Introduzione alla Statistica
Computazionale con R

BRUNO BERTACCINI

FIRENZE
UNIVERSITY
PRESS

STRUMENTI
PER LA DIDATTICA E LA RICERCA

– 199 –

Bruno Bertaccini

Introduzione alla Statistica Computazionale con R

Firenze University Press
2018

Introduzione alla Statistica Computazionale con R / Bruno Bertaccini. – Firenze : Firenze University Press, 2018.
(Strumenti per la didattica e la ricerca ; 199)

http://digital.casalini.it/9788864536743

ISBN 978-88-6453-673-6 (print)
ISBN 978-88-6453-674-3 (online)

Il logo di **R** è stato registrato nel 2016 da "The R Foundation".
Il suo utilizzo in questo testo è reso possibile grazie ai termini della licenza internazionale "Creative Commons Attribution-ShareAlike 4.0" (CC-BY-SA 4.0).
https://www.r-project.org/logo

Certificazione scientifica delle Opere
Tutti i volumi pubblicati sono soggetti ad un processo di referaggio esterno di cui sono responsabili il Consiglio editoriale della FUP e i Consigli scientifici delle singole collane. Le opere pubblicate nel catalogo della FUP sono valutate e approvate dal Consiglio editoriale della casa editrice. Per una descrizione più analitica del processo di referaggio si rimanda ai documenti ufficiali pubblicati sul catalogo on-line della casa editrice (www.fupress.com).

Consiglio editoriale Firenze University Press
A. Dolfi (Presidente), M. Boddi, A. Bucelli, R. Casalbuoni, M. Garzaniti, M.C. Grisolia, P. Guarnieri, R. Lanfredini, A. Lenzi, P. Lo Nostro, G. Mari, A. Mariani, P.M. Mariano, S. Marinai, R. Minuti, P. Nanni, G. Nigro, A. Perulli, M.C. Torricelli.

La presente opera è rilasciata nei termini della licenza Creative Commons Attribution 4.0 International (CC BY 4.0: http://creativecommons.org/licenses/by/4.0/legalcode).

This book is printed on acid-free paper

CC 2018 Firenze University Press
Università degli Studi di Firenze
Firenze University Press
via Cittadella, 7, 50144 Firenze, Italy
www.fupress.com
Printed in Italy

Oltre
Oltre voi due ci sono io,
che vi coccolerò per sempre.
sul Liburna, il 25 giugno 2017

Indice

Premessa 1

CAPITOLO I
Cosa è R 5

CAPITOLO II
Perché usare R? 7

CAPITOLO III
Installazione e layout di R 9

CAPITOLO IV
Pacchetti aggiuntivi 11

CAPITOLO V
Una calcolatrice estremamente potente 13

CAPITOLO VI
Personalizzazione delle funzioni 19

CAPITOLO VII
Strumenti grafici per lo studio di funzioni 23

CAPITOLO VIII
Vettori 29

CAPITOLO IX
Matrici e array 37

B. Bertaccini, *Introduzione alla Statistica Computazionale con R*, ISBN (print) 978-88-6453-673-6,
ISBN (online) 978-88-6453-674-3, CC BY 4.0, 2018 Firenze University Press.

CAPITOLO X
Liste 47

CAPITOLO XI
Dataframe 49

CAPITOLO XII
Tipi di variabili 59

CAPITOLO XIII
Elementi di programmazione strutturata 63

CAPITOLO XIV
Loop functions: le funzioni della famiglia apply 71

CAPITOLO XV
Iterazioni VS programmazione ricorsiva 81

CAPITOLO XVI
Principali funzionalità per la descrizione statistica delle informazioni 85

CAPITOLO XVII
Generazione di numeri (pseudo) casuali 95

CAPITOLO XVIII
Elementi di calcolo combinatorio 103

CAPITOLO XIX
Funzioni per il trattamento delle distribuzioni di Variabili Casuali 109

CAPITOLO XX
Simulazioni Monte Carlo 121

CAPITOLO XXI
Selezione di esercizi svolti 127

Bibliografia essenziale 165

Premessa

Le premesse dei testi didattico / divulgativi sono sovente caratterizzate dall'elencazione delle motivazioni che hanno indotto gli autori a proporre la propria opera. Cercando allora di soffocare la mia inguaribile natura anarchica che mi rende tendenzialmente allergico ad ogni forma di convenzione, e mettendomi nei panni di uno studente - magari del mio corso - che vorrà utilizzare questo testo, ho sinceramente, e più volte, rovistato nella mia mente alla ricerca di tali motivazioni. Purtroppo ancora oggi non riesco ad individuare un motivo prevalente. Perché in effetti i motivi sono vari e, per dirla in gergo "tecnico", quasi tutti equamente probabili (come in un esperimento casuale …, ma solo perché ancora non ho deciso quale tra questi è quello che alla fine ha prevalso).

Prima di provare ad elencarli, è bene cercare di definirne l'ambito scientifico di riferimento, data la sua assoluta rilevanza in tutto ciò che caratterizza l'epoca che stiamo vivendo. La velocità con cui si trasferiscono oggi le informazioni sul web richiede la predisposizione di strumenti di analisi sempre più adeguati a trattare moli di dati, di algoritmi sempre più veloci che permettano ai cosiddetti *decision maker* di operare scelte basate su informazioni che il trascorrere del tempo (non più degli anni, ma forse dei giorni se non addirittura delle ore) può rendere obsolete molto velocemente. Con l'avvento di questo secolo si è, non a caso, assistito al susseguirsi della diffusione di nuove terminologie e tecniche di gestione e analisi delle informazioni (spesso strettamente connesse tra loro) quali: *Knowledge Discovery in Databases*, *Data-warehousing Solutions*, *Machine Learning*, *Data Mining*, *Big Data analytics* ecc.

In questo vasto panorama, appare evidente come gli statistici assumano un ruolo rilevante se non strategico. La trasposizione dei metodi e modelli statistici classici alle opportunità offerte dai moderni calcolatori ha ampliato il ventaglio delle prospettive di analisi e delle possibili applicazioni. Molte di queste metodologie sono superficialmente identificate come appartenenti al generico ambito della Statistica Computazionale, ovvero all'interfaccia

B. Bertaccini, *Introduzione alla Statistica Computazionale con R*, ISBN (print) 978-88-6453-673-6, ISBN (online) 978-88-6453-674-3, CC BY 4.0, 2018 Firenze University Press.

tra la Statistica e l'Informatica. Superficialmente appunto, perché nella letteratura anglosassone si ritiene opportuno compiere uno sforzo semantico che noi trascuriamo, volendo distinguere cosa è inquadrabile come *Computational Statistics* da ciò che invece riguarda lo *Statistical Computing*, ovvero l'applicazione del calcolo elettronico (anche intensivo) ai tradizionali metodi e strumenti propri della Statistica. Nell'ambito della *Computational Statistics* rientra invece tutto ciò che riguarda la progettazione e lo sviluppo di algoritmi finalizzati ad implementare metodi statistici con (e per) i computer, o a risolvere problemi complessi se non impossibili da trattarsi dal punto di vista analitico (Albert e Gentle, 2004; Lauro, 1996; Wilkinson, 2008). Mentre tutti gli statistici dovrebbero mostrare una qualche abilità nell'utilizzo dei più comuni pacchetti statistici, le competenze nell'ambito della *Computational Statistics* non sono per niente scontate.

Molti testi didattici (se in lingua italiana, spesso frutto di un'opera di traduzione) nonostante facciano genericamente riferimento all'ambito della Statistica Computazionale, in realtà si limitano ad introdurre all'utilizzo di un particolare software o pacchetto per l'applicazione dei principali metodi statistici per l'analisi dei dati (per cui dovrebbero essere catalogati come testi di *Statistical Computing*). Mancava a mio avviso un testo che introducesse alla vera arte della Statistica Computazionale, ovvero in grado di illustrare come le competenze di programmazione informatica nello sviluppo di algoritmi possano essere messe al servizio della Statistica per la simulazione e replicazione virtuale di realtà ed esperimenti più o meno complessi. Nella realtà attuale, saper simulare scenari decisionali complessi potrebbe rivelarsi di fondamentale importanza nell'assicurarsi un vantaggio nei confronti dei reali o potenziali competitors.

Ci sono centinaia di testi sul pacchetto **R**. Questo non deve stupire: negli ultimi anni il numero di utilizzatori di **R** al mondo è cresciuto in maniera esponenziale, perché stiamo parlando di un pacchetto estremamente versatile per l'analisi statistico-matematica ed il *Data Mining*, realizzato attorno ad un linguaggio di programmazione vettoriale *java-like*. Il linguaggio quindi ben si presta a svolgere tutte quelle attività di ricerca e analisi che definiscono l'ambito della statistica computazionale. Mi sembra però che siano (in proporzione si intende) relativamente pochi i testi che si occupano di introdurre all'arte della simulazione. E non con il livello di esemplificazione che questo testo intende fornire.

Non verranno pertanto volutamente presentate le funzioni proprie dell'analisi statistica multivariata e per la stima dei parametri dei modelli più comuni. Sarà invece dedicata attenzione all'implementazione di algoritmi per la realizzazione di effetti grafici avanzati, altro elemento a mio avviso carente nei testi didattici che introducono a **R**, che ha appunto nella capacità di sviluppo grafico uno dei suoi punti di forza. Il tutto non dimenticando che quest'opera deve supportare lo studio di un insegnamento da (circa) 9 Crediti Formativi Universitari nell'ambito di un corso di laurea triennale in Statistica.

Sono un perito matematico-informatico come formazione superiore ed uno statistico come formazione universitaria. Ho iniziato a prendere dimestichezza con **R** alla fine del mio percorso di laurea magistrale e solo per condurre analisi finalizzate, e certamente **R** non era quello che è oggi. Quasi tutto quello che verrà qui presentato l'ho quindi auto-appreso in molti anni di studio e lavoro, ed in corsi di aggiornamento certificati dalla John Hopkins University, che mi sono divertito a svolgere grazie all'ottima piattaforma Coursera.

Da studente un testo così mi avrebbe fatto certamente comodo, ma **R** era ancora agli albori. Ecco… forse è questo il motivo principale per cui ho deciso di prendere carta, penna e calamaio (metafora molto poetica… in realtà come immaginerete ho aperto Word).

Buono studio!

fine estate 2017, a Vaglia

Capitolo 1

Cosa è R

R è un ambiente di sviluppo specifico per l'analisi statistica dei dati che utilizza un linguaggio di programmazione derivato (e in larga parte compatibile) con S-plus. È un software libero in quanto viene distribuito con la licenza **GNU GPL**[1], ed è disponibile per diversi sistemi operativi (ad esempio Unix, GNU/Linux, Mac OS X, Microsoft Windows - http://www.r-project.org). Il suo linguaggio orientato agli oggetti deriva direttamente dal pacchetto S-plus distribuito con licenza commerciale e sviluppato da John Chambers e altri presso i Bell Laboratories.

R venne scritto inizialmente da Robert Gentleman e Ross Ihaka. Dalla metà del 1997 si è aggiunto ai due sviluppatori iniziali un gruppo di programmatori (tra cui lo stesso John Chambers) con diritti di scrittura sul progetto.

Nell'ambito dei software di tipo statistico, costituisce una valida alternativa ai package più diffusi (SAS, SPSS, ecc.)[2]. La popolarità di **R**, in rapida crescita, è dovuta soprattutto alla

[1] GNU è un acronimo ricorsivo e significa ***GNU is Not Unix*** (ovvero "GNU non è Unix").
L'acronimo ricorsivo è un particolare tipo di acronimo nel quale una delle lettere che lo compongono (solitamente la prima) è l'iniziale della parola che costituisce l'acronimo stesso. Tale tipo di acronimo è ampiamente diffuso nella comunità *hacker*.
Il Progetto GNU, lanciato nel 1983 da Richard Stallman, si basa su una gestione particolare dei diritti d'autore sul software, secondo la definizione di software libero. Fulcro di tutta l'attività del Progetto GNU è la licenza chiamata **GNU General Public License** (GNU GPL), che sancisce e protegge le libertà fondamentali che, secondo Stallman, permettono l'uso e lo sviluppo collettivo e naturale del software (Wikipedia, marzo 2010).

[2] Gli enti nazionali deputati alla produzione di statistiche ufficiali stanno riducendo l'uso dei principali pacchetti commerciali (SAS e SPSS) e sono sempre più frequenti sul mercato richieste di profili professionali con una buona conoscenza di **R** (per approfondimenti, cfr. Espa e Micciolo, 2008; Muggeo e Ferrara, 2005; Venables ed altri, 2016).

B. Bertaccini, *Introduzione alla Statistica Computazionale con R*, ISBN (print) 978-88-6453-673-6, ISBN (online) 978-88-6453-674-3, CC BY 4.0, 2018 Firenze University Press.

ampia disponibilità di moduli e librerie distribuiti con licenza GPL e organizzati in un apposito sito chiamato CRAN (Comprehensive **R** Archive Network). Tramite questi moduli è possibile estendere notevolmente le funzionalità base del programma.

Anche se il linguaggio è fornito con un'interfaccia a linea di comando, sono disponibili diverse interfacce grafiche GUI (*Graphical User Interface*) che consentono di integrare **R** con diversi pacchetti (una rassegna delle GUI disponibili può essere trovata sul blog **R-statistics** (https://www.r-statistics.com/tag/r-gui).

Fin dal 2001, nell'ambito del progetto, è stata pubblicata **R** *News*, una rivista elettronica a scadenza irregolare (da due fino 5 edizioni annue) riguardante l'utilizzo di moduli o l'applicazione di **R** per alcuni problemi di statistica applicata. Da maggio 2009 **R** News è stata sostituita dal più ampio *The **R** Journal*.

Capitolo II

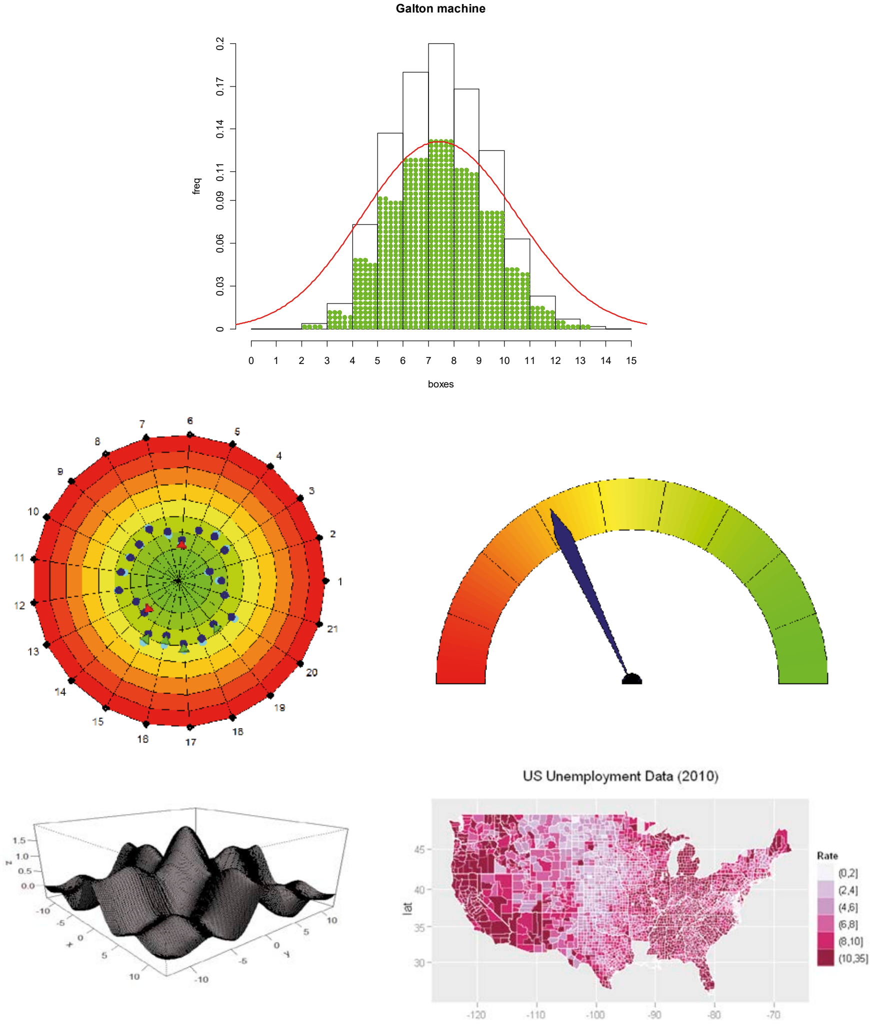
Perché usare R?

Come detto in precedenza, nell'ambito dei software di tipo statistico, **R** costituisce una libera e valida alternativa ai package commerciali più diffusi. Ma la licenza **GNU GPL** non può certamente essere l'unico elemento in grado di far propendere verso l'adozione di tale software. In realtà molte altre caratteristiche contribuiscono a rendere **R** un pacchetto estremamente interessante e raccomandabile nei più svariati ambiti scientifici d'applicazione:

- **R** mostra un funzionamento piuttosto snello che generalmente non impegna molto le risorse di un elaboratore; l'utilizzo della memoria è incrementale in base agli oggetti e ai pacchetti modulari aggiuntivi richiesti dall'utente;
- **R** è stato sviluppato secondo una logica vettoriale, per cui le strutture informative di base sono vettori e matrici. Questo concetto diverrà più chiaro nel prosieguo della trattazione. Per il momento basti pensare che in **R** non si apprezzano differenze di elaborazione tra operazioni che coinvolgono semplici scalari rispetto a quelle che coinvolgono vettori e matrici, anche di grandi dimensioni;
- **R** si dimostra una valida piattaforma per l'analisi esplorativa ed interattiva;
- **R** contiene un potente linguaggio di programmazione (*Java-like*) per lo sviluppo di strumenti e librerie personalizzate;
- **R** dispone in assoluto di uno tra i più completi e potenti set di funzionalità grafiche per l'analisi di dati complessi. Plot e grafici che in **R** sono realizzabili con poche righe di codice, in altri pacchetti e linguaggi diventano, sempre se tecnicamente implementabili, il risultato di procedure algoritmiche di elevata complessità (cfr. Fig. 1).

