INFN ROMA
Istituto Nazionale di Fisica Nucleare
Sezione di Roma

# Introduzione ad 

## Uso ed elementi di programmazione

Francesco Safai Tehrani
*francesco.safaitehrani@roma1.infn.it*

13 Marzo 2018

# Materiale e tre link

Il materiale per questa lezione (inclusa la lezione stessa) si trova a:

https://www.roma1.infn.it/~safai/R/

R si scarica da:

https://www.r-project.org/

Rstudio invece è a:

https://www.rstudio.com/

Infine il sito di Giulio D'Agostini su R:

https://www.roma1.infn.it/~dagos/R/

# Il linguaggio R

R è un linguaggio (ed un ambiente) di programmazione ed è l'implementazione più recente del linguaggio S progettato ai Bell Labs nel 1976[1]

È un linguaggio *general purpose*, ma con una forte inclinazione per la manipolazione, l'analisi statistica e la visualizzazione dei dati.

R nasce concettualmente nel 1992, con un annuncio ufficiale nell'Agosto del 1993, ed una prima release nel 1995, raggiungendo una forma relativamente stabile nel 2000.

La versione corrente è la 3.4.3 (*Kite-Eating Tree*) ed è stata rilasciata il 30 Novembre del 2017.

[1] peraltro, lo stesso posto dove è stato inventato il C nel 1969...

# Uso di R: ambienti interattivi

Una fondamentale ed immediata differenza è nel modo di usare R rispetto a quello che siete abituati a fare per il C.

Il ciclo di sviluppo del C è:

- scrittura del sorgente in un editor di testo (oppure un IDE)
- compilazione / link
- esecuzione / debug

Ogni modifica o correzione richiede una ricompilazione.

Nel caso di R il modo naturale di lavorare, invece, è tramite l'ambiente interattivo. Tutte le operazioni del linguaggio possono essere eseguite in maniera interattiva.

Questo ha il considerevole vantaggio di semplificare la sperimentazione con il linguaggio e la scrittura del codice, che può essere testato passo per passo durante la creazione del programma.

# Documentazione di R ed aiuto on-line

R è un ambiente vasto ed estremamente ricco di funzionalità. Inoltre è facilmente estendibile tramite *package*, ovvero collezioni di funzioni che possono essere importate nell'environment per arricchirlo.

L'aiuto *on-line* è disponibile tramite le funzioni :

`help(argomento)` oppure `?argomento`

oppure in un browser esterno tramite la funzione:

`help.start()`

Il browser punta al server HTTP interno di R per visualizzare la documentazione generale e quella di tutti i *package* installati.

È possibile anche fare una ricerca per argomento usando

`help.search("argomento")`[2] ovvero: `??argomento`

Questa funzione restituisce una lista di funzioni nella cui descrizione compare l'argomento cercato. Per tutti i dettagli: `?help.search`.

[2] notate le virgolette!

# Sessioni

R ricorda lo stato in cui si trovava tra una esecuzione e l'altra, p.e. quali dataset erano definiti, quali funzioni e così via. L'entità logica che contiene questa informazione prende il nome di **sessione** oppure **workspace**.

Alla partenza R cerca alcuni file nella directory corrente (ad esempio, **.RData** e **.Rhistory**), e se li trova, ne carica il contenuto, ricreando la sessione precedente.

È possibile avere differenti directory per differenti sessioni.

Per vedere il contenuto di una sessione si usa la funzione `ls()`, oppure `ls.str()`, per rimuovere definitivamente degli oggetti contenuti nella sessione, la funzione `rm(nome1, nome2, ...)`.

Per uscire da R si usa la funzione `q()` che prima di uscire, chiede se vogliamo salvare la sessione corrente, sovrascrivendo la precedente.

# Package

I package sono collezioni di funzioni che estendono le funzionalità di  .

Esistono numerosi package per  che spesso sono disponibili da repository comuni come, ad esempio, Rforge:
https://r-forge.r-project.org/
oppure CRAN:
https://cran.r-project.org/

Per installare un package, se abbiamo i privilegi adeguati possiamo eseguire `install.packages("pkgname")` che scarica il pacchetto e lo installa.

Per usare un package installato dobbiamo caricarlo, usando `library(pkgname)` che carica `pkgname` oppure `require(pkgname)` che verifica se il package è caricato e non lo è, lo carica.

# Come scrivere codice

L'uso interattivo di R è estremamente semplice ma può essere frustrante se dobbiamo ripetere le stesse operazioni, o correggere delle funzioni complicate, o semplicemente distribuire il codice prodotto.

