}

    where (expr 1) must evaluate to a **logical value**
  - a vectorized version of the if/else construct, the **ifelse** function: this has the form ifelse(condition, yes=a, no=b)

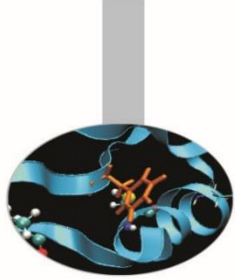
Repetitive executions



- **for()** loops, repeat and while
  - for(name in (expr 1)) {expr 2}

    where *name* is the loop variable, *expr 1* is a vector expression, (often a sequence like 1:20), and *expr 2* is often a grouped expression with its sub-expression. *expr 2* is repeatedly evaluated as name ranges through the values in the vector result of *expr 1*.
- Other looping facilities include the **repeat()** statement and the **while()** statement. The break statement can be used to terminate any loop, possibly abnormally. This is the only way to terminate repeat() loops.
- The next statement can be used to discontinue one particular cycle and skip to the “next”.

Branching

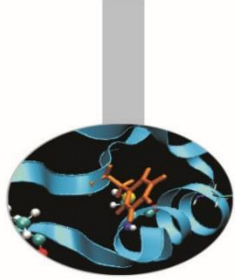


- **if** (logical expression) {
  statements
  } **else** { alternative statements}

- **if** (test_expression1) {
  statement1
  } **else if** (test_expression2) {
  statement2
  } **else** statement3

NB: else branch is optional

Loops

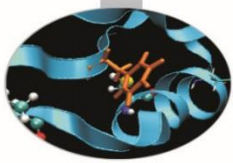


- Very useful when similar tasks need to be performed multiple times
  - Monte Carlo Simulation
  - Cross-Validation (delete one and etc.)

-
```
for(i in 1:10) {
    print(i*i)
     }
```

-
```
i=1
while(i<=10) {
        print(i*i)
        i=i+sqrt(i)
         }
```

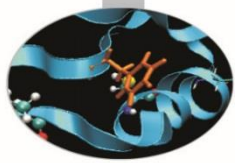
lapply, sapply, apply



- Useful when similar tasks need to be performed multiple times for all elements of a list or for all columns of an array.
  - May be easier and much faster than “for” loops
- lapply(li, *function*)
  - To each element of the list *li*, the function *function* is applied.
  - The result is a list whose elements are the individual *function* results.

```
> li = list("klaus","martin","georg")
> lapply(li, toupper)
> [[1]]
> [1] "KLAUS"
> [[2]]
> [1] "MARTIN"
> [[3]]
> [1] "GEORG"
```

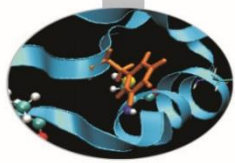
apply



- apply(arr, margin, fun)
- Apply the function *fun* along some dimensions of the array *arr*, according to *margin* (1=rows, 2=columns), and return a vector or array of the appropriate size.

```
> x
     [,1] [,2] [,3]
[1,]    5    7    0
[2,]    7    9    8
[3,]    4    6    7
[4,]    6    3    5
> apply(x, 1, sum)
[1] 12 24 17 14
> apply(x, 2, sum)
[1] 22 25 20
```

Functions and operators

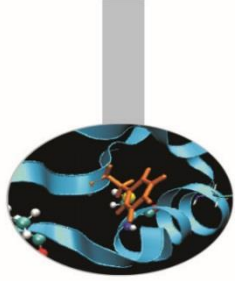


- Functions do things with data
  - “Input”: function arguments (0,1,2,…)
  - “Output”: function result (exactly one)
- Example:

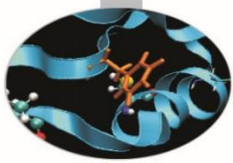
```
add = function(a,b)
{ result = a+b
return(result) }
```

- Operators:
- Short-cut writing for frequently used functions of one or two arguments.
  - Examples: + - * / ! & | %%

Using R on PICO



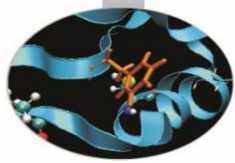
- R can be used within HPC environments
  - PBS Batch jobs: running R using **qsub batch** instructions
  - Interactive PBS Batch jobs: **interactive qsub**
  - Graphical sessions: R & Rstudio via **RCM**
- Some examples will be given using PICO
  - One of Cineca's HPC clusters
  - Made of 74 nodes of different types
    - 54 Compute nodes
    - 4 Visualization nodes
    - 2 Login nodes
    - 14 other nodes
  - 1080 cores available for computational tasks (Batch jobs only)
  - http://www.hpc.cineca.it/content/pico-user-guide



Parallel Computing with R

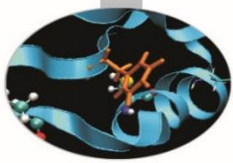
- Under some circumstances, a R job can be speeded up
- Several ways of parallelization are available
- Such methods can be divided in four broad categories:
  - **lapply-based** (*shared memory* and *distributed memory*)
  - **foreach-based** (*shared memory* and *distributed memory*)
  - **Poor man's parallelism**
  - **Hands-off parallelism**
- Parallel jobs with R & MPI based R packages
  - parallel, doParallel, foreach
  - Rmpi,doMPI,foreach
  - Rmpi,snow,snowfall

Parallelization parameters



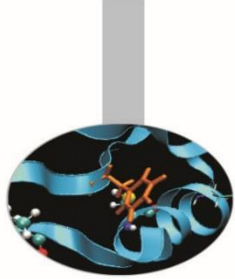
- How many cores?
  - If Rstudio is launched via RCM, only the cores of the visualization node used can be exploited (up to 20)
- How many nodes?
  - If a **qsub** (Batch or Interactive) job is submitted, more than a single computing node can be exploited
  - The job will be queued and scheduled as any PBS Batch job
  - Careful! If too many resources are requested, the priority of the process launched will be lowered
- How much memory?

Parallelization (**qsub** jobs)



- How to manage parallelization in qsub jobs?
  - The resources needed can be directly specified within the code:
    - Number of nodes
    - Number of processors per node
    - Memory needed
    - Maximum job time
  - The number of cores to exploit must also be specified within the R code by using the built-in functions of the R packages mentioned before

PICO login



- Open a SSH client
- Press ENTER
- Connect to Remote Host:
  - Host Name: **login.pico.cineca.it**
  - User Name: **the personal User Name**
  - Port Number: **the default one**
  - Authentication Method: **<Profile Settings>**
- Enter the given Password