B. Bertaccini, *Introduzione alla Statistica Computazionale con R*, ISBN (print) 978-88-6453-673-6, ISBN (online) 978-88-6453-674-3, CC BY 4.0, 2018 Firenze University Press.

Figura 1 – Esempi di analisi grafiche realizzabili con R



Capitolo III

Installazione e layout di R

L'installazione di **R** su un qualsiasi sistema operativo è un'operazione che può essere compiuta in pochi e semplici passi. Occorre preliminarmente scaricare la versione desiderata selezionando un server "*mirror*" dalla lista pubblicata alla pagina: https://cran.r-project.org/mirrors.html. Successivamente, si viene reindirizzati ad una pagina che propone l'elenco delle distribuzioni compilate di **R** per i sistemi Linux, Mac OsX e Windows (cfr. Fig. 2).

Figura 2 – Pagina web del CRAN (Comprehensive R Archive Network)

The Comprehensive R Archive Network

Download and Install R

Precompiled binary distributions of the base system and contributed packages, **Windows and Mac** users most likely want one of these versions of R:

- Download R for Linux
- Download R for (Mac) OS X
- Download R for Windows

R is part of many Linux distributions, you should check with your Linux package management system in addition to the link above.

Source Code for all Platforms

Windows and Mac users most likely want to download the precompiled binaries listed in the upper box, not the source code. The sources have to be compiled before you can use them. If you do not know what this means, you probably do not want to do it!

- The latest release (Tuesday 2016-06-21, Bug in Your Hair) R-3.3.1.tar.gz, read what's new in the latest version.
- Sources of R alpha and beta releases (daily snapshots, created only in time periods before a planned release).
- Daily snapshots of current patched and development versions are available here. Please read about new features and bug fixes before filing corresponding feature requests or bug reports.
- Source code of older versions of R is available here.
- Contributed extension packages

Questions About R

- If you have questions about R like how to download and install the software, or what the license terms are, please read our answers to frequently asked questions before you send an email.

B. Bertaccini, *Introduzione alla Statistica Computazionale con R*, ISBN (print) 978-88-6453-673-6, ISBN (online) 978-88-6453-674-3, CC BY 4.0, 2018 Firenze University Press.

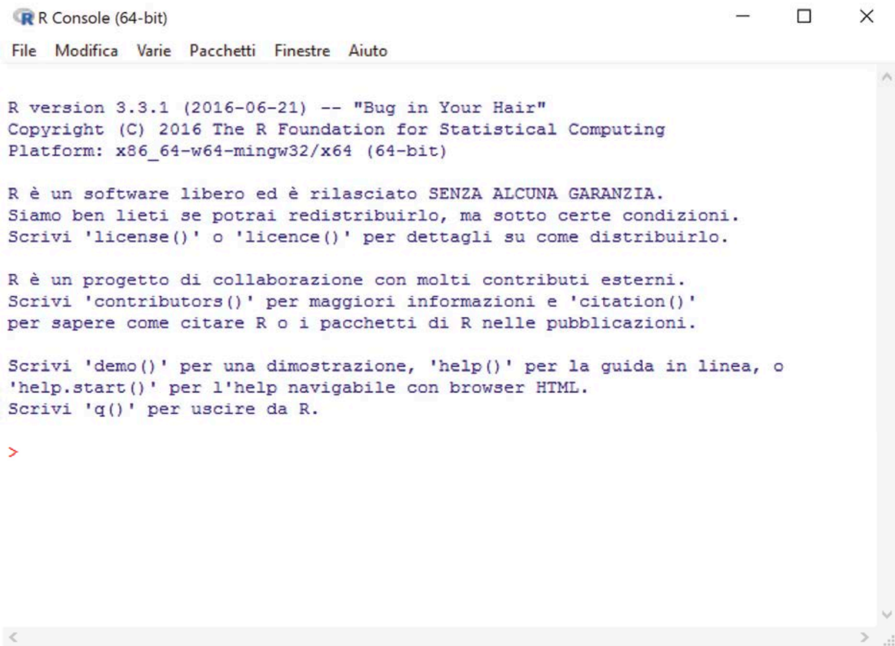
La selezione della piattaforma desiderata permette di avviare il download del relativo pacchetto di installazione.

Si suggerisce di integrare l'installazione di **R** con *tool* integrati di sviluppo o editor dedicati in grado di dialogare con la consolle di comando (quali ad esempio: R-Studio, Tinn-R o StatET che è un IDE - *Integrated Development Environment* - basato su Eclipse).

Una volta avviato, **R** si presenta mediante una interfaccia grafica (RGui) essenziale nelle funzionalità utente attivabili tramite la barra dei menu e le icone pulsante (cfr. Fig. 3). A differenza del pacchetto commerciale S-plus da cui deriva, in **R** non è possibile produrre le analisi desiderate mediante il ricorso all'interfaccia utente della distribuzione di default. Come detto in precedenza, sono però disponibili diversi pacchetti che consentono l'attivazione di interfacce grafiche GUI alternative rispetto a quella in distribuzione.

A meno che non venga installata anche una GUI esterna, in **R** tutte le istruzioni sono eseguibili dalla linea di comando dell'ambiente, tipicamente ma non necessariamente, identificata mediante il carattere prompt "**>**". Ovviamente, ogni linea di comando viene eseguita semplicemente premendo il tasto "Invio" della tastiera.

Figura 3 – Interfaccia utente di R in Windows



Capitolo IV

Pacchetti aggiuntivi

Come vedremo meglio nel prosieguo, il ventaglio delle funzionalità base di **R** è molto esteso. Ci sono però molti strumentazioni interessanti che non compongono il pacchetto base e che sono disponibili sotto forma di librerie aggiuntive. Questi pacchetti, liberamente scaricabili dal sito del CRAN (alla pagina web: https://cran.r-project.org/web/packages), raggiungono oggi (alla data di stampa di questo testo) le 9.000 unità consentendo a **R** di adattarsi alle necessità analitiche dei più svariati ambiti scientifici.

Tecnicamente, un pacchetto è una collezione strutturata di funzioni, dataset e codice compilato. La directory all'interno del disco fisso, deputata al mantenimento di tutti i pacchetti base e aggiuntivi è detta libreria. La funzione `.libPaths()` informa l'utente sulla localizzazione della libreria. La funzione `library()` invece elenca tutti i pacchetti salvati all'interno della libreria.

Per installare un pacchetto è possibile usare la funzione `install.packages()`. Una volta selezionato il server "mirror", si attiva la lista dei pacchetti aggiuntivi disponibili per la selezione ed il download. La stessa funzione consente anche di installare direttamente un pacchetto conoscendone il nome:

```
> install.packages("ggplot2")

--- Please select a CRAN mirror for use in this session ---
provo con l'URL
'https://cran.wu.ac.at/bin/windows/contrib/3.3/ggplot2_2.1.0.zip'
Content type 'application/zip' length 2002629 bytes (1.9 MB)
downloaded 1.9 MB
```

B. Bertaccini, *Introduzione alla Statistica Computazionale con R*, ISBN (print) 978-88-6453-673-6, ISBN (online) 978-88-6453-674-3, CC BY 4.0, 2018 Firenze University Press.

```
package 'ggplot2' successfully unpacked and MD5 sums checked

The downloaded binary packages are in
C:\Users\Bruno\AppData\Local\Temp\RtmpwBbJ8R\downloaded_packages
```

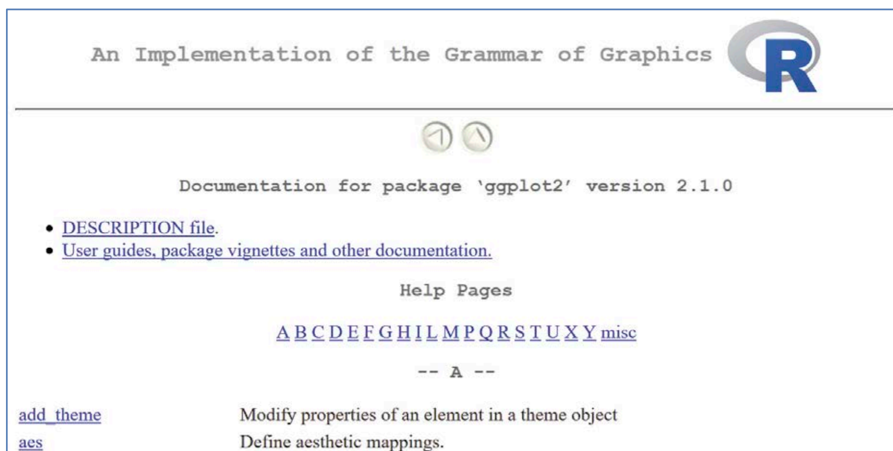
Per iniziare ad usare il pacchetto, occorre che questo venga caricato in memoria. Ciò avviene tramite la funzione `library()`. Ad esempio, il comando sottostante estende le funzionalità base di **R** con quelle disponibili all'interno del pacchetto aggiuntivo "ggplot2":

```
> library(ggplot2)
```

Infine, la funzione `help()` costituisce un valido punto di riferimento per esaminare il contenuto delle librerie, accedere alle note informative delle funzioni e ai file d'esempio in esse contenute (cfr. Fig. 4):

```
> help(package=ggplot2)
```

Figura 4 – Finestra attivata dalla funzione `help(package=ggplot2)` in Windows



Capitolo V

Una calcolatrice estremamente potente

Nel suo utilizzo più semplice, **R** può essere utilizzato come una calcolatrice scientifica estremamente potente. Dal prompt dei comandi ci si rende subito conto di come **R** esegua le operazioni richieste rispettando le regole aritmetiche di precedenza degli operatori:

```
> 6+7
[1] 13

> 4/3
[1] 1.333333

> 7-3*2
[1] 1

> 7-2^3
[1] -1
```

Se si desidera che le operazioni vengano eseguite in una successione differente allora è necessario ricorrere all'uso delle parentesi tonde[3].

```
> (7-2)^3
[1] 125
```

Sulla stessa riga possono essere anche eseguite due operazioni distinte ricorrendo al carattere separatore ";":

[3] Le parentesi tonde sono l'unico tipo di parentesi utilizzabile nelle espressioni aritmetiche; le parentesi quadre e graffe sono deputate ad altri usi.

B. Bertaccini, *Introduzione alla Statistica Computazionale con R*, ISBN (print) 978-88-6453-673-6, ISBN (online) 978-88-6453-674-3, CC BY 4.0, 2018 Firenze University Press.

```
> 4*5-2^3; 4/5 # questa è la seconda operazione
[1] 12
[1] 0.8
```

Il carattere ' # ' è utilizzato per indicare l'inizio di un commento; tutti i caratteri che seguono vengono da **R** ignorati.

La Tabella 1 illustra le principali *funzioni* matematiche presenti nella strumentazione matematica di base di **R**.

Tabella 1 – *Principali funzioni matematiche di R base*.

`x%%y`	Resto della divisione	`> 5%%3` `[1] 2`
`x%/%y`	Divisione intera	`> 5%/%3` `[1] 1`
`abs(x)`	Valore assoluto di un'espressione	`> abs(3-4^2)` `[1] 13`
`sign(x)`	Segno di un'espressione	`> sign(3-4^2)` `[1] -1`
`sqrt(x)`	Radice quadrata	`> sqrt(3^2+4^2)` `[1] 5`
`log(x)`	Logaritmo naturale	`> log(10)` `[1] 2.302585`
`log(x, base=a)`	Logaritmo in base *a*	`> log(10,base=10)` `[1] 1`
`exp(x)`	Esponenziale	`> exp(1)` `[1] 2.718282`

`sin(x)`
`cos(x)`
`tan(x)`
`asin(x)`
`acos(x)`
`atan(x)`
	Funzioni trigonometriche e loro inverse[4]	`> sin(pi/2)`
		`[1] 1`
		`> cos(pi/2)`
		`[1] 6.123032e-17`
		`> tan(pi/2)`
		`[1] 1.633178e+16`

`factorial(n)`	Fattoriale	`> factorial(10)` `[1] 3628800`
`choose(n,k)`	Coefficiente combinatorio	`> choose(10,5)` `[1] 252`

Spesso può essere utile eseguire operazioni "a cascata" utilizzando calcoli effettuati in precedenza. In questo è necessario memorizzare i risultati ottenuti per poterli utilizzarli successivamente.

In **R** per salvare "qualcosa" occorre costruire un **oggetto**. Con tale termine in **R** sono identificati numeri, espressioni, formule, funzioni, fogli dati, ecc. Ovviamente, ogni oggetto può essere più o meno complesso a seconda delle sue peculiarità o proprietà. Per costruire un

[4] **R** di default memorizza nella variabile `pi` il valore del *pi-greco*. Tale variabile è però riutilizzabile dall'utente per scopi diversi.

oggetto si possono utilizzare indistintamente i caratteri: “**<-**”, “**=**” o “**->**”. Per visualizzarlo è sufficiente richiamarlo dal prompt dei comandi.

```
> a=4
> b<-4
> a-b
[1] 0

> 3->c
> a%%c
[1] 1
```

Si osservi che in questo semplice esempio `a`, `b` e `c` sono oggetti semplici assimilabili alle variabili di tipo reale dei più comuni linguaggi di programmazione. Nel prosieguo della presentazione saranno presentati oggetti più complessi.

Agli oggetti di **R** possono essere assegnati nomi qualsiasi; è però bene tenere presente che, nel momento in cui si decide di creare un oggetto, eventuali oggetti pre-esistenti che possiedono lo stesso nome vengono sovrascritti e cancellati. **R** inoltre è un ambiente *case-sensitive* (ovvero fa distinzione tra caratteri maiuscoli e minuscoli), per cui gli oggetti dell’esempio seguente sono riconosciuti come distinti:

```
> a.1 = 4
> A.1 = 5

> a.1 + A.1
[1] 9
```

Come illustrato, i nomi degli oggetti possono contenere anche il carattere “`.`” (che in altri ambienti di programmazione può assumere un significato speciale). L’unica avvertenza nella denominazione degli oggetti è quindi quella di prestare attenzione all’impiego di nomi già utilizzati dal sistema, ovvero alle così dette *parole chiave* che sono “parole riservate” che non possono essere utilizzate dall'utente a proprio piacimento, ma solo per le finalità per le quali sono state costruite[5]; queste sono:

```
FALSE, TRUE, Inf, NA, NaN, NULL,

break, else, for, function, if, in, next, repeat, while
```

Oltre a queste parole chiave è sconsigliabile anche l’impiego dei nomi delle funzioni di sistema sebbene questo possa rivelarsi più un problema per l’utente (eventuali difficoltà nella corretta identificazione degli oggetti) che un problema per **R**:

```
> cos
function (x)  .Primitive("cos")
```

[5] Più avanti saranno illustrati i contesti di impiego di queste parole.

```
> cos=2

> cos
[1] 2

> cos(0)
[1] 1
```

Solo con la pratica si impareranno a conoscere i nomi che dovrebbero essere evitati, ma fino a quel momento per maggiore sicurezza si potrebbe interrogare **R** prima della denominazione di un oggetto per comprendere se quel nome è già utilizzato oppure no, come fatto nell'esempio precedente quando si è inizialmente interrogato l'oggetto "*cos*" ed il sistema ha risposto che `cos()` è una funzione primitiva. Ad esempio:

```
> B
Error: object 'B' not found
```

Tramite linea di commando è possibile anche effettuare verifiche sullo stato logico delle proposizioni. Il confronto fra due oggetti è reso possibile tramite l'utilizzo degli operatori "**==**" (operatore "*uguale* a" da distinguere dall'operatore "=" da utilizzare in caso di assegnazione), "**>=**", "**<=**", "**>**","**<**" e "**!=**" (operatore "*diverso da*"). Ciascuno di questi operatori restituisce come risultato la costante `TRUE` o un `FALSE` a seconda del risultato logico dell'espressione:

```
> a=3
> a>0
[1] TRUE

> a==b
[1] FALSE

> a-b>0
[1] FALSE

> a!=b
[1] TRUE
```

Le costanti logiche `TRUE` e `FALSE` corrispondono rispettivamente ai valori 1 e 0 possono convenientemente essere utilizzate in molte operazioni numeriche (e, come vedremo più avanti, in operazioni di indicizzazione)[6]:

```
> TRUE+TRUE
[1] 2

> TRUE*5
[1] 5
```

[6] **R**, come per la variabile `pi`, all'inizio di ogni sessione di lavoro assegna alle variabili `T` e `F` rispettivamente il valore delle costanti logiche `TRUE` e `FALSE`, ma l'utente ha la possibilità di cambiare tale assegnazione e utilizzare le variabili `T` e `F` a proprio piacimento.

```
> TRUE-FALSE
[1] 1

> TRUE*FALSE
[1] 0

> FALSE/TRUE
[1] 0

> TRUE/FALSE
[1] Inf

> FALSE/FALSE
[1] NaN
```

Si osservi come le forme indeterminate sono correttamente prese in considerazione da **R** che riesce a restituire i risultati “corretti”, Inf (Infinito) e NaN (Not-a-Number).