In questi casi è preferibile scrivere il codice in un editor esterno, magari uno che riconosce la sintassi di R, e poi caricare nell'environment il codice prodotto.

Per caricare il codice contenuto in un file si può usare:

```
source("path completo del file da caricare")
```

oppure:

```
withAutoprint("path completo del file da caricare")
```

che stampa il risultato della valutazione di ogni singolo comando.

È possibile scrivere tutto l'output su un file usando la funzione `sink("file di output")`. Per disabilitare la scrittura su file: `sink()`.

Per tutti i dettagli: `?source` e `?sink`.

# Alcune convenzioni

Alcune caratteristiche di R:

- è *case sensitive* (Funzione != funzione )
- nomi di variabili: tutti i caratteri alfanumerici in aggiunta a ., non possono iniziare con numero
- operatori aritmetici: +, -, *, /
- operatori relazionali: >, <, ==, >=, <=, !=
- operatori logici: !, &, &&, |, ||, xor, TRUE o T, FALSE o F
- ';' per separare istruzioni sulla stessa riga
- parentesi graffe per raggruppare istruzioni: `if (cond) { ist1; ist2; ist3 }`
- costanti "speciali": `1/0` ⇒ `Inf`, `NA` ⇒ Not Available, `NaN` ⇒ Not a Number
- funzioni di arrotondamento: `ceiling(x)`, `floor(x)`, `trunc(x,...)`, `round(x, digits = 0)`, `signif(x, digits = 6)`

# Vettori

I vettori di R sono analoghi agli array del C, salvo che non è necessario preassegnare una dimensione: l'array viene allocato automaticamente secondo necessità. L'indice del primo elemento è 1 invece che 0.

La funzione che costruisce un vettore a partire da un insieme di valori è `c`:

```
> c(2,3,5,7,11,13,17,19) ⇒ [1] 2 3 5 7 11 13 17 19
```

Gli intervalli producono vettori. Si chiama intervallo una coppia di numeri separata dal carattere due punti ':' (la funzione `seq` è più flessibile):

```
> 1:10 ⇒ [1] 1 2 3 4 5 6 7 8 9 10
```

Varie funzioni ed operatori lavorano direttamente sui vettori, incluse quelle definite dagli utenti:

```
> (function(x) x*x)(1:5) => [1] 1 4 9 16 25
```

L'elemento i-esimo di un vettore è `vettore[i]`, ma con la stessa sintassi è possibile selezionare sottoinsiemi di un vettore (p.e. solo gli elementi pari):

```
(1:10)[c(F,T)] => 2 4 6 8 10
```

# Matrici

Le matrici si possono costruire direttamente o a partire dai vettori:

```
vect <- 1:20
```

e “modificando” la struttura del vettore tramite un altro vettore. La funzione `dim` restituisce la dimensione di un oggetto, ma può anche essere utilizzata per cambiarla:

```
dim(vect) <- c(4,5)
```

trasforma il vettore `vect` in una matrice 4x5.

In alternativa si può usare la funzione `matrix` che permette maggiore flessibilità.

# Distribuzioni statistiche

Naturalmente R offre una ricca selezione di funzioni di carattere statistico, ed in particolare:

- `dDIST(x)` => pdf f(x)
- `pDIST(x)` => cumulativa F(x)
- `qDIST(x)` => quantili, ovvero x tale che p = F(x)
- `rDIST(n)` => n numeri pseudo-casuali secono f(x)

dove DIST, ad esempio può essere una di queste
`binom, pois, norm, unif, geom, chisq`

L'iterazione diretta non è il pattern d'uso preferito in R, e tipicamente produce codice più lento dell'uso diretto delle funzioni vettoriali. Nonostante questo, esistono tutte le parole chiave per iterare;

| | |
|---|---|
| iterazioni *bound* | `for (index in sequence) istruzioni` |
| iterazioni *unbound* | `while(condizione) istruzioni` |
| iterazioni *unbound* | `repeat istruzioni` |

Sono disponibili le usuali keyword `break` e `next` con lo stesso significato che hanno in C: `break` esce dall'iterazione corrente, mentre `next` salta al prossimo elemento dell'iterazione.