Capitolo VI

Personalizzazione delle funzioni

Come visto, in **R** è possibile specificare un insieme di comandi in grado di risolvere un determinato problema matematico. Tali comandi possono essere spesso (ma come vedremo non sempre) specificati “a cascata” dal prompt dei comandi oppure possono, molto più agevolmente, essere organizzati “in modo compatto” in modo da poter essere eseguiti in un’unica istruzione.

Un modo semplice che consente ciò è organizzare una o più di linee di codice all’interno di una **funzione**. La funzione è anch’essa un oggetto di **R** e può essere generalizzata, ovvero costruita in maniera tale che il risultato restituito sia appunto funzione di uno o più parametri in ingresso.

Per eseguire una generica funzione chiamata “***funz***” (a patto che risulti definita nell’ambiente di lavoro) ed ottenere il relativo risultato è sufficiente digitare il suo nome specificando fra parentesi tonde gli eventuali argomenti a cui deve essere applicata.

R è ovviamente dotato di una grande quantità di funzioni che assolvono i compiti più “comuni” in ambito matematico e statistico. La Tabella 1 illustrata in precedenza, riporta esempi di funzioni matematiche più o meno complesse. Le funzioni statistiche (dalle più elementari alle più avanzate) verranno presentate nel prosieguo della trattazione[7].

Possiamo però pensare di costruire funzioni personalizzate in grado di risolvere problemi specifici ed aggiungerle nell’ambiente di lavoro. Supponiamo ad esempio di avere la necessità di conoscere il valore in ordinata della parabola $y = 3x^2 + 5x + 2$.

[7] Per ogni funzione “*funz*” di sistema, **R** è dotato di un file guida, visionabile digitando `?funz` o `help(funz)` dal prompt dei comandi.

B. Bertaccini, *Introduzione alla Statistica Computazionale con R*, ISBN (print) 978-88-6453-673-6, ISBN (online) 978-88-6453-674-3, CC BY 4.0, 2018 Firenze University Press.

Utilizzando lo *script editor* di sistema (attivabile dal menù *File*) scriviamo:

```
mia.parabola1 <- function(x){

y=3*x^2+5*x+2
return(y)
}
```

Copiamo quanto scritto (si può agevolmente utilizzare la funzionalità “seleziona tutto” dell’editor) e lo incolliamo nel prompt di **R**, prestando attenzione alla eventuale necessità di premere il tasto *Invio* al termine dell’operazione se il prompt dei comandi segnala un “+” invece che l’usuale carattere “>”. Se abbiamo commesso errori sintattici nella definizione della funzione o nelle operazioni di trasferimento (*copia - incolla*) dall’editor, **R** segnalerà l’errore commesso; altrimenti saremo sicuri dell’avvenuta definizione della nuova funzione all’interno dell’ambiente di lavoro.

Si noti l’utilità della funzione `return()`, che consente di restituire una determinata espressione / oggetto all’area di lavoro dalla quale la funzione `mia.parabola1()` è stata chiamata (nel caso precedente la y). L’impiego della `return()` può essere anche omesso: se viene raggiunta la fine di una funzione senza tale chiamata, viene restituito di default il valore dell’ultima espressione / oggetto elaborato.

L’oggetto funzione di sistema `ls()`, non parametrizzato, restituisce un vettore di stringhe identificanti i nomi degli oggetti presenti nell’ambiente di lavoro. Tramite questo oggetto è possibile appurare la presenza della nostra nuova funzione nell’ambiente di lavoro.

```
> ls()
[1] "mia.parabola1"
```

Adesso la nostra funzione è richiamabile dal prompt dei comandi, indicando tra parentesi il valore del dominio in cui deve essere calcolata, e ricordando comunque che **R** è un ambiente a logica vettoriale. In particolare la seconda chiamata richiede a **R** di calcolare i valori della funzione nel dominio a valori interi da 1 a 4 (le sequenze a valori interni verranno riprese nel capitolo 8 dedicato ai “Vettori”):

```
> mia.parabola1(0)
[1] 2

> mia.parabola1(1:4)
[1] 10 24 44 70

> mia.parabola1(50)
[1] 7752

> mia.parabola1(Inf)
[1] Inf
```

Ovviamente, ogni nuovo oggetto può anche essere rimosso dall'ambiente di lavoro. Un modo estremamente semplice per farlo è utilizzare la funzione `rm()`:

```
> rm(mia.parabola1)
```

La funzione `mia.parabola1()` definita in precedenza era generalizzata a tutti i possibili valori assumibili dalla funzione all'intero del suo dominio, ma non è stata generalizzata alla classe di tutte i problemi simili, ovvero all'individuazione dei valori assunti in ordinata della generica parabola $y = ax^2 + bx + c$. Utilizzando lo script editor, possiamo modificare la funzione precedente nel seguente modo:

```
mia.parabola2 <- function(x,a,b,c){

y=a*x^2+b*x+c
return(y)
}
```

Adesso i parametri in ingresso della funzione sono diventati 4: il valore del dominio in cui deve essere calcolata la parabola ed i tre usuali parametri di identificazione della parabola nella classe di funzioni definita dalla formula $y = ax^2 + bx + c$. Dopo aver incollato nell'ambiente di lavoro il contenuto dell'editor, richiamiamo la nostra funzione prestando attenzione all'ordine in cui sono stati definiti i parametri (a meno che non si effettui una dichiarazione esplicita dei valori da assegnare ai parametri, come nei primi due casi sotto riportati):

```
> mia.parabola2(1,a=2,b=3,c=1)
[1] 6

> mia.parabola2(0,b=5,c=-1,a=-2)
[1] -1

> mia.parabola2(0,5,-1,-2)
[1] -2

> mia.parabola2(1:5,0,2,2)
[1]  4  6  8 10 12
```

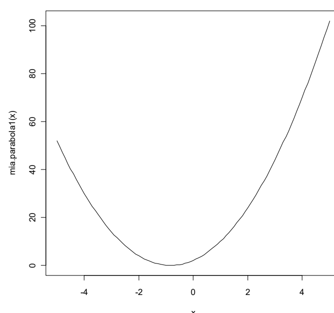
Capitolo VII

Strumenti grafici per lo studio di funzioni

R, oltre ad essere una calcolatrice estremamente potente, dispone anche di importanti funzionalità grafiche che lo rendono forse il software statistico più completo e versatile per la visualizzazione di dati ed informazioni[8].

Inizialmente ci limiteremo ad illustrare alcune semplici funzionalità per lo studio delle funzioni matematiche. La funzione `curve()` consente di disegnare funzioni matematiche, a patto che queste risultino definite nell'intervallo specificato dagli argomenti `from=` e `to=` (che ne definiscono gli estremi). Partiamo dal grafico della parabola descritta dalla funzione `mia.parabola1()`:

```
> curve(mia.parabola1(x), from=-5, to=5)
```

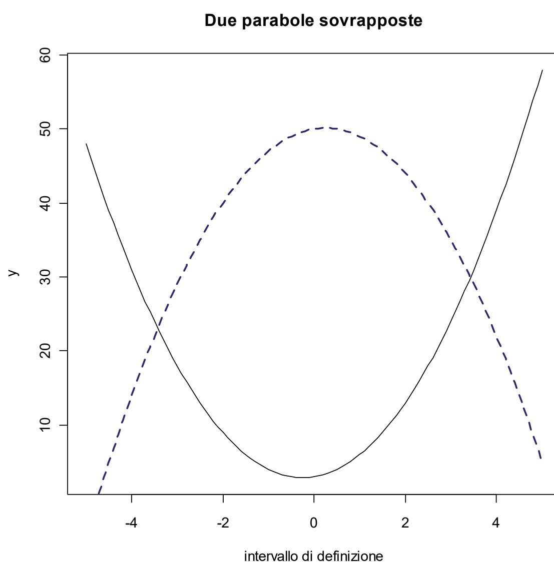


[8] Per un'idea delle potenzialità grafiche di **R** si consulti il sito ufficiale del progetto alla pagina web: http://www.r-project.org/screenshots/screenshots.html.

B. Bertaccini, *Introduzione alla Statistica Computazionale con R*, ISBN (print) 978-88-6453-673-6, ISBN (online) 978-88-6453-674-3, CC BY 4.0, 2018 Firenze University Press.

Si osservi che all'interno di una finestra grafica è possibile "sovrapporre" più curve; tale operazione è consentita ponendo a `TRUE` l'argomento di tipo logico `add=` (che per default è settato al valore `FALSE`, ed in tal caso ogni nuova chiamata della funzione curve attiverà una nuova finestra grafica). Per sovrapporre grafici di parabole differenti ricorriamo alla funzione `mia.parabola2()`: gli argomenti `lwd=` e `lty=` specificano rispettivamente la dimensione e il tipo di tratteggio, l'argomento `col= ne` specifica il colore. Per assegnare titoli al grafico utilizziamo la funzione `title()` agendo sui suoi argomenti `main=` , `xlab=` e `ylab=` .

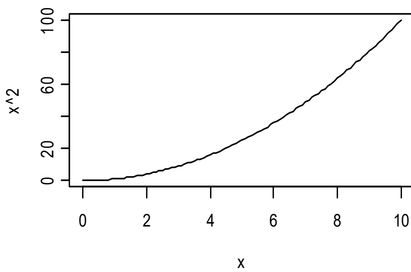
```
> curve(mia.parabola2(x,a=2,b=1,c=3), from=-5, to=5, ylab="", xlab="")
> curve(mia.parabola2(x,a=-2,b=1,c=50), from=-5, to=5,add=T, lty=2,
  lwd=2, col="navy")
> title(main="Due parabole sovrapposte", ylab="y", xlab="intervallo di
  definizione")
```

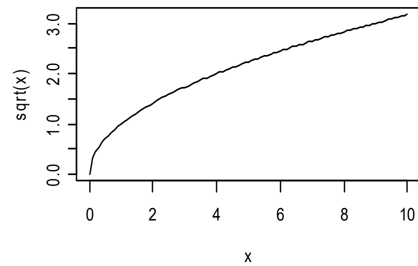


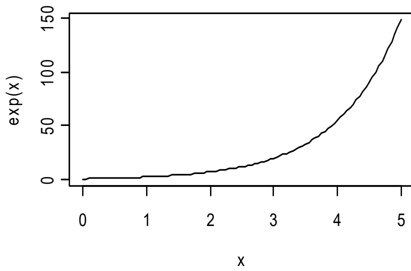
Oltre a sovrapporre più curve all'interno dello stesso grafico è possibile produrre più grafici all'interno della stessa area grafica; tramite la funzione `par()` è possibile definire la dimensione della matrice di grafici desiderata (specificandone righe e colonne) e se i grafici devono essere disegnati per riga o per colonna:

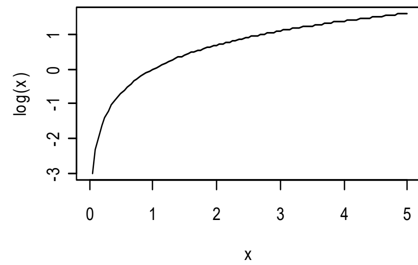
```
> par(mfrow=c(3,2)) # matrice di grafici 3x2 riprodotti per riga
> curve(x^2,0,10)
> title(main="Parabola")
> curve(sqrt(x),0,10)
> title(main="Radice quadrata")
```

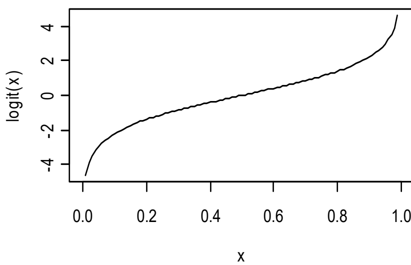
```
> curve(exp(x),0,5)
> title(main="Esponenziale")

> curve(log(x),0,5)
> title(main="Logaritmo naturale")

> curve(log(x/(1-x)),0,1,ylab="")
> title(main="Logit", ylab="logit(x)")

> curve(exp(x)/(exp(x)+1),-4,4,ylab="")
> title(main="Inverse logit", ylab="prob")
```

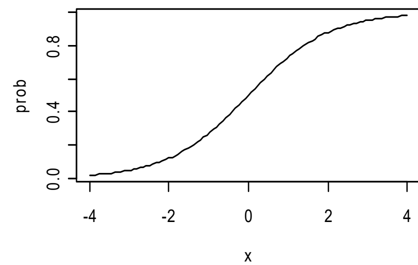












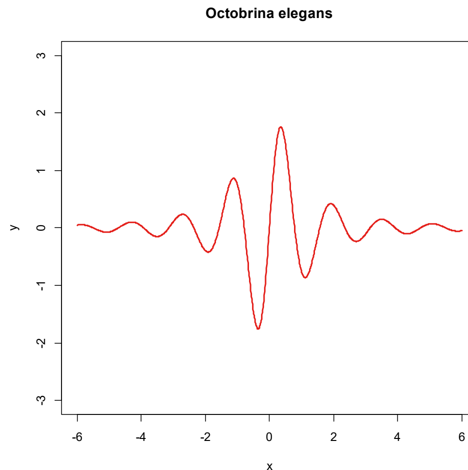
La funzione `curve()` è il risultato di un processo di raffinamento della funzione generalizzata `plot()`, tramite la quale, come vedremo, è possibile produrre grafici complessi che vanno ben oltre le problematiche connesse allo studio grafico delle funzioni matematiche. Analizziamone il funzionamento in relazione all'analisi delle cosiddette *curve parametrizzate* in $\mathbb{R}^2$, ovvero applicazioni:

$$\varphi : A \to \mathbb{R}^2 \quad \text{con } A \subset \mathbb{R}$$

per cui se $t \in A$, $\varphi(t) = \begin{cases} x(t) \\ y(t) \end{cases}$.

Ad esempio, la parametrizzazione $\varphi(t) = \begin{cases} x = t \\ y = \dfrac{2\sin(4t)}{1+t^2} \end{cases}$, con $t \in [-6, 6]$, produce la cosiddetta *Octobrina elegans*:

```
OctobrinaE=function(){
t=seq(-6,6,.01)
x=t
y=2*sin(4*x)/(1+x^2)
plot(x, y, xlim=c(-6,6), ylim=c(-3,3), type="l", col="red", lwd=2)
title(main="Octobrina elegans")
}

> OctobrinaE()
```

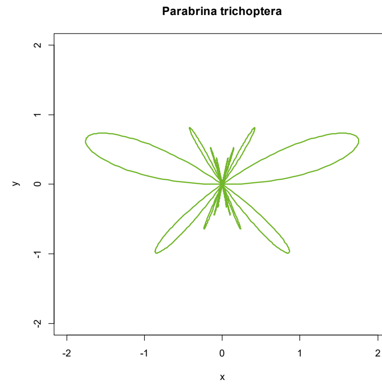


I parametri `xlim=` e `ylim=` definiscono i limiti (ovvero la scala) della finestra grafica. Il parametro `type=` consente di specificare la modalità con cui deve essere realizzato il grafico; nell'esempio, il valore "`l`" (iniziale di *lines*) indica a **R** di unire tutti i punti che compongono il vettore `(x, y)` tramite segmenti lineari.

Parametrizzazioni più sofisticate producono grafici molto più "interessanti". Ad esempio, se $f(t)=\dfrac{2\sin(4t)}{1+t^2}$ allora:

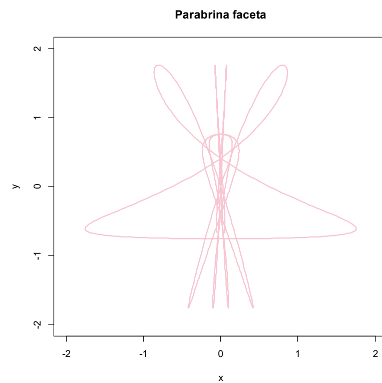
Parabrina trichoptera

$$\varphi(t)=\begin{cases} x=f(t) \\ y=t\cdot f(t) \end{cases}$$



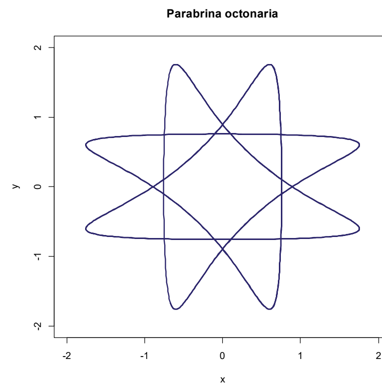
Parabrina faceta

$$\varphi(t)=\begin{cases} x=f(t) \\ y=f\left(\cos(t)\right) \end{cases}$$



Parabrina octonaria

$$\varphi(t)=\begin{cases} x=f\left(\cos(t)\right) \\ y=f\left(\sin(t)\right) \end{cases}$$



Il codice di realizzazione dei grafici riprodotti in questa pagina è lasciato al lettore come esercizio.

Capitolo VIII

Vettori

Finora, tutte le funzioni illustrate sono state quasi sempre chiamate in causa per condurre operazioni con scalari, ovvero con singoli numeri appartenenti allo spazio $\mathbb{R}$ dei numeri reali. Per **R**, in realtà tali numeri sono solo un caso particolare di una famiglia molto più grande di oggetti per la gestione dei quali è stato appositamente programmato: i vettori. Si osservi che uno scalare può essere considerato un vettore di lunghezza 1.

Come noto, un vettore di dimensione *n* può essere definito come una sequenza ordinata di *n* numeri; ad esempio, $\left(2;3.1;\sqrt{2};5;1.1\right)\in\mathbb{R}^5$. In **R**, tale vettore può essere agevolmente definito utilizzando la funzione `c()`:

```
> v=c(2,3.1,sqrt(2),5,1.1)
> v
[1] 2.000000 3.100000 1.414214 5.000000 1.100000
```

Si osservi che in **R** ogni vettore è un vettore colonna anche se la sua visualizzazione è in sequenza orizzontale.

Per richiamarne le singole componenti, ogni vettore è indicizzabile mediante il valore indice della posizione richiesta racchiusa tra parentesi quadre:

```
> v[3]
[1] 1.414214
```

Tra parentesi quadre possono essere specificati anche vettori di indici: ad esempio:

```
> v[c(1,2,5)]
[1] 2.0 3.1 1.1
```

B. Bertaccini, *Introduzione alla Statistica Computazionale con R*, ISBN (print) 978-88-6453-673-6, ISBN (online) 978-88-6453-674-3, CC BY 4.0, 2018 Firenze University Press.

oppure, vettori indici particolari costituiti dai valori `TRUE` e `FALSE`:

```
> v[c(T,T,F,F,T)]
[1] 2.0 3.1 1.1
```

Si osservi che gli indici possono essere utilizzati anche in esclusione; il segno "–" che precede il vettore di indici segnala a **R** quali sono gli elementi da escludere nella procedura di estrazione. Ad esempio, la seguente scrittura richiede a **R** di estrarre tutte le componenti del vettore esclusa la 5:

```
> v[-5]
[1] 2.000000 3.100000 1.414214 5.000000
```

Oltre alla funzione `c()`, ci sono altri metodi per generare un vettore.

Ad esempio, la sintassi `a:b` genera un vettore le cui componenti sono tutti gli incrementi unitari tra gli estremi `a` e `b` compresi (a patto ovviamente che b sia un multiplo unitario di `a`):

```
> 1:5
[1] 1 2 3 4 5

> 2.2:4.3
[1] 2.2 3.2 4.2
```

La funzione `seq(from, to)` genera una sequenza di valori tra un valore minimo ed un valore massimo, specificati quali parametri in ingresso (con i parametri `by` e `length` esclusivi ed opzionali, tramite i quali è possibile specificare rispettivamente il passo della sequenza oppure la sua lunghezza):

```
> seq(from=3,to=15)
 [1]  3  4  5  6  7  8  9 10 11 12 13 14 15

> seq(3,15,by=2)
[1]  3  5  7  9 11 13 15

> seq(3,15,length=5)
[1]  3  6  9 12 15
```

La funzione `rep(x,times)` serve per generare un vettore della lunghezza specificata tramite il parametro `times`, i cui componenti sono tutte uguali al valore `x`:

```
> rep(4,5)
[1] 4 4 4 4 4
```

Come abbiamo visto, **R** considera uno scalare come un particolare tipo di vettore; per questo, è immediato verificare come tutte le operazioni aritmetiche e logiche illustrate in precedenza possano essere estese direttamente ai vettori:

```
> v+3
[1] 5.000000 6.100000 4.414214 8.000000 4.100000
> v/2
[1] 1.0000000 1.5500000 0.7071068 2.5000000 0.5500000
> v*1.2
[1] 2.400000 3.720000 1.697056 6.000000 1.320000
> v-2
[1]  0.0000000  1.1000000 -0.5857864  3.0000000 -0.9000000
> v^2
[1]  4.00  9.61  2.00 25.00  1.21
> v>3
[1] FALSE  TRUE FALSE  TRUE FALSE
```

Si osservi come l'ultima operazione possa rivelarsi estremamente utile per eseguire operazioni di indicizzazione:

```
> v[v>3]
[1] 3.1 5.0
```

La precedente istruzione produce come risultato un vettore di lunghezza 2 (i due elementi di v che sono maggiori di 3). Ovviamente **R** consente indicizzazioni successive sullo stesso oggetto; ad esempio possiamo chiedere a **R** di restituire solo il secondo elemento del vettore generato in precedenza:

```
> v[v>3][2]
[1] 5
```

Essendo **R** vettoriale, tutte le funzioni presentate in Tabella 1 sono applicabili ai vettori. Il risultato dell'operazione sarà un vettore le cui componenti sono i risultati dell'applicazione della funzione a ciascuna componente del vettore originario:

```
> sqrt(v)
[1] 1.414214 1.760682 1.189207 2.236068 1.048809
> sin(v)
[1]  0.90929743  0.04158066  0.98776595 -0.95892427  0.89120736
```

Stessa cosa vale per le funzioni definite dall'utente. Ad esempio, riconsideriamo la funzione `mia.parabola2()` costruita nel capitolo 6:

```
mia.parabola2 <- function(x,a,b,c){

y=a*x^2+b*x+c
return(y)
}
```

Possiamo pensare di farci restituire tutti i valori assunti dalla funzione in un certo insieme di punti del suo dominio:

```
> x=seq(-5,5,by=.5)
> x
[1] -5.0 -4.5 -4.0 -3.5 -3.0 -2.5 -2.0 -1.5 -1.0 -0.5  0.0  0.5  1.0
1.5  2.0  2.5  3.0  3.5  4.0  4.5  5.0
> mia.parabola2(x,1,2,3)
[1] 18.00 14.25 11.00  8.25  6.00  4.25  3.00  2.25  2.00  2.25  3.00
4.25  6.00  8.25 11.00 14.25 18.00 22.25 27.00 32.25 38.00
```

Questa è una importante caratteristica di **R**: le operazioni con oggetti "multipli" (quali vettori e, come vedremo in seguito, matrici) sono eseguite su ogni elemento dell'oggetto; naturalmente si deve trattare di una operazione ammissibile.

Sono ovviamente ammissibili le operazioni tra vettori della stessa dimensione. Ad esempio, creando il vettore `w` possiamo eseguire:

```
> w=1:5
> w
[1] 1 2 3 4 5
> v+w
[1] 3.000000 5.100000 4.414214 9.000000 6.100000
> v*w
[1]  2.000000  6.200000  4.242641 20.000000  5.500000
> v/w
[1] 2.0000000 1.5500000 0.4714045 1.2500000 0.2200000
> v-w
[1]  1.000000  1.100000 -1.585786  1.000000 -3.900000
> v^w
[1]   2.000000   9.610000   2.828427 625.000000   1.610510
```

Ognuna delle operazioni precedenti è stata eseguita sugli elementi con stessa posizione nei due vettori. **R** considera ammissibile anche la seguente operazione, perché la lunghezza del vettore `k` è multipla della lunghezza del vettore `v`:

```
> k=1:10
> k
 [1]  1  2  3  4  5  6  7  8  9 10
> v
[1] 2.000000 3.100000 1.414214 5.000000 1.100000
> k+v
[1]  3.000000  5.100000  4.414214  9.000000  6.100000
     8.000000 10.100000  9.414214 14.000000 11.100000
```

L'operazione successiva invece non è considerata ammissibile da **R** perché la lunghezza di `z` non è multipla della lunghezza di `v`; in questo caso **R** effettua comunque un calcolo[9], avvertendo con un messaggio (*warning*) che il risultato prodotto potrebbe non essere quello desiderato:

```
> z=v[v>2]
> z
[1] 3.1 5.0

> v+z
[1]  5.100000  8.100000  4.514214 10.000000  4.200000

Warning message:
In v + z : longer object length is not a multiple of shorter object
length
```

R dispone di alcune comode funzioni per la gestione dei vettori; tra queste, la funzione `length()` informa l'utente sulla lunghezza del vettore, la funzione `sort()` ne ordina gli elementi, la funzione `order()` restituisce gli indici di permutazione necessari per ordinare il vettore e la funzione `which()` informa sugli indici corrispondenti agli elementi `TRUE` di un vettore logico:

```
> v
[1] 2.000000 3.100000 1.414214 5.000000 1.100000

> length(v)
[1] 5

> sort(v)
[1] 1.100000 1.414214 2.000000 3.100000 5.000000

> order(v)
[1] 5 3 1 2 4

> which(v>3)
[1] 2 4
```

Si osservi che la funzione `order()` può essere agevolmente utilizzata per ordinare un vettore, in alternativa alla funzione `sort()`:

```
> v[order(v)]
[1] 1.100000 1.414214 2.000000 3.100000 5.000000
```

e che la funzione `which()` è una scorciatoia per farsi restituire quanto ottenibile con questa semplice riga di codice:

```
> (1:length(v))[v>3]
[1] 2 4
```

[9] Il riciclo degli argomenti (spesso vettori) è una operazione abbastanza frequente effettuata da **R** per cercare di eseguire comandi inizialmente non compatibili.

In altre parole, l'utente di **R** potrebbe non essere a conoscenza di tutte le possibili funzioni di cui l'ambiente dispone; ciò nonostante molte funzioni sono semplicemente sostituibili tramite poche semplici righe di codice da implementarsi grazie alla strumentazione di base di **R**.

Fin qui sono stati considerati vettori numerici, ovvero vettori su cui è possibile effettuare operazioni aritmetiche. In realtà **R** consente anche la costruzione di vettori di stringhe[10], sui cui è possibile effettuare solo un numero limitato di operazioni:

```
> s=c("Ciao","Paolino","Paperino")
> s
[1] "Ciao"     "Paolino"  "Paperino"

> length(s)
[1] 3

> s[-2]
[1] "Ciao"     "Paperino"
```

Alle componenti di un vettore possono essere anche assegnati dei nomi, ovvero delle etichette: tale assegnazione è possibile sia al momento della sua costruzione (inserendo le etichette quando si utilizza la funzione `c()`), sia attraverso la funzione `names()`. Si osservi che quest'ultima funzione può essere utilizzata sia per restituire i nomi assegnati in precedenza, sia per assegnarne dei nuovi:

```
> vett=c(v1=3,v2=pi)

> vett
      v1       v2
3.000000 3.141593

> names(vett)
[1] "v1" "v2"

> names(vett)=c("Eta","Beta")
> vett
     Eta     Beta
3.000000 3.141593
```

Concludiamo questo paragrafo con alcune considerazioni sulla capacità computazionale di **R**. Come detto, **R** è stato concepito per lavorare a livello vettoriale e non scalare, per cui <u>le operazioni sui vettori sono eseguite alla stessa velocità con sui sono eseguite quelle con gli scalari</u> (sempre ovviamente che lo spazio di memoria riservato all'ambiente di lavoro non sia prossimo alla saturazione). La funzione di sistema `Sys.time()`, che restituisce il tempo

[10] Sebbene sia possibile formare vettori che comprendano sia caratteri sia numeri, i numeri inseriti in vettori \`misti' vengono comunque considerati come caratteri.

scandito dall'orologio di sistema, consente di verificare e confrontare le prestazioni computazionali di **R**:

```
> t0=Sys.time()
> a1=5+3
> a2=1+2
> a3=3+6
> a4=7+1
> a5=9+10
> a6=4*5
> Sys.time()-t0
Time difference of 0.0940001 secs

> t0=Sys.time()
> b1=c(2,3)+3
> b2=c(1,4)+c(2,7)
> b3=3+c(6,5,7,8,9,1)
> b4=(1:70)+6
> b5=(1:9000)+c(2,3)
> b6=(1:4000)*5
> Sys.time()-t0
Time difference of 0.09399986 secs
```

Capitolo IX

Matrici e array

Una matrice $n \times k$ può essere considerata come un vettore di lunghezza *n* i cui elementi sono vettori di lunghezza *k* oppure come un vettore di lunghezza *k* i cui elementi sono vettori tutti di lunghezza *n*.

Ciascuno degli elementi che compongono tale rettangolo di valori è univocamente individuato da una coppia di numeri interi, che costituiscono l'indice di riga e quello di colonna.

In **R** le matrici possono essere definite in svariati modi; il modo più semplice è quello di utilizzare direttamente la funzione `matrix()`, che crea una matrice a partire da un dato set di valori:

```
> m1=matrix(1:20,ncol=5)
> m1
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    5    9   13   17
[2,]    2    6   10   14   18
[3,]    3    7   11   15   19
[4,]    4    8   12   16   20
```

```
> m2=matrix(1:20,nrow=2)
> m2
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]    1    3    5    7    9   11   13   15   17    19
[2,]    2    4    6    8   10   12   14   16   18    20
```

Si osservi che, di default, la funzione `matrix()` opera riempiendo le matrici `m1` e `m2` per colonna. È ovviamente possibile modificare questa impostazione:

```
> m3=matrix(1:20,nrow=5, byrow=T)
```

B. Bertaccini, *Introduzione alla Statistica Computazionale con R*, ISBN (print) 978-88-6453-673-6, ISBN (online) 978-88-6453-674-3, CC BY 4.0, 2018 Firenze University Press.

```
> m3
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12
[4,]   13   14   15   16
[5,]   17   18   19   20
```

Si osservi inoltre che qualora il numero delle righe o delle colonne da generare non sia un divisore del numero totale di elementi indicati per creare la matrice, allora **R** genererà un messaggio di *warning*, avvertendo che il risultato prodotto potrebbe non essere quello desiderato. Nell'esempio sottostante, l'ultima cella della matrice viene riempita riutilizzando il primo elemento del vettore di valori `1:20`.

```
> m4=matrix(1:20,nrow=3, byrow=T)
Warning message:
In matrix(1:20, nrow = 3, byrow = T): data length [20] is not a sub-
multiple or multiple of the number of rows [3]

> m4
     [,1] [,2] [,3] [,4] [,5] [,6] [,7]
[1,]    1    2    3    4    5    6    7
[2,]    8    9   10   11   12   13   14
[3,]   15   16   17   18   19   20    1
```

Infine, se la funzione `matrix()` viene utilizzata senza il suo primo argomento, **R** genera una matrice di valori mancanti (*missing*), etichettati con NA. Il riempimento di tale matrice è sempre possibile a patto di utilizzare le parentesi quadre dopo il nome della matrice (altrimenti **R** riscriverebbe l'oggetto):

```
> m4=matrix(nrow=2,ncol=5)
> m4
     [,1] [,2] [,3] [,4] [,5]
[1,]   NA   NA   NA   NA   NA
[2,]   NA   NA   NA   NA   NA

> m4[]=5:14
> m4
     [,1] [,2] [,3] [,4] [,5]
[1,]    5    7    9   11   13
[2,]    6    8   10   12   14

> m4=5:14; # m4 adesso non è più una matrice ma un vettore
> m4
 [1]  5  6  7  8  9 10 11 12 13 14
```

In accordo con la definizione, le matrici possono essere create anche come risultato di operazioni di concatenazione (per colonna, mediante la funzione `cbind()`, o per riga, mediante la funzione `rbind()`) tra vettori di <u>lunghezza compatibile</u>:

```
> v1=1:5
> v2=6:10

> m5=cbind(v1,v2)
> m5
     v1 v2
[1,]  1  6
[2,]  2  7
[3,]  3  8
[4,]  4  9
[5,]  5 10


> m6=rbind(v1,v2)
> m6
   [,1] [,2] [,3] [,4] [,5]
v1    1    2    3    4    5
v2    6    7    8    9   10


> m7=cbind(1:10,c(v2,v2))
> m7
      [,1] [,2]
 [1,]    1    6
 [2,]    2    7
 [3,]    3    8
 [4,]    4    9
 [5,]    5   10
 [6,]    6    6
 [7,]    7    7
 [8,]    8    8
 [9,]    9    9
[10,]   10   10
```

Si osservi che la generazione di `m7`, pur se formalmente corretta, produce un risultato analogo a quello ottenibile dalla più semplice notazione `m7=cbind(1:10,v2)` perché la lunghezza di `v2` è un divisore della lunghezza del vettore `1:10`. Infatti:

```
> m7=cbind(1:10,v2)
> m7
         v2
 [1,]  1  6
 [2,]  2  7
 [3,]  3  8
 [4,]  4  9
 [5,]  5 10
 [6,]  6  6
 [7,]  7  7
 [8,]  8  8
 [9,]  9  9
[10,] 10 10
```

Le matrici possono essere generate anche per via indiretta sfruttando alcune proprietà delle matrici stesse cui si accede tramite le funzioni `dim()` (dimensione della matrice) o `diag()` (diagonale della matrice):

```
> m8=1:20
> dim(m8)=c(4,5)
> m8
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    5    9   13   17
[2,]    2    6   10   14   18
[3,]    3    7   11   15   19
[4,]    4    8   12   16   20
```

```
> m9=diag(1:4)
> m9
     [,1] [,2] [,3] [,4]
[1,]    1    0    0    0
[2,]    0    2    0    0
[3,]    0    0    3    0
[4,]    0    0    0    4
```

Come per i vettori, l'utilizzo delle parentesi quadre `[ ]` consente di selezionare elementi della matrice utilizzando il carattere virgola (`,`) per separare gli indici di riga e di colonna:

```
> m8[1,2]
[1] 5
> m9[4,4]
[1] 4
```

Si osservi che per selezionare tutti gli elementi di una riga o di una colonna occorre indicare, tra parentesi quadre, solo l’indice della riga (seguito da virgola), o della colonna (preceduto da virgola) che vogliamo ottenere:

```
> m8[1,]
[1]  1  5  9 13 17
> m9[,3]
[1] 0 0 3 0
```

Le regole di indicizzazione viste per i vettori valgono ovviamente anche per le matrici; con riferimento a `m8` generata in precedenza:

```
> m8[,-1]
     [,1] [,2] [,3] [,4]
[1,]    5    9   13   17
[2,]    6   10   14   18
[3,]    7   11   15   19
[4,]    8   12   16   20
```

```
> m8[2:4,]
     [,1] [,2] [,3] [,4] [,5]
[1,]    2    6   10   14   18
[2,]    3    7   11   15   19
[3,]    4    8   12   16   20

> m8>5
      [,1]  [,2] [,3] [,4] [,5]
[1,] FALSE FALSE TRUE TRUE TRUE
[2,] FALSE  TRUE TRUE TRUE TRUE
[3,] FALSE  TRUE TRUE TRUE TRUE
[4,] FALSE  TRUE TRUE TRUE TRUE
> m8[m8>5]
 [1]  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
> m8[,c(T,F,F,T,T)]
     [,1] [,2] [,3]
[1,]    1   13   17
[2,]    2   14   18
[3,]    3   15   19
[4,]    4   16   20
```