La keyword `repeat` esegue una iterazione infinita, per cui sta al blocco di codice che segue l'istruzione richiamare `break` per uscire (in C potete fare la stessa cosa con `while(1)...`)

Per eseguire codice condizionale la keyword è:
`if(cond) blocco_true else blocco_false`

# Funzioni

La sintassi per definire una funzione è:

```
nomeFunzione <- function(args) corpoDellaFunzione
```

dove gli argomenti possono essere:

- uno o più nomi di variabile, ovvero argomenti posizionali
- una o più coppie chiave=valoreDiDefault, ovvero argomenti keyword

Gli argomenti posizionali devono sempre ricevere un valore esplicito, mentre i keyword possono essere ignorati (nel qual caso prendono il valore di default), oppure assegnati in maniera posizionale o esplicita.

Le funzioni restituiscono automaticamente l'ultimo valore calcolato, oppure il valore indicato dalla keyword `return`.

# Un esempio di funzione

Ad esempio, definiamo la funzione $retta = a \cdot x + b$:

```
retta <- function(x, a=1, b=0) a*x+b
```

Notate che è disponibile la keyword `return(valore)`, ma se manca, R restituisce automaticamente il valore dell'ultima espressione calcolata nel corpo della funzione. Usando `return` esplicitamente, avremmo potuto scrivere:

```
retta <- function(x, a=1, b=0) return(a*x+b)
```

Ora possiamo invocare `retta` in questi modi:

```
> retta(1)              [1] 1
> retta(1,3,5)          [1] 8
> retta(2, a=20, b=2)   [1] 42
> retta(2, b=20, a=2)   [1] 24
> retta(1, a=7/12)      [1] 0.5833333
> retta(1, b=pi)        [1] 4.141593
```

# Grafici

R fornisce numerosi strumenti per creare rappresentazioni grafiche dei dati, ma quello che ci interessa è come visualizzarli e/o salvarli.

R internamente utilizza il concetto di *device* grafico. Alcuni esempi di device grafici:

- `X11()`, `windows()`, `quartz()`
- `pdf("filename.pdf")`, `png("filename.png")`, ...

Il package `grDevices` contiene tutti i device grafici.

Una volta creato/inizializzato un device grafico, tutto l'output dei comandi grafici seguenti va automaticamente in quel dispositivo. Se il device è un file, sembrerà vuoto fino a quando non eseguiamo `dev.off()` per disabilitare il device grafico.

Esempi di grafici che si possono generare:

- `plot(vettore)` genera un diagramma cartesiano usando come ascissa l'indice e come ordinata `vettore[i]`,
- `plot(v1,v2)` scatter plot di `v1` e `v2`
- `plot(v1 ~ v2)` grafico di `v1` in funzione di `v2`

# Titoli

R offre una grande varietà di funzioni per produrre grafici, ma un grafico senza un titolo o una indicazione di quali variabili siano riportate sugli assi è poco utile (specialmente in una relazione...).

Possiamo quindi usare la funzione `title` con i seguenti argomenti (che possono anche essere specificati nella funzione di plot stessa):

- `main`: il titolo "principale" del grafico
- `sub`: un sottotitolo per il grafico
- `xlab` / `ylab`: label per l'asse delle *x* e per l'asse delle *y*
- `col.<labelname>` permette di definire il colore della label (p.e. `col.main="blue"`)
- `font.<labelname>` permette di definire il font della label

Nelle intestazioni è possibile usare simboli, lettere greche, formule...

È anche possibile configurare gli assi usando la funzione `axis`, che prende come primo argomento il numero dell'asse da configurare (1 per l'asse delle *x*, 2 per l'asse delle *y*).

Infine la funzione `text` permette di aggiungere testo sul grafico.

# Tabelle

Può essere utile produrre delle tabelle a partire dai dati che abbiamo generato dentro R. Questa operazione è decisamente semplice:

supponiamo di avere tre vettori di uguale dimensione: `v1`, `v2`, `v3`.

Possiamo combinare questi vettori in un `data frame` in questo modo:

```
df <- data_frame(v1,v2,v3)
```

I nomi delle variabili nel data frame sono semplicemente i nomi dei vettori (`v1`, `v2`, `v3`), e possono essere modificati tramite la funzione `names`: p.e. `names(df)[2] <- "var2"`.

Come per i grafici è possibile esportare il contenuto di un dataframe selezionando il device di destinazione, ma è anche possibile creare delle tabelle pronte per essere importate in un file LaTeX, usando il package `xtable` in questo modo:

```
print(xtable(df), file="tabella_df.tex")
```

# Un esempio pratico di uso di R

Abbiamo un file di dati: https://www.roma1.infn.it/~safai/R/gaus-data.dat È un file CSV (*comma separated values*), che contiene 10000 misure di tre variabili casuali.