Come per i vettori, sono ovviamente ammissibili le operazioni tra matrici di dimensione analoga. Ad esempio, creando la matrice `m10=matrix(20:1,ncol=5)` possiamo eseguire:

```
> m1+m10
     [,1] [,2] [,3] [,4] [,5]
[1,]   21   21   21   21   21
[2,]   21   21   21   21   21
[3,]   21   21   21   21   21
[4,]   21   21   21   21   21
> m1-m10
     [,1] [,2] [,3] [,4] [,5]
[1,]  -19  -11   -3    5   13
[2,]  -17   -9   -1    7   15
[3,]  -15   -7    1    9   17
[4,]  -13   -5    3   11   19
> m1*m10
     [,1] [,2] [,3] [,4] [,5]
[1,]   20   80  108  104   68
[2,]   38   90  110   98   54
[3,]   54   98  110   90   38
[4,]   68  104  108   80   20
> m1/m10
          [,1]      [,2]      [,3]  [,4]  [,5]
[1,] 0.0500000 0.3125000 0.7500000 1.625  4.25
[2,] 0.1052632 0.4000000 0.9090909 2.000  6.00
[3,] 0.1666667 0.5000000 1.1000000 2.500  9.50
[4,] 0.2352941 0.6153846 1.3333333 3.200 20.00
```

Si osservi che nonostante la matrice m6 creata in precedenza sia di dimensione compatibile con m10, l'operazione elemento per elemento in questo caso non viene eseguita:

```
> m6
   [,1] [,2] [,3] [,4] [,5]
v1    1    2    3    4    5
v2    6    7    8    9   10
> m10+m6
Error in m10 + m6 : non-conformable arrays
```

Ovviamente, è però possibile sommare le prime due righe di m10 con m6:

```
> m10[1:2,] + m6
   [,1] [,2] [,3] [,4] [,5]
v1   21   18   15   12    9
v2   25   22   19   16   13
```

Oltre a queste operazioni è possibile eseguire in **R** il prodotto tra matrici (prodotto riga per colonna "%*%"), <u>a patto che la loro dimensione sia compatibile</u>; nell'esempio che segue, m5 è compatibile per tale prodotto con m10, in quanto il numero di colonne di m10 è uguale al numero di righe di m5 per cui:

```
> m10
     [,1] [,2] [,3] [,4] [,5]
[1,]   20   16   12    8    4
[2,]   19   15   11    7    3
[3,]   18   14   10    6    2
[4,]   17   13    9    5    1
> m5
     v1 v2
[1,]  1  6
[2,]  2  7
[3,]  3  8
[4,]  4  9
[5,]  5 10
> m10%*%m5
      v1  v2
[1,] 140 440
[2,] 125 400
[3,] 110 360
[4,]  95 320
```

R ovviamente dispone anche di funzioni per la gestione delle matrici:

Funzione	Descrizione

`dim(m)`	restituisce in un vettore la dimensione di una matrice `m`:
	`> dim(m8)`
	`[1] 4 5`

`diag(m)`	produce la diagonale di una matrice `m` (anche non quadrata):
	`> diag(m8)`
	`[1]  1  6 11 16`
	`> diag(rep(1,5))`
	`     [,1] [,2] [,3] [,4] [,5]`
	`[1,]    1    0    0    0    0`
	`[2,]    0    1    0    0    0`
	`[3,]    0    0    1    0    0`
	`[4,]    0    0    0    1    0`
	`[5,]    0    0    0    0    1`

`lower.tri(m, diag = FALSE)`
`upper.tri(m, diag = FALSE)`
	generano una matrice di valori logici (T, F) in cui i TRUE indicano rispettivamente gli elementi del triangolo superiore o inferiore:
	`> lower.tri(m8)`
	`      [,1]  [,2]  [,3]  [,4]  [,5]`
	`[1,] FALSE FALSE FALSE FALSE FALSE`
	`[2,]  TRUE FALSE FALSE FALSE FALSE`
	`[3,]  TRUE  TRUE FALSE FALSE FALSE`
	`[4,]  TRUE  TRUE  TRUE FALSE FALSE`
	`> m8[lower.tri(m8)]=NA`
	`> m8`
	`     [,1] [,2] [,3] [,4] [,5]`
	`[1,]    1    5    9   13   17`
	`[2,]   NA    6   10   14   18`
	`[3,]   NA   NA   11   15   19`
	`[4,]   NA   NA   NA   16   20`

`t(m)`	produce la trasposta della matrice `m`:
	`> t(m8)`
	`     [,1] [,2] [,3] [,4]`
	`[1,]    1   NA   NA   NA`
	`[2,]    5    6   NA   NA`
	`[3,]    9   10   11   NA`
	`[4,]   13   14   15   16`
	`[5,]   17   18   19   20`

`det(m)`	calcola il determinante della matrice `m` (<u>solo se `m` è quadrata</u>):
	`> det(m8)`
	`Error in determinant.matrix(x, ...) :`
	`  'x' must be a square matrix`
	`> det(m9)`
	`[1] 24`

<table>
<tr>
<td><code>solve(m)</code></td>
<td>produce l’inversa di una matrice <code>m</code>
(<u>solo se <code>m</code> è quadrata</u>):
<pre>
> solve(m9)
     [,1] [,2]   [,3] [,4]
[1,]   1 0.0 0.0000000 0.00
[2,]   0 0.5 0.0000000 0.00
[3,]   0 0.0 0.3333333 0.00
[4,]   0 0.0 0.0000000 0.25

> m9%*%solve(m9)
     [,1] [,2] [,3] [,4]
[1,]   1   0   0   0
[2,]   0   1   0   0
[3,]   0   0   1   0
[4,]   0   0   0   1
</pre>
</td>
</tr>
</table>

Esercizio: riprodurre in una matrice le tabelline delle moltiplicazioni per tutti i valori da 1 a 12

```
> (1:12) %*% t(1:12)
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12]
 [1,]    1    2    3    4    5    6    7    8    9    10    11    12
 [2,]    2    4    6    8   10   12   14   16   18    20    22    24
 [3,]    3    6    9   12   15   18   21   24   27    30    33    36
 [4,]    4    8   12   16   20   24   28   32   36    40    44    48
 [5,]    5   10   15   20   25   30   35   40   45    50    55    60
 [6,]    6   12   18   24   30   36   42   48   54    60    66    72
 [7,]    7   14   21   28   35   42   49   56   63    70    77    84
 [8,]    8   16   24   32   40   48   56   64   72    80    88    96
 [9,]    9   18   27   36   45   54   63   72   81    90    99   108
[10,]   10   20   30   40   50   60   70   80   90   100   110   120
[11,]   11   22   33   44   55   66   77   88   99   110   121   132
[12,]   12   24   36   48   60   72   84   96  108   120   132   144
```

Se le matrici possono intendersi come vettori di vettori, gli array costituiscono una estensione multidimensionale delle matrici. In un array ogni suo elemento è individuato da un vettore di indici: in questo senso una matrice può essere considerata come un array bidimensionale per cui per gli array valgono le stesse regole di operabilità viste per le matrici.

In **R,** per generare un array si può utilizzare direttamente la funzione `array()`, a partire da un dato set di valori, analogamente a quanto visto per la funzione `matrix()`:

```
> a<-array(1:24, dim=c(3,4,2))
```

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

, , 2
     [,1] [,2] [,3] [,4]
[1,]   13   16   19   22
[2,]   14   17   20   23
[3,]   15   18   21   24

> a[1,,]
     [,1] [,2]
[1,]    1   13
[2,]    4   16
[3,]    7   19
[4,]   10   22
```

Capitolo X

Liste

A differenza di vettori, matrici e più in generale array i cui elementi sono tutti dello stesso tipo (numero o stringhe carattere), una lista (list) è un oggetto particolare composto da una collezione di altri oggetti, anche differenti tra loro.

Una lista può essere creata tramite il comando `list` ed il numero degli oggetti che la costituiscono ne definisce la sua lunghezza; l'estrazione di oggetti appartenenti alla lista stessa avviene mediante una notazione che prevede l'impiego delle doppie parentesi quadre "`[[ ]]`" e consente di trattare ogni singola componente come un oggetto separato e indipendente su cui potere effettuare tutte le operazioni consentite con il tipo di oggetto estratto:

```
> l<-list(matrix(1:10,nrow=2),seq(-2,2,.5),c("Paperino","è","bello"))
> l
[[1]]
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10

[[2]]
[1] -2.0 -1.5 -1.0 -0.5  0.0  0.5  1.0  1.5  2.0

[[3]]
[1] "Paperino" "è"        "bello"

> length(l)
[1] 3

> l[[1]]
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
```

B. Bertaccini, *Introduzione alla Statistica Computazionale con R*, ISBN (print) 978-88-6453-673-6, ISBN (online) 978-88-6453-674-3, CC BY 4.0, 2018 Firenze University Press.

```
> dim(l[[1]])
[1] 2 5
> t(l[[2]])%*%l[[2]]
     [,1]
[1,]   15
> paste(l[[3]][1],l[[3]][2],l[[3]][3],sep=" ")
[1] "Paperino è bello"
```

La funzione `paste()` consente di concatenare stringhe di caratteri, specificando l'eventuale carattere separatore.

Si osservi che è possibile assegnare un nome a ciascun elemento della lista. Una volta assegnati i nomi, è possibile accedere ad ogni elemento della lista invocandone direttamente il nome, attraverso una notazione che prevede l'impiego del simbolo $:

```
> names(l)=c("elem.1","elem.2","elem.3")
> names(l)
[1] "elem.1" "elem.2" "elem.3"
> l
$elem.1
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
$elem.2
[1] -2.0 -1.5 -1.0 -0.5  0.0  0.5  1.0  1.5  2.0
$elem.3
[1] "Paperino" "è"        "bello"
> l$elem.3
[1] "Paperino" "è"        "bello"
> l$elem.3[1]
[1] "Paperino"
```

Infine, è possibile esaminare la struttura di una lista utilizzando la funzione `str()`:

```
> str(l)
List of 3
 $ : int [1:2, 1:5] 1 2 3 4 5 6 7 8 9 10
 $ : num [1:9] -2 -1.5 -1 -0.5 0 0.5 1 1.5 2
 $ : chr [1:3] "Paperino" "è" "bello"
```

Capitolo XI

Dataframe

Il dataframe è uno degli oggetti più importanti di **R**, se considerato come pacchetto statistico deputato alla gestione e all'analisi dei dati. Il dataframe rappresenta la tabella statistica dei dati grezzi: ad ogni sua riga corrisponde una osservazione (caso statistico) e ad ogni colonna una variabile rilevata su tale osservazione. Assume una conformazione identica ad una matrice (ovvero rettangolare), ma viene di fatto trattato da **R** come una lista, per cui, come visto in precedenza, `length()` restituisce il numero delle variabili, `names()` i rispettivi nomi, `row.names()` gli eventuali nomi assegnati ai casi di studio. Si osservi che queste ultime due funzioni possono essere utilizzate anche in assegnazione. È possibile costruire un dataframe direttamente con la funzione `data.frame()`:

```
> DF=data.frame(sesso=rep(c("M","F"),5), peso=71:80,
  altezza=seq(170,188,by=2))

> DF
   sesso peso altezza
1      M   71     170
2      F   72     172
3      M   73     174
4      F   74     176
5      M   75     178
6      F   76     180
7      M   77     182
8      F   78     184
9      M   79     186
10     F   80     188

> length(DF)
[1] 3
```

B. Bertaccini, *Introduzione alla Statistica Computazionale con R*, ISBN (print) 978-88-6453-673-6, ISBN (online) 978-88-6453-674-3, CC BY 4.0, 2018 Firenze University Press.

```
> names(DF)
[1] "sesso"   "peso"    "altezza"

> row.names(DF)
 [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10"

> row.names(DF)=21:30
> DF
   sesso peso altezza
21     M   71     170
22     F   72     172
23     M   73     174
24     F   74     176
25     M   75     178
26     F   76     180
27     M   77     182
28     F   78     184
29     M   79     186
30     F   80     188
```

A differenza di vettori, matrici ed array, è possibile aggiungere una variabile ad un dataframe indicandone direttamente il nome preceduto dal nome del dataframe e dal simbolo “$” ed effettuare operazioni sui dati ricorrendo a questa particolare notazione.

Nell’esempio successivo, viene aggiunta al dataframe la variabile `ratio` e a tale variabile viene assegnato il valore di rapporto tra peso ed altezza per i soli individui di sesso maschile (l’istruzione `DF$sesso=="M"` è utilizzata per selezionare solo i casi in corrispondenza dei valori `TRUE`):

```
> DF$ratio=NA

> DF
   sesso peso altezza ratio
21     M   71     170    NA
22     F   72     172    NA
23     M   73     174    NA
24     F   74     176    NA
25     M   75     178    NA
26     F   76     180    NA
27     M   77     182    NA
28     F   78     184    NA
29     M   79     186    NA
30     F   80     188    NA

> DF$sesso=="M"
 [1]  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE

>DF$ratio[DF$sesso=="M"]=
         DF$peso[DF$sesso=="M"]/DF$altezza[DF$sesso=="M"]
```

```
> DF
   sesso peso altezza     ratio
21     M   71     170 0.4176471
22     F   72     172        NA
23     M   73     174 0.4195402
24     F   74     176        NA
25     M   75     178 0.4213483
26     F   76     180        NA
27     M   77     182 0.4230769
28     F   78     184        NA
29     M   79     186 0.4247312
30     F   80     188        NA
```

Le variabili possono comunque sempre essere selezionate ricorrendo alla notazione tipica delle matrici tramite indice di colonna tra parentesi quadre, oppure sostituendo l'indice al nome della variabile se scritto tra virgolette. Il risultato di tale operazione sarà comunque sempre un vettore:

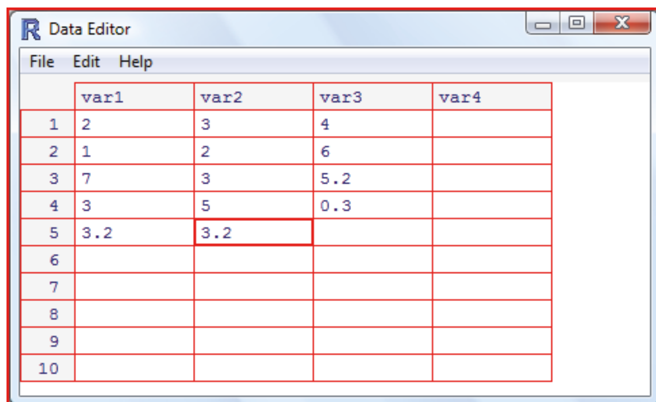
```
> DF$ratio
 [1] 0.4176471        NA 0.4195402        NA 0.4213483        NA
     0.4230769        NA 0.4247312        NA
> DF[,4]
 [1] 0.4176471        NA 0.4195402        NA 0.4213483        NA
     0.4230769        NA 0.4247312        NA
> DF[,"ratio"]
 [1] 0.4176471        NA 0.4195402        NA 0.4213483        NA
     0.4230769        NA 0.4247312        NA
```

Ovviamente la funzione `data.frame()` non esaurisce i possibili modi di riprodurre in **R** un tabella statistica casi-variabili; un modo estremamente semplice di ottenere un dataframe è forzare una matrice ad assumere le proprietà di tale particolare struttura dati; riconsiderando la matrice `m3` generata all'inizio del capitolo 9:

```
> dfm3=as.data.frame(m3)
> names(dfm3)=LETTERS[1:4]
> dfm3
   A  B  C  D
1  1  2  3  4
2  5  6  7  8
3  9 10 11 12
4 13 14 15 16
5 17 18 19 20
```

Un'altra interessante opportunità fornita da **R** è quella di generare un dataframe direttamente tramite data entry, sfruttando le proprietà dell'editor interno di generazione degli oggetti, attivabile tramite la funzione `fix()`. L'editor interno richiamato dalla funzione, in corrispondenza di un dataframe, assume la conformazione tipica di un foglio elettronico:

```
> DF2=data.frame()
> fix(DF2)
```



	var1	var2	var3	var4
1	2	3	4	
2	1	2	6	
3	7	3	5.2	
4	3	5	0.3	
5	3.2	3.2		
6				
7				
8				
9				
10				

Una volta chiuso l'editor, è ovviamente possibile richiamare l'oggetto creato:

```
> DF2
  var1 var2 var3
1  2.0  3.0  4.0
2  1.0  2.0  6.0
3  7.0  3.0  5.2
4  3.0  5.0  0.3
5  3.2  3.2   NA
```

Nelle librerie di **R** sono disponibili (sotto forma di dataframe) molti dataset d'esempio. Per richiamarli occorre utilizzare la funzione `data()` che trasferisce il dataset desiderato all'interno dell'area di lavoro, in modo che sia poi semplicemente richiamabile tramite il suo nome. Richiamiamo ad esempio, il dataset `iris`, relativo ad un esperimento di misurazione (lunghezza larghezza) dei petali e dei sepali di 3 differenti specie di iris (50 iris per specie):

```
> data(iris)
> iris[1:2,]
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa

> unique(iris$Species)
[1] setosa     versicolor virginica
Levels: setosa versicolor virginica
```

Infine, le funzioni `head()` e `tail()` consentono (convenientemente quando il dataset è molto grande) di "dare uno sguardo" rispettivamente alla parte iniziale o finale del dataset:

```
> tail(iris)
    Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
145          6.7         3.3          5.7         2.5 virginica
146          6.7         3.0          5.2         2.3 virginica
147          6.3         2.5          5.0         1.9 virginica
148          6.5         3.0          5.2         2.0 virginica
149          6.2         3.4          5.4         2.3 virginica
150          5.9         3.0          5.1         1.8 virginica
```

Spesso e volentieri emerge però la necessità di acquisire da una fonte esterna la tabella dei dati statistici che si vuole analizzare. In questo caso, la procedura di importazione più semplice da seguire è quella di salvare preventivamente la tabella dei dati statistici in formato testo[11], ricorrendo eventualmente per questa operazione all'utilizzo di un foglio elettronico esterno (tipo MS Excel o OpenOffice Calc), e di utilizzare la funzione `read.table()` per la procedura di importazione in **R**.

Nell'esempio che segue, il file studenti.txt (salvato nella directory "prova" del disco "D") è stato generato tramite MS Excel e salvato in formato testo separato da tabulazione. Tale file viene importato in **R**, e convertito in un dataframe, secondo la seguente sintassi[12]:

```
> DF3=read.table("D:\\prova\\studenti.txt",header=T,sep="\t",dec=",")

> DF3
   sesso peso altezza ratio
1      1   80     182  2.28
2      2   70     172  2.46
3      1   70     170  2.43
4      2   55     160  2.91
5      1   85     184  2.16
6      2   57     162  2.84
7      1   82     183  2.23
8      2   63     167  2.65
9      2   61     171  2.80
10     1   82     178  2.17

> DF3$sesso
 [1] 1 2 1 2 1 2 1 2 2 1
```

Sulla funzione `read.table()` sono a questo punto necessarie alcune puntualizzazioni:

a) l'argomento `header=T` (TRUE) specifica che la prima linea del file di testo contiene i nomi delle variabili per cui non deve essere considerata nell'acquisizione dell'area dati, ma solo nell'assegnazione dei nomi alle variabili lette;

[11] I due standard più diffusi sono il formato testo separato da tabulazione o il formato testo separato da virgola.

[12] Si osservi la necessità di raddoppiare il carattere "\" che definisce in Windows il percorso di cartelle e sottocartelle (*path*). Sotto Linux tale coppia di caratteri è sostituita dal solo carattere "/".

b) l'argomento `sep="\t"` indica che il file è in formato testo separato da tabulazione; se il carattere separatore fosse stato la virgola "," o il punto e virgola";" sarebbe stato sufficiente specificare tale carattere tra virgolette;
c) l'argomento facoltativo `na.strings="NA"` può essere particolarmente utile se nel file sono presenti valori mancanti, che in questo caso verranno individuati con NA;
d) l'argomento `dec=","` specifica il tipo di carattere utilizzato nel file per separare i decimali;
e) l'argomento facoltativo `rows=[numero]` specifica il numero di righe da leggere qualora il file sia di grosse dimensioni;
f) l'argomento facoltativo `skip=[numero]` specifica il numero iniziale di righe da non importare qualora il file sia di grosse dimensioni.

Vale a questo punto certamente la pena soffermare l'attenzione sulla libreria `data.table` (pubblicata sul CRAN per la prima volta nel 2006, ma con un numero esponenziale di download negli ultimi anni) che mette a disposizione funzioni per l'acquisizione e la manipolazione delle fonti dati che si rivelano estremamente più rapide ed efficaci[13] delle analoghe presenti nella libreria base di **R**. In estrema sintesi, le funzioni della libreria creano oggetti avanzati rispetto a quelli della classe `data.frame`.

```
> library(data.table)
> DT = data.table(a=letters[1:10], b = 1:10, c = 10:1)
> DT
    a  b  c
 1: a  1 10
 2: b  2  9
 3: c  3  8
 4: d  4  7
 5: e  5  6
 6: f  6  5
 7: g  7  4
 8: h  8  3
 9: i  9  2
10: j 10  1
```

Fin qui, tutto "quasi" analogo alla funzione `data.frame()`, con le uniche differenze che le colonne di tipo carattere non vengono convertite in tipo "`factor`" di default, che i numeri di riga sono stampati con il carattere ":" a separare la prima colonna di dati e che, se i numeri di riga eccedono il massimo numero di righe stampabili (default = 100), vengono automaticamente riprodotte le prime 5 e le ultime 5 righe del dataset. Ovviamente, un oggetto esistente può essere sempre convertito in uno della classe `data.table` ricorrendo alla funzione `as.data.table()`, conformemente a quanto fatto con la matrice `m3` a pag. 51.

[13] *"Briefly, if you are interested in reducing* programming *and* compute *time tremendously, then this package is for you."*

Ma la novità più interessante di questa libreria è certamente, come detto, l'estrema rapidità ed efficienza con cui riesce ad importare archivi di rilevanti dimensioni tramite la funzione `fread()`. Nell'esempio sottostante, le performance di questa funzione sono confrontate con quelle della tradizionale `read.table()` durante l'importazione di un file di testo di circa 300 Mbyte, contenente una matrice di valori numerici di dimensione 12656250 x 12:

```
> t0=Sys.time()
> dati=read.table("D:\\prova\\dati.txt",header=T)
> Sys.time()-t0
Time difference of 45.02853 secs


> t0=Sys.time()
> dati2=fread("D:\\prova\\dati.txt",header=T)
Read 25.8% of 12656250 rows
Read 42.6% of 12656250 rows
Read 54.2% of 12656250 rows
Read 71.7% of 12656250 rows
Read 85.6% of 12656250 rows
Read 99.2% of 12656250 rows
Read 12656250 rows and 12 (of 12) columns from 0.295 GB file in
00:00:08
> Sys.time()-t0
Time difference of 7.281237 secs
```

Questi risultati sono resi possibili da più concisa ed efficiente filosofia di memorizzazione e organizzazione dei dati che si traduce in una sintassi innovativa. Un dataset della classe `data.table` è infatti un oggetto della forma:

`DT[i,j,by]`

dove:

- `i` è l'indice di selezione delle righe interessate da una certa operazione;
- `j` è l'indice che specifica le variabili interessate ed il tipo di operazione;
- `by` è l'indice che specifica l'eventuale criterio di raggruppamento.

Il codice sottostante illustra, tramite esempi, le differenze di sintassi per l'esecuzione dello stesso tipo di operazioni:

```
> DF=data.frame(id=1:5,cod=c("a","a","b","b","c"), a=10:6, b=101:105)
> DF
  id cod  a   b
1  1   a 10 101
2  2   a  9 102
3  3   b  8 103
4  4   b  7 104
5  5   c  6 105
```

```
> DT=as.data.table(DF)
> DT
   id cod  a   b
1:  1   a 10 101
2:  2   a  9 102
3:  3   b  8 103
4:  4   b  7 104
5:  5   c  6 105
```

operazioni di riduzione della base dati:

oggetti di classe `data.frame`	*oggetti di classe* `data.table`

`> DF[DF$id>2,]`
`  id cod a   b`
`3  3   b 8 103`
`4  4   b 7 104`
`5  5   c 6 105`
	`> DT[id>2]`
	`   id cod a   b`
	`1:  3   b 8 103`
	`2:  4   b 7 104`
	`3:  5   c 6 105`

operazioni sulle variabili:

oggetti di classe `data.frame`	*oggetti di classe* `data.table`

`> DF[,'a']`
`[1] 10  9  8  7  6`
`> sum(DF[,'a'])`
`[1] 40`
	`> DT[,a]`
	`[1] 10  9  8  7  6`
	`> DT[,sum(a)]`
	`[1] 40`

operazioni sulle variabili di una base dati ridotta:

oggetti di classe `data.frame`	*oggetti di classe* `data.table`
`> sum(DF[DF$id>2,'a'])` `[1] 21`	`> DT[id>2,sum(a)]` `[1] 21`

operazioni sulle variabili e raggruppamento:

oggetti di classe `data.frame`	*oggetti di classe* `data.table`

`> aggregate(a~cod, data=DF, sum)`
`  cod  a`
`1   a 19`
`2   b 15`
`3   c  6`
	`> DT[,sum(a),cod]`
	`   cod V1`
	`1:   a 19`
	`2:   b 15`
	`3:   c  6`

operazioni sulle variabili di una base dati ridotta e raggruppamento:

oggetti di classe `data.frame`	*oggetti di classe* `data.table`

`> aggregate(a~cod,`
`data=DF[DF$id>1,], sum)`
`  cod  a`
`1   a  9`
`2   b 15`
`3   c  6`
	`> DT[id>1,sum(a),cod]`
	`   cod V1`
	`1:   a  9`
	`2:   b 15`
	`3:   c  6`

Gli ultimi due esempi chiariscono definitivamente l'efficienza sintattica delle strutture `data.table`. Per eseguire le stesse operazioni disponendo di un `data.frame`, occorre ricorrere alla funzione `aggregate()`, il cui funzionamento è abbastanza complesso in relazione agli obiettivi che questo testo si è posto. La libreria `data.table` meriterebbe ovviamente un'attenzione maggiore di quella fin qui dedicatale; per gli stessi motivi di cui sopra, per approfondimenti sulle caratteristiche e possibilità offerte da questa eccellente libreria si rimanda necessariamente al sito: https://cran.r-project.org/web/packages/data.table/

In **R** è comunque possibile importare direttamente una fonte dati salvata con MS Excel (*.xls oppure *.xlsx) o con MS Access (*mdb o *.accdb), evitando il passaggio intermedio di salvare il foglio dati di interesse in formato testo. Tale procedura d'importazione richiede l'utilizzo di una libreria esterna denominata `RODBC`, che deve essere preventivamente selezionata e installata da uno dei server "mirror" del CRAN. Questa libreria consente di stabilire un collegamento tramite l'API (Application Programming Interface) ODBC (Open Database Connectivity) ai *pc-files* e ai database.

Ad esempio, la funzione per stabilire una connessione ad un file Excel salvato secondo il nuovo formato 2007/2010 " *.xlsx " è `odbcConnectExcel2007()` che richiede come unico parametro in ingresso il percorso in cui è memorizzato il file. Tramite la funzione `sqlFetch()` è possibile indicare a **R** quale foglio di lavoro deve essere importato. Si osservi che è buona norma chiudere la connessione una volta completata la procedura di importazione (utilizzando la funzione `odbcClose()`).

```
> library(RODBC)
> path="D:\\prova\\studenti.xlsx"
> channel <- odbcConnectExcel2007(path)
> dati <- sqlFetch(channel,"Foglio1")
> odbcClose(channel)
```

```
> dati
   sesso peso altezza    ratio
1      1   80     182 2.275000
2      2   70     172 2.457143
3      1   70     170 2.428571
4      2   55     160 2.909091
5      1   85     184 2.164706
6      2   57     162 2.842105
7      1   82     183 2.231707
8      2   63     167 2.650794
9      2   61     171 2.803279
10     1   82     178 2.170732
```

Si osservi come il ricorso alla connessione ODBC consenta di evitare la specificazione dell'eventuale riga di intestazione, del formato delle variabili nonché dei caratteri di delimitazione e di separazione decimale.

Capitolo XII

Tipi di variabili

In Statistica, come noto, si distinguono due principali gruppi di variabili: quelle numeriche e quelle categoriali (ordinali o non ordinali). Negli esempi illustrati in precedenza, sono state considerate quasi esclusivamente variabili di tipo numerico.

Come visto, una variabile categoriale potrebbe essere inserita direttamente mediante le sue "etichette", ovvero ad esempio `sesso<-c("M","M","F",..)`. Sebbene tale procedimento sia del tutto ammissibile, esiste un modo "più corretto" per definire una variabile categoriale e per specificarne le sue modalità: tale metodo prevede l'impiego della funzione `factor()`, concepita per associare valori numerici ad etichette:

```
> x<-factor(c(1,4,3,3,2,2,1))
> x
[1] 1 4 3 3 2 2 1
Levels: 1 2 3 4
```

```
> x<-factor(c(1,4,3,3,2,2,1), labels=c("A","B","C","D"))
> x
[1] A D C C B B A
Levels: A B C D
```

Si osservi che l'argomento `labels` della funzione `factor()` consente semplicemente di stabilire una corrispondenza tra il numero e la sua etichetta, in modo tale che **R** visualizzi i nomi delle categorie invece che i semplici numeri.

Si osservi inoltre che durante l'importazione di fonti informative esterne, la funzione `read.table()` genera di default un dataframe contenente tutte variabili numeriche. Se nel file di dati da importare sono però presenti variabili categoriali con modalità espresse da numeri piuttosto che esplicitamente da etichette di categoria, è sufficiente applicare la funzione `factor()` a tali variabili per modificarne il tipo (da numerica a categoriale).

B. Bertaccini, *Introduzione alla Statistica Computazionale con R*, ISBN (print) 978-88-6453-673-6, ISBN (online) 978-88-6453-674-3, CC BY 4.0, 2018 Firenze University Press.

Facendo riferimento al file importato e memorizzato nel dataframe `DF3` (la cui procedura è stata presentata nel capitolo precedente), la variabile `sesso`, come visto, è stata acquisita in formato numerico. Per modificarne il tipo:

```
> DF3$sesso=factor(DF3$sesso, labels=c("M","F"))
> DF3
   sesso peso altezza ratio
1      M   80     182  2.28
2      F   70     172  2.46
3      M   70     170  2.43
4      F   55     160  2.91
5      M   85     184  2.16
6      F   57     162  2.84
7      M   82     183  2.23
8      F   63     167  2.65
9      F   61     171  2.80
10     M   82     178  2.17

> DF3$sesso
 [1] M F M F M F M F F M
Levels: M F
```

L'utilizzo della funzione `factor()` tornerà utile in molte situazioni, dalle analisi statistiche descrittive di base alla produzione di grafici.

Infine, a volte può essere necessario operare una categorizzazione di una variabile quantitativa, ripartendone in classi il campo di variazione. Tale operazione è possibile ricorrendo alla funzione `cut()`:

```
> DF3$classi.ratio=cut(DF3$ratio, breaks=c(2,2.33,2.66,3))
> DF3
   sesso peso altezza ratio classi.ratio
1      M   80     182  2.28     (2,2.33]
2      F   70     172  2.46  (2.33,2.66]
3      M   70     170  2.43  (2.33,2.66]
4      F   55     160  2.91     (2.66,3]
5      M   85     184  2.16     (2,2.33]
6      F   57     162  2.84     (2.66,3]
7      M   82     183  2.23     (2,2.33]
8      F   63     167  2.65  (2.33,2.66]
9      F   61     171  2.80     (2.66,3]
10     M   82     178  2.17     (2,2.33]

> DF3$classi.ratio=cut(DF3$ratio,breaks=c(2,2.33,2.66,3),
   labels=c("basso","medio","alto"))
```

```
> DF3
   sesso peso altezza ratio classi.ratio
1      M   80     182  2.28        basso
2      F   70     172  2.46        medio
3      M   70     170  2.43        medio
4      F   55     160  2.91         alto
5      M   85     184  2.16        basso
6      F   57     162  2.84         alto
7      M   82     183  2.23        basso
8      F   63     167  2.65        medio
9      F   61     171  2.80         alto
10     M   82     178  2.17        basso
```

Come illustrato dall'esempio precedente, la funzione `cut()` produce per default una variabile categoriale dividendo una variabile numerica in intervalli aperti a sinistra e chiusi a destra. Si osservi che gli estremi degli intervalli possono o meno essere inclusi negli stessi in funzione di come vengono specificati gli argomenti opzionali della funzione. Per maggiori informazioni si consulti la guida, digitando al *prompt* dei comandi `?cut`.

Capitolo XIII

Elementi di programmazione strutturata

In precedenza sono state illustrate alcune funzioni per la soluzione di semplici problemi algebrici. In questo paragrafo saranno illustrate le istruzioni base della programmazione strutturata comprese da **R**, istruzioni che consentono di estendere le potenzialità computazionali del pacchetto permettendo l'implementazione e la personalizzazione di particolari algoritmi oltre i confini delle funzioni di libreria disponibili nel pacchetto base e nelle librerie aggiuntive.

Al fine di introdurre l'organizzazione dei programmi in blocchi, ogni linguaggio strutturato deve poter gestire la "sequenza informatica" (ovvero un elenco ordinato di istruzioni semplici e/o di strutture di controllo); per fare ciò deve prevedere almeno una struttura di tipo *alternativa*, e almeno una struttura di tipo *iterativa*. Si osservi che le strutture illustrate in questo paragrafo sono sempre comprese da **R**, se specificate all'interno delle funzioni utente; in altre parole, alcuni costrutti potrebbero non essere compresi da **R** se si cerca di editarli o incollarli direttamente da *prompt* di comando.

La principale struttura *alternativa* di **R** si basa sul costrutto sintattico:

```
if (condizione) {
            ...............
            # blocco istruzioni se la condizione è vera
            ...............
            }
       [else{
            ...............
            # blocco istruzioni se la condizione è falsa
            ...............
            }]
```

B. Bertaccini, *Introduzione alla Statistica Computazionale con R*, ISBN (print) 978-88-6453-673-6, ISBN (online) 978-88-6453-674-3, CC BY 4.0, 2018 Firenze University Press.

R non prevede la presenza dell'istruzione THEN come invece accade per altri linguaggi di programmazione (es. FORTRAN). Le parentesi graffe delimitano il blocco di istruzioni da eseguire qualora la condizione su cui si basa il costrutto risulti vera oppure falsa, e sono necessarie qualora le istruzioni di tali blocchi siano almeno due. Si osservi che il blocco `else` è scritto fra parentesi quadre in quanto blocco facoltativo.

Esempio:

```
> a=2
> if(a==2) print("Ciao mondo") else print("Ciao Marte")
[1] "Ciao mondo"
> a=3
> if(a==2) print("Ciao mondo") else print("Ciao Marte")
[1] "Ciao Marte"
```

Come segnalato in precedenza, **R** comprende tutto il costrutto "*alternativa*" perché scritto su un'unica riga. Si provi, in alternativa, a scrivere le seguenti istruzioni all'interno della finestra dello script editor di **R**, su più righe rispettando le regole di *indentazione*,

```
a=3
if (a==2) print("Ciao mondo")
          else ("Ciao Marte")
```

per poi copiarle ed incollarle nel *prompt* di comando. Il risultato è il seguente messaggio d'errore:

```
> a=3
> if (a==2) print("Ciao mondo")
>     else ("Ciao Marte")
Error: unexpected 'else' in "    else"
```

In altri termini, **R** non comprende l'istruzione `else` se questa viene incollata o digitata nel prompt su una riga differente dall'istruzione `if`.

La principale struttura *iterativa* di **R** si basa sul costrutto sintattico:

```
for (variabile in vettore) {
            ...............
            # blocco istruzioni eseguite per tutti i differenti valori
            presenti in vettore
            ...............
            }
```

Anche in questo caso, così come per il blocco *alternativa*, le parentesi graffe delimitano il blocco di istruzioni da iterare, e sono necessarie qualora le istruzioni di tali blocchi siano almeno due.

Esempi:

```
> v=1:10
> for (j in v)  print (j)
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
[1] 10

> v=c(2,5,8)
> for (j in v) print (j)
[1] 2
[1] 5
[1] 8


> v=c("Paperino","è","bello")
> for (j in 1:length(v))
  print (paste("stringa n°",j," del vettore v: ",v[j],sep=""))
[1] "stringa n°1 del vettore v: Paperino"
[1] "stringa n°2 del vettore v: è"
[1] "stringa n°3 del vettore v: bello"
```

Esercizio: costruire una funzione in grado di indicare il numero di colonne di una matrice (parametro in ingresso alla funzione) contenenti valori *missing* (ovvero `NA`).