Per leggere un file CSV, si usa la funzione `read.csv("nomefile")` [3], dove nomefile può anche essere un URL. Questa funzione produce un *data frame*. R assume che la prima linea contenga i "nomi" delle variabili e le righe successive le misure.

La variabili contenute in un *data frame* sono accessibili tramite la sintassi `nomeFrame$nomeVariabile`, mentre `str(nomeFrame)` dà la lista delle variabili nel frame.

Per scrivere un file CSV, esiste la funzione `write.csv(...)`[4].

---

[3] `??read` vi dà la lista completa di tutti i formati che R conosce.

[4] `??write` per tutti i formati di output noti

# Un esempio pratico di uso di R *(2)*

- caricate il file della slide precedente in un data frame
- calcolate media campionaria e deviazione standard delle variabili ivi contenute
- generate degli istogrammi per visualizzare il comportamento di queste variabili

Utile da sapere:

R offre queste due funzioni tra quelle di default: `mean` e `sd`,

La funzione per generare un istogramma di una variabile è `hist`. Il numero di classi dell'istogramma è specificato tramite il parametro `breaks`.

Per ottenere un istogramma normalizzato, bisogna specificare l'argomento `prob=T`, mentre per sovrapporre all'istogramma la curva relativa alla distribuzione di probabilità si usa il parametro `add=T` nella funzione `plot`.

Ad esempio:
```
data <- rnorm(10000)
hist(data, prob=T, breaks=30)
plot(dnorm, -5, 5, add=T, col="blue")
```

# Generazione di dati distribuiti secondo una distribuzione normale

Per generare un campione di dati distribuiti secondo una distribuzione normale $\mathcal{N}(\mu, \sigma)$, usando un generatore uniforme di numeri pseudocasuali possiamo usare il seguente algorimo:

- estrarre 12 numeri: $a_i \in [-0.5, 0.5]$
- calcolare $g_k = \sum_{i=1}^{12} a_i$, che segue (più o meno) una distribuzione standard $\mathcal{N}(0, 1)$
- calcolare $x_k = g_k * \sigma + \mu$, che invece segue (sempre più o meno) la distribuzione normale $\mathcal{N}(\mu, \sigma)$

Vogliamo verificare le proprietà statistiche del campione prodotto, in particolare calcolandone la media campionaria e la deviazione standard.

# Generazione di dati distribuiti secondo una distribuzione normale *(2)*

La definizione esplicita di queste grandezze è:

$$\mu_{st} = \sum_{i=1}^{N} \frac{x_i}{N}$$

e la deviazione standard:

$$\sigma_{st} = \sqrt{\sum_{i=1}^{N} \frac{(x_i - \mu_{st})^2}{N-1}}$$

I valori calcolati dovrebbero “somigliare” ai $\mu$ e $\sigma$ “veri” utilizzati per generare il campione.

# Generazione di dati distribuiti secondo una distribuzione normale *(3)*

Cosa dobbiamo fare:

- la funzione nrand(mu, sigma) che restituisce un valore distribuito secondo $\mathcal{N}(\mu, \sigma)$
- creiamo un campione di *N* elementi
- vediamo qualche istogramma e studiamo le proprietà statistiche dei dati

L'unico elemento che ci manca per cominciare è l'equivalente della funzione `rand()` del C. Possiamo usare la funzione:

```
runif(N, min, max)
```

che restituisce un vettore di numeri distribuiti uniformemente tra `min` e `max` di lunghezza `N`.

# Iterazione verso operazioni vettoriali

Abbiamo scritto varie implementazioni delle funzioni che ci interessano, partendo tipicamente da una versione che è una traduzione quasi esatta del codice C, per arrivare a delle versioni un po' più idiomatiche di codice R, usando costrutti vettoriali.

Naturalmente la corretta applicazione di questo approccio richiede comunque uno sforzo ulteriore, come abbiamo visto, ad esempio, tra `makeSample` e `makeSample2`.

In altri casi il cambiamento è concettuale, come ad esempio iterare direttamente sul vettore, invece che su un indice che poi usiamo per accedere al vettore.

R è estensivamente ottimizzato per usare operazioni vettoriali, e l'incremento di prestazioni può essere significativo, ma nulla vi impedisce di partire da una implementazione che vi sembra naturale, per poi raffinarla con tecniche più avanzate.

In questo caso la combinazione tra l'ambiente interattivo e `source` è utilissima.