```
mis.col=function(a){
if (!is.matrix(a))
      stop("il parametro in ingresso non è una matrice")

ncol=dim(a)[2]
c=0
for (j in 1:ncol){
      if (sum(is.na(a[,j]))>0) c=c+1
      }
print(paste("numero di colonne con dati mancanti:",c))
}

> a=3
```

```
> mis.col(a)
Error in mis.col(a) : il parametro in ingresso non è una matrice!
```

```
> m8=matrix(1:20,nrow=4)
> m8[lower.tri(m8)]=NA
> m8
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    5    9   13   17
[2,]   NA    6   10   14   18
[3,]   NA   NA   11   15   19
[4,]   NA   NA   NA   16   20
> mis.col(m8)
[1] "numero di colonne con dati mancanti: 3"
```

Esercizio: ordinare un vettore secondo questa procedura: ogni coppia di elementi nel vettore viene comparata e se essi sono nell'ordine sbagliato vengono invertiti. L'algoritmo scorre tutto il vettore finché non vengono più eseguiti scambi, situazione che indica che il vettore è ordinato.

```
my.sort=function(a){
if (!is.vector(a))
      stop("il parametro in ingresso non è un vettore")

nn=length(a)
for(i in 1:(nn-1))
      for(j in (i+1):nn)
            if(a[i]>a[j]){
                  aux=a[i]
                  a[i]=a[j]
                  a[j]=aux
                  }
return(a)
}

> v=sample(1:10)
> v
 [1]  9  7  4  2  1  5 10  3  8  6
> my.sort(v)
 [1]  1  2  3  4  5  6  7  8  9 10
```

La funzione `sample()`[14] consente di estrarre un generico campione casuale di x elementi da un vettore di lunghezza n. Di default, l'estrazione avviene senza reimmissione sebbene sia ovviamente possibile specificare la modalità di estrazione con reimmissione (impostando il parametro `replace=T`). Nello specifico, `sample(1:10)` restituisce una permutazione casuale del vettore `1:10`.

Si osservi che il risultato prodotto dalla funzione `my.sort()` è ovviamente lo stesso della funzione interna `sort()`. Quest'ultima risulta però nettamente più efficiente in termini di velocità di esecuzione, in quanto sfrutta algoritmi di ordinamento più efficienti abbinati alla logica vettoriale su cui è costruito **R**:

```
> v=sample(5000)

> t0=Sys.time()
> v1=sort(v)
> Sys.time()-t0
Time difference of 0.01499987 secs


> t1=Sys.time()
> v2=my.sort(v)
> Sys.time()-t1
Time difference of 59.264 secs
```

Una struttura *iterativa* alternativa tipica dei linguaggi di programmazione strutturata, compresa dall'interprete **R**, si basa sul costrutto sintattico:

```
while (condizione) {
            ...............
            # blocco istruzioni eseguite mentre la condizione è vera
            ...............
            }
```

Ancora una volta, le parentesi graffe delimitano il blocco di istruzioni da iterare, e sono necessarie qualora le istruzioni di tali blocchi siano almeno due.

La sintassi successiva mostra l'equivalenza tra le sintassi iterative *for* e *while* quando il numero di iterazioni da eseguire è stabilito a priori:

```
for (i in 1:10) {
            ...............
            # blocco istruzioni da iterare
            ...............
            }
```

equivalente a:

[14] Gli algoritmi di generazione dei numeri casuali e le funzioni connesse saranno approfonditi nel capitolo 17.

```
i=1
while (i<=10) {
            ...............
            # blocco istruzioni da iterare
            ...............
            i=i+1
            }
```

La struttura iterativa `while()` si rivela però particolarmente utile quando non è possibile stabilire a priori il numero delle iterazioni da effettuare.

Il codice successivo illustra il funzionamento del `while()`; il numero delle iterazioni varia in funzione delle estrazioni effettuate:

```
> a=0
> while(a<10){
+     a = a + sample(1:10,1)
+     print(a)
+     }
[1] 4
[1] 8
[1] 15
```

Un esempio da <u>NON imitare</u>:

```
a=0
while (a==0){
            print("questo potrebbe essere un virus!!!")
            }
```

Il codice precedente è un esempio di programmazione che viola le regole di implementazione degli algoritmi: il numero di iterazioni di un algoritmo deve essere sempre finito. Nel caso illustrato, invece, il codice produce quello che in informatica è identificato come "*loop infinito*"[15].

Esercizio (Ricerca Binaria): costruire una funzione di ricerca di un numero all'interno di un vettore ordinato di elementi, basato sulla seguente logica: poiché il vettore è ordinato, si inizia la ricerca non dal primo elemento, ma da quello centrale, cioè a metà del vettore. Si confronta questo elemento con quello cercato:

1. se corrisponde, la ricerca termina indicando che l'elemento è stato trovato;

[15] Nel linguaggio informatico, per *loop* infinito si intende un algoritmo o un frammento di codice formulato per mezzo della ripetizione di sé stesso un numero infinito di volte (infiniti cicli).

2. se è inferiore, la ricerca viene ripetuta sugli elementi precedenti (ovvero sulla prima metà del vettore), scartando quelli successivi;
3. se invece è superiore, la ricerca viene ripetuta sugli elementi successivi (ovvero sulla seconda metà del vettore), scartando quelli precedenti.

Quando tutti gli elementi sono stati scartati, la ricerca deve terminare indicando che il valore non è stato trovato.

```
ricbin = function(v,a){
if (!is.vector(v)) stop("first input parameter must be a vector")
if ( sum(v==sort(v)) < length(v) ) stop("vector must be ordered")
pos=0
flag=0
while (flag==0){
    meta=ceiling(length(v)/2)
    if (v[meta]==a){
      print(paste("trovato in posizione:", pos+meta))
      flag=1
      }
      else if (meta==1){
            print("non trovato")
            flag=1
            }
            else if (v[meta]>a) v=v[1:(meta-1)]
                  else{
                      pos=pos+meta
                      v=v[(meta+1):length(v)]
                      }
    }
}
```

Capitolo XIV

Loop functions: le funzioni della famiglia apply

R è un pacchetto statistico concepito secondo logica vettoriale. Per questo motivo, quando in sede di programmazione di un algoritmo si ritiene di dover ricorrere ad una struttura iterativa, la regola d'oro è quella di… "contare fino a 1000", per valutare la sussistenza di soluzioni alternative. Qualora si sia tentati di confermare il ricorso ad un ciclo `for()` o ad un `while()`, molti sviluppatori di **R** concordano (ed io con loro) che in tal caso la scelta migliore sarebbe quella di… ricominciare a contare. Si osservi inoltre che le strutture `for()` e `while()` sono difficilmente gestibili lavorando in modalità interattiva, ovvero direttamente dal prompt di comando.

Le regole dell'algebra in questo ci aiutano: sovente le istruzioni iterative possono essere efficientemente ed efficacemente sostituite da istruzioni che sfruttano le proprietà delle operazioni tra vettori e matrici. **R** mette comunque a disposizione una serie di strumenti estremamente potenti (le funzioni della famiglia `apply`) che implementano *loop* in forma compatta e che sono richiamabili in maniera interattiva.

La funzione `apply()` consente di valutare una generica funzione lungo i margini di un array. In particolare, è convenientemente usata per applicare una funzione a tutte le righe o a tutte le colonne di una matrice:

```
> x=matrix(sample(1:20),ncol=5)
> x
     [,1] [,2] [,3] [,4] [,5]
[1,]    4   17   16   19   11
[2,]    6    8   15   14   20
[3,]    2    5    3    1   10
[4,]   12    7   13   18    9
```

B. Bertaccini, *Introduzione alla Statistica Computazionale con R*, ISBN (print) 978-88-6453-673-6, ISBN (online) 978-88-6453-674-3, CC BY 4.0, 2018 Firenze University Press.

```
> apply(x,1,min)
[1] 4 6 1 7
```

```
> apply(x,2,sum)
[1] 24 37 47 52 50
```

In particolare, per la somma e la media lungo le righe o le colonne di una matrice, **R** propone 4 funzioni "scorciatoia"[16]:

- `rowSums(x)` coincide con `apply(x, 1, sum)`
- `rowMeans(x)` coincide con `apply(x, 1, mean)`
- `colSums(x)` coincide con `apply(x, 2, sum)`
- `colMeans(x)` coincide con `apply(x, 2, mean)`

```
> colSums(x)
[1] 24 37 47 52 50
```

La funzione `lapply()` applica, in maniera iterativa, una determinata funzione a tutti gli elementi di una lista. Il risultato della funzione `lapply()` è ancora una lista:

```
> x <- list(x1 = sample(1:50,20), x2 = matrix(1:100,nrow=10))
```

```
> l=lapply(x,mean)
```

```
> l
$x1
[1] 20.7
$x2
[1] 50.5
```

Esercizio: data una lista di matrici di differente dimensione, estrarre la diagonale principale di ciascuna matrice.

```
> x <- list(a = matrix(1:9, ncol=3), b = matrix(1:12, ncol=4),
c=matrix(sample(1:200,25),ncol=5))
```

```
> x
$a
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

[16] La funzione `mean()`, deputata al calcolo della media aritmetica, verrà ripresa nel capitolo 16, dedicato alle "Principali funzionalità per la descrizione statistica delle informazioni".

```
$b
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
$c
     [,1] [,2] [,3] [,4] [,5]
[1,]  144  124   35  135   22
[2,]   67  198   78   47   52
[3,]  156  137  149   50   18
[4,]   20  119   59   99  165
[5,]  115   29  179  158   41

> lapply(x,diag)
$a
[1] 1 5 9
$b
[1] 1 5 9
$c
[1] 144 198 149  99  41
```

Esercizio: data una lista di matrici di differente dimensione, estrarre il primo e terzo quartile della diagonale principale di ciascuna matrice.

```
> x <- list(a = matrix(1:9, ncol=3), b = matrix(1:12, ncol=4),
c=matrix(sample(1:200,25),ncol=5))
```

```
> x
$a
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
$b
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
$c
     [,1] [,2] [,3] [,4] [,5]
[1,]   93  120  162  192   37
[2,]  135  171  189  154   79
[3,]   91   73   33   11   99
[4,]   45  145  165   83   70
[5,]  113  116   80   20   18
```

```
> lapply(x, function(y) quantile(diag(y), prob=c(.25,.75)) )
$a
25% 75%
  3   7
$b
25% 75%
  3   7
$c
25% 75%
 33  93
```

La funzione `sapply()` opera in maniera analoga, cercando di semplificare, se possibile, la struttura dell'oggetto restituito. In particolare:

a) se il risultato della funzione `lapply()` è una lista composta da elementi di lunghezza 1, la funzione `sapply()` restituisce tali elementi in un vettore;
b) se il risultato della funzione `lapply()` è una lista composta da vettori tutti della stessa lunghezza (> 1), la funzione `sapply()` restituisce tali elementi in una matrice;
c) altrimenti, in tutti gli altri casi, la semplificazione non è possibile e la funzione `sapply()` restituisce una lista.

```
> sapply(x, diag)
$a
[1] 1 5 9
$b
[1] 1 5 9
$c
[1] 93 171 33 83 18
```

Nell'esempio precedente la semplificazione non ha avuto effetto perché la lista è composta da elementi di differente dimensione. Nella situazione successiva, invece la semplificazione ha effetto e la funzione `sapply()` restituisce una matrice:

```
> sapply(x, function(y) quantile(diag(y), prob=c(.25,.75)) )
    a b  c
25% 3 3 33
75% 7 7 93
```

La funzione `tapply()` si rivela invece utile quando è stata indotta una partizione su un vettore e si vuole valutare una determinata funzione su tutti i sottoinsiemi che definiscono tale partizione:

```
> x=sample(1:100)
> y=rep(letters[1:10],10)
> tapply(x,y,mean)
   a    b    c    d    e    f    g    h    i    j
41.0 61.5 56.0 51.0 49.0 53.6 38.9 57.0 47.9 49.1
```

Il parametro `simplify = FALSE` può essere utilizzato per ottenere la restituzione dello stesso risultato senza semplificazione. In tal caso, analogamente al funzionamento di `lapply()`, l'oggetto restituito è una lista:

```
> tapply(x,y,mean,simplify=F)[1:5]
$a
[1] 41
$b
[1] 61.5
$c
[1] 56
$d
[1] 51
$e
[1] 49


> tapply(x, y, quantile, probs=c(0,.5,1))[1:3]
$a
  0%  50% 100%
 3.0 31.5 94.0
$b
  0%  50% 100%
31.0 56.5 97.0
$c
  0%  50% 100%
 4.0 59.5 93.0
```

La funzione `split()` è stata congeniata per scomporre un oggetto (si noti quindi che la funzione si applica non solo a vettori, ma anche a matrici o dataframe) in sottogruppi identificati da una vettore di elementi che ne definiscono il partizionamento. L'oggetto restituito dalla funzione `split()` è una lista contenente i singoli sottogruppi.

Nel caso di partizionamento di un vettore:

```
> split(x,y)
$a
 [1] 18 26  3 94 84 29  9 34 55 58
$b
 [1] 40 86  78 60 32 96 31 97 42 53
$c
 [1] 77 43  47 57 70 62  4 69 38 93
```

```
$d
 [1] 85 67  37 72 17 16 41 74 80 21
$e
 [1] 20 79  73 71 68 14  6 45 51 63
$f
 [1] 75 50   8 30 81 15 87 76 22 92
$g
 [1] 91 23 100 10 35  7  5 12 95 11
$h
 [1] 66 90  64 25 44 89 83 19 88  2
$i
 [1] 82  1  99 52 36 27 33 56 39 54
$j
 [1] 28 61  65 59 46 98 49 13 24 48
```

Per cui:

```
> sapply(split(x,y),mean)
   a    b    c    d    e    f    g    h    i    j
41.0 61.5 56.0 51.0 49.0 53.6 38.9 57.0 47.9 49.1
```

produce lo stesso risultato di `tapply(x,y,mean)`.

Per illustrare il partizionamento di un dataframe si ricorre all'ausilio del dataset d'esempio `iris`, descritto alla fine del capitolo 11:

```
> data(iris)
> s=split(iris, iris$Species)
> str(s)
```

```
List of 3
 $ setosa    :'data.frame':     50 obs. of  5 variables:
  ..$ Sepal.Length: num [1:50] 5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
  ..$ Sepal.Width : num [1:50] 3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
  ..$ Petal.Length: num [1:50] 1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
  ..$ Petal.Width : num [1:50] 0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
  ..$ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
 $ versicolor:'data.frame':     50 obs. of  5 variables:
  ..$ Sepal.Length: num [1:50] 7 6.4 6.9 5.5 6.5 5.7 6.3 4.9 6.6 5.2 ...
  ..$ Sepal.Width : num [1:50] 3.2 3.2 3.1 2.3 2.8 2.8 3.3 2.4 2.9 2.7 ...
  ..$ Petal.Length: num [1:50] 4.7 4.5 4.9 4 4.6 4.5 4.7 3.3 4.6 3.9 ...
  ..$ Petal.Width : num [1:50] 1.4 1.5 1.5 1.3 1.5 1.3 1.6 1 1.3 1.4 ...
  ..$ Species     : Factor w/ 3 levels "setosa","versicolor",..: 2 2 2 2 2 2 2 2 2 2 ...
 $ virginica :'data.frame':     50 obs. of  5 variables:
  ..$ Sepal.Length: num [1:50] 6.3 5.8 7.1 6.3 6.5 7.6 4.9 7.3 6.7 7.2 ...
  ..$ Sepal.Width : num [1:50] 3.3 2.7 3 2.9 3 3 2.5 2.9 2.5 3.6 ...
  ..$ Petal.Length: num [1:50] 6 5.1 5.9 5.6 5.8 6.6 4.5 6.3 5.8 6.1 ...
  ..$ Petal.Width : num [1:50] 2.5 1.9 2.1 1.8 2.2 2.1 1.7 1.8 1.8 2.5 ...
  ..$ Species     : Factor w/ 3 levels "setosa","versicolor",..: 3 3 3 3 3 3 3 3 3 3 ...
```

```
> lapply(s, function(x) colMeans(x[,names(iris)[1:4]]))

$setosa
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
       5.006        3.428        1.462        0.246
$versicolor
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
       5.936        2.770        4.260        1.326
$virginica
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
       6.588        2.974        5.552        2.026
```

Infine, la funzione `mapply()` è stata concepita quale versione multivariata della funzione `sapply()`. `mapply()` applica ad una generica funzione, in maniera iterativa, tutti i possibili valori specificati per i suoi argomenti. L'esempio seguente ne chiarisce il funzionamento meglio di quanto non riescano a fare le varie definizioni (quella proposta in questo testo ha peraltro cercato di migliorare quella rilasciata dalla pagina di help di **R** cui si accede digitando `?mapply` al prompt di comando). Richiamiamo, a tale scopo, la funzione `mia.parabola2()` costruita nel capitolo 6:

```
> mia.parabola2
function(x,a,b,c){
y=a*x^2+b*x+c
return(y)
}

> mapply(mia.parabola2, x=0:5, a=1:2, b=1:2, c=5)
[1]  5  9 11 29 25 65
```

equivalente a scrivere:

```
> c(mia.parabola2(0,1,1,5),
mia.parabola2(1,2,2,5),
mia.parabola2(2,1,1,5),
mia.parabola2(3,2,2,5),
mia.parabola2(4,1,1,5),
mia.parabola2(5,2,2,5))
[1]  5  9 11 29 25 65
```

Altro esempio:

```
> set.seed(14)
> mapply(function(x,y) sample(x,y), 3:10, 1:4)
[[1]]
[1] 1
```

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

[[3]]
[1] 3 4 2

[[4]]
[1] 6 3 2 4

[[5]]
[1] 7

[[6]]
[1] 2 4

[[7]]
[1] 8 7 6

[[8]]
[1] 4 7 9 5
```

equivalente a scrivere:

```
> set.seed(14)

> list(sample(3,1), sample(4,2),
sample(5,3), sample(6,4),
sample(7,1), sample(8,2),
sample(9,3), sample(10,4))

 [[1]]
[1] 1

[[2]]
[1] 3 4

[[3]]
[1] 3 4 2

[[4]]
[1] 6 3 2 4

[[5]]
[1] 7

[[6]]
[1] 2 4

[[7]]
[1] 8 7 6

[[8]]
[1] 4 7 9 5
```

La funzione `set.seed()`, che compare nelle righe di codice precedenti e che verrà meglio approfondita nel capitolo 17 dedicato alla "Generazione dei Numeri Pseudo-Casuali", serve per impostare il "seme" di generazione dei numeri casuali. In altre parole, il valore del suo parametro definisce una certa sequenza di generazione casuale: impostando tale valore è possibile garantirsi la replicabilità dell'estrazione casuale, come avviene nell'esempio precedente quando si è voluto confrontare l'applicazione di `mapply()` con il codice tradizionale necessario per ottenere lo stesso risultato.

Capitolo XV

Iterazioni *VS* programmazione ricorsiva

Si è detto che **R** è un pacchetto statistico concepito per lavorare a livello vettoriale; la sua architettura pertanto sconsiglia di utilizzare istruzioni iterative se queste possono essere efficientemente sostituite da istruzioni che sfruttino le proprietà dell'algebra dei vettori e delle matrici.

Quando non è possibile sviluppare una funzione sfruttando i principi della logica vettoriale, l'utilizzo di istruzioni iterative pare inevitabile. Talvolta, però alcuni algoritmi che sembrano necessariamente richiedere la presenza di istruzioni iterative, possono essere conveniente ridisegnati sfruttando la logica della programmazione ricorsiva.

In altre parole, ricorsione e iterazione rappresentano due scelte alternative per risolvere problemi che richiedano l'esecuzione ripetuta di certe operazioni. I termini di confronto sono:

- la semplicità di codifica;
- l'efficienza di esecuzione.

Il confronto va fatto caso per caso ma, in linea di massima:

- la ricorsione privilegia la semplicità di codifica;
- l'iterazione privilegia l'efficienza di esecuzione.

L'uso dell'iterazione è da preferire quando la soluzione iterativa e ricorsiva sono paragonabili dal punto di vista della complessità, oppure se l'occupazione di memoria generata dalla ricorsione viene evitata tramite la soluzione iterativa. Viceversa, l'uso della ricorsione è da preferire quando la complessità della soluzione iterativa è decisamente superiore a quella della soluzione ricorsiva, oppure se l'occupazione di memoria è necessaria alla soluzione del

B. Bertaccini, *Introduzione alla Statistica Computazionale con R*, ISBN (print) 978-88-6453-673-6, ISBN (online) 978-88-6453-674-3, CC BY 4.0, 2018 Firenze University Press.

problema (ad ogni passo si deve tener traccia dello stato) e si verificherebbe anche nella soluzione iterativa (Burattini ed altri, 2016).

Se si pensa di poter implementare una funzione secondo logica ricorsiva, l'importante è "usare la testa e non le gambe": per verificare la correttezza di una procedura ricorsiva non conviene cercare di ricostruire laboriosamente e macchinosamente la sua esecuzione da parte dell'elaboratore; bisogna invece verificare la correttezza logica della condizione di terminazione e del passo di ricorsione.

Esercizio: sfruttando prima la logica iterativa e poi la logica ricorsiva, costruire una funzione in grado di calcolare il fattoriale di un numero.

```
fattoriale.iter = function(nn){
res=1
if (nn>0)
      for(i in 1:nn)
        res=res*i
return(res)
}

fattoriale.ric = function(nn){
res=1
if (nn>0) res=nn*fattoriale.ric(nn-1)

return(res)
}
```

Le due soluzioni proposte producono lo stesso risultato: nel secondo caso però non vengono utilizzate istruzioni iterative e si perviene al risultato voluto mediante ricorsione, ovvero per chiamate successive della stessa funzione.

Esercizio: implementare l'algoritmo di ricerca binaria presentato nel capitolo 13, secondo la logica ricorsiva.

```
ricbin.rec=function(v,a,pos=0){

if (!is.vector(v)) stop("first input parameter must be a vector")
if ( sum(v==sort(v)) < length(v) ) stop("vector must be ordered")

meta=ceiling(length(v)/2)
if (v[meta]==a) print(paste("trovato in posizione:", pos+meta))
  else if (meta==1) print("non trovato")
        else if (v[meta]>a) ricbin.rec(v[1:(meta-1)],a,pos)
              else{
                  pos=pos+meta
                  ricbin.rec(v[(meta+1):length(v)],a,pos)
                  }
}
```

Esercizio: sfruttando la logica ricorsiva, costruire una funzione che produca tutti i possibili anagrammi (permutazioni) di un vettore di caratteri.

In PSEUDOcodice:

```
stampaAnagrammi( parolaDiPartenza, anagramma ){
    IF (parolaDiPartenza ha una sola cifra) THEN
        stampo anagaramma + parolaDiPartenza;
    ELSE{
    FOR (tutti i caratteri di parolaDiPartenza){
        nuovaParolaDiPartenza = parolaDiPartenza - carattereAttuale
        nuovoAnagramma = anagramma + carattereAttuale
        stampaAnagrammi(nuovaParolaDiPartenza, nuovoAnagramma)
        }
    }
}
```

In **R**:

```
anagramma = function(parola, anagr=NULL){
if (!is.vector(parola)) stop("richiesto un vettore come parametro ")
if (is.null(anagr)) perm<<-NULL

lp=length(parola)
if (lp==1){
  perm<<-rbind(perm, c(anagr,parola))
  }
  else
    for (i in 1:lp){
        nuova.parola=parola[-i]
        new.anagr=c(anagr,parola[i])
        anagramma(nuova.parola, new.anagr)
        }
}

> v=c("c","i","a","o")
> index=1:length(v)
> anagramma(index)
> anagrammi=v[perm]
> dim(anagrammi)=dim(perm)
```

```
> anagrammi
      [,1] [,2] [,3] [,4]
 [1,] "c"  "i"  "a"  "o"
 [2,] "c"  "i"  "o"  "a"
 [3,] "c"  "a"  "i"  "o"
 [4,] "c"  "a"  "o"  "i"
 [5,] "c"  "o"  "i"  "a"
 [6,] "c"  "o"  "a"  "i"
 [7,] "i"  "c"  "a"  "o"
 [8,] "i"  "c"  "o"  "a"
 [9,] "i"  "a"  "c"  "o"
[10,] "i"  "a"  "o"  "c"
[11,] "i"  "o"  "c"  "a"
[12,] "i"  "o"  "a"  "c"
[13,] "a"  "c"  "i"  "o"
[14,] "a"  "c"  "o"  "i"
[15,] "a"  "i"  "c"  "o"
[16,] "a"  "i"  "o"  "c"
[17,] "a"  "o"  "c"  "i"
[18,] "a"  "o"  "i"  "c"
[19,] "o"  "c"  "i"  "a"
[20,] "o"  "c"  "a"  "i"
[21,] "o"  "i"  "c"  "a"
[22,] "o"  "i"  "a"  "c"
[23,] "o"  "a"  "c"  "i"
[24,] "o"  "a"  "i"  "c"
```

L'utilizzo dello speciale operatore di assegnazione `<<-` indica ad **R** che la variabile `perm` deve essere trattata come un oggetto di tipo "globale" e non come un oggetto esclusivamente riferito all'ambiente locale di esecuzione della funzione `anagramma()`. Infatti, come in ogni altro linguaggio di programmazione, tutti gli oggetti creati all'interno di una funzione cessano di esistere alla fine dell'esecuzione della stessa, a meno che non si specifichi la volontà di estenderne la visibilità agli ambienti superiori. Nel caso in esempio, `perm` è una matrice che non deve perdere le righe che le vengono man mano aggiunte nelle chiamate ricorsive della funzione.

Capitolo XVI

Principali funzionalità per la descrizione statistica delle informazioni

R dispone di un'ampia libreria di funzioni statistiche per sintetizzare i dati. Riteniamo opportuno iniziare tale rassegna, illustrando le funzioni atte alla produzione di tabelle di frequenza.

La funzione `unique()` restituisce i distinti valori (modalità) assunti dalla variabile argomento della funzione; in riferimento al dataframe `DF3` la cui procedura di importazione è stata presentata nel capitolo 11:

```
> unique(DF3$sesso)
[1] M F
Levels: F M

> unique(DF3$altezza)
 [1] 182 172 170 160 184 162 183 167 171 178

> unique(DF3$peso)
[1] 80 70 55 85 57 82 63 61
```

In alternativa, la funzione `duplicate()` restituisce un vettore di valori logici, dove i valori `TRUE` indicano le posizioni dei valori che sono già stati rilevati in precedenza.

```
> duplicated(DF3$altezza)
 [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE

> d=duplicated(DF3$peso)

> DF3$peso[d]
[1] 70 82
```

B. Bertaccini, *Introduzione alla Statistica Computazionale con R*, ISBN (print) 978-88-6453-673-6, ISBN (online) 978-88-6453-674-3, CC BY 4.0, 2018 Firenze University Press.

L'esempio precedente chiarisce l'assenza di valori replicati per la variabile `altezza` e di due valori ripetuti per la variabile `peso`, come confermato dalla funzione `table()` che calcola le frequenze assolute per tutte le distinte modalità ordinate della variabile parametro:

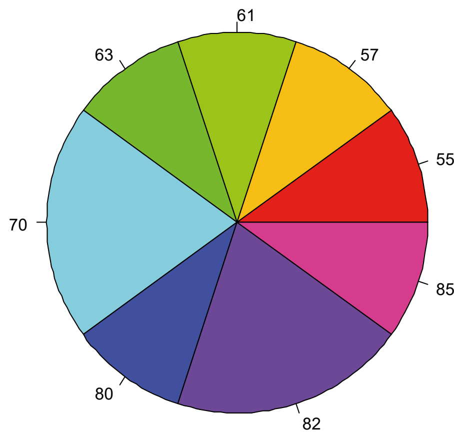
```
> t=table(DF3$peso)
> t

55 57 61 63 70 80 82 85
 1  1  1  1  2  1  2  1
```

Notare che, all'apparenza, il risultato della funzione sembra una matrice, ma è in realtà un vettore alle cui componenti sono state assegnate delle etichette:

```
> length(t)
[1] 8

> names(t)
[1] "55" "57" "61" "63" "70" "80" "82" "85"
```

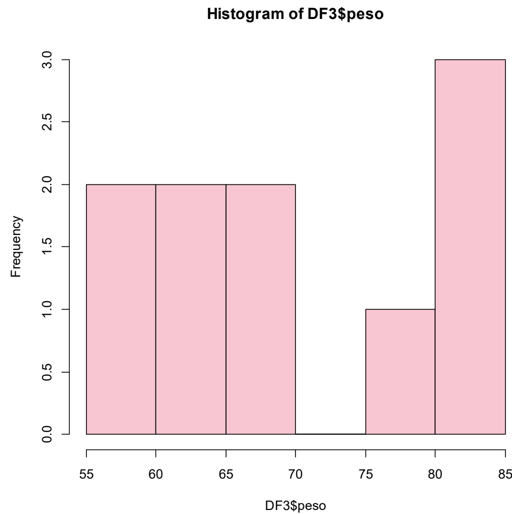
L'oggetto restituito dalla funzione `table()` può essere graficamente visualizzato tramite un grafico a torta:

```
> pie(t, col = rainbow(length(t)))
```



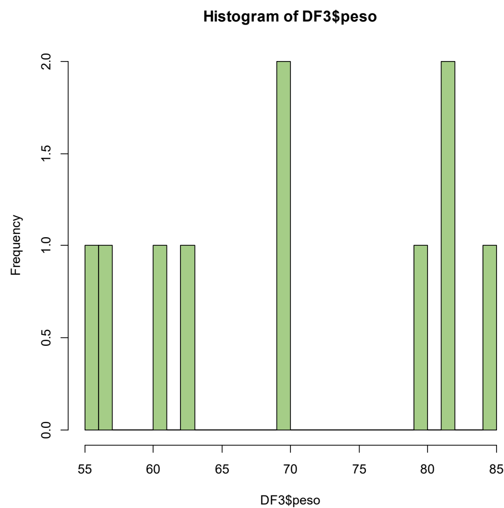
Sempre per visualizzare graficamente la distribuzione di frequenza di una certa variabile, si può in alternativa ricorrere alla funzione `hist()`, deputata alla rappresentazione grafica dell'istogramma delle componenti di un generico vettore:

```
> hist(DF3$peso,col="pink")
```



La funzione `hist()`, di default, calcola il numero delle classi e la loro ampiezza secondo la formula di Sturges (Sturgers, 1926) che è funzione del campo di variazione dei dati. È sempre possibile definire un numero di classi differente impostando il parametro `breaks=`:

```
> hist(DF3$peso,breaks=30, col="light green")
```



Per quanto riguarda le possibilità di riduzione delle informazioni, le librerie base di **R** includono una vasta serie di funzioni per il calcolo di indici di posizione e variabilità.

Nell'ambito degli indici di posizione è necessario citare le funzioni `mean()` e `median()`, da utilizzare rispettivamente per il calcolo della media aritmetica e della mediana di un insieme di informazioni (indipendentemente dal tipo di oggetto scelto per la loro memorizzazione):

```
> v=c(1,3,4,2,7,6,5,8,9,10)
> mean(v)
[1] 5.5
> median(v)
[1] 5.5
> m.1=matrix(v,nrow=2)
> m.1
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    4    7    5    9
[2,]    3    2    6    8   10
> mean(m.1)
[1] 5.5
> median(m.1)
[1] 5.5
```

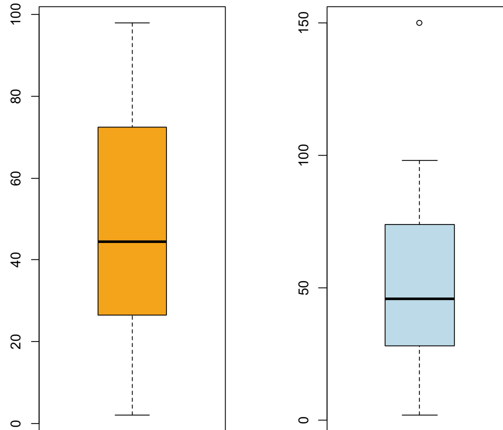
In maniera del tutto analoga agisce la funzione `quantile()`, deputata all'estrazione dei generici quantili da un vettore o da una matrice di dati. La funzione richiede che vengano specificati i valori dei quantili da calcolare ricorrendo al parametro `probs`, con valori di default `probs = seq(0, 1, 0.25)` finalizzati al calcolo del minimo, del primo quartile, della mediana, del terzo quartile e del massimo:

```
> s=sample(1:100,20)
> quantile(s)
   0%   25%   50%   75%  100%
 2.00 27.25 44.50 71.75 98.00
> quantile(s, probs=c(.10,.90))
 10%  90%
15.4 90.4
```

Mediana e quartili di una distribuzione di valori possono essere convenientemente rappresentati in forma grafica ricorrendo alla funzione `boxplot()`, capace anche di segnalare la presenza di eventuali osservazioni anomale in base alle seguenti regole grafiche:

a) il rettangolo indica il grado di dispersione del 50% dei valori centrali della distribuzione (ovvero dei valori compresi tra il primo e terzo quartile);
b) i segmenti che partono dai lati minori del rettangolo indicano l'estensione delle code di sinistra e di destra della distribuzione. Tali segmenti hanno un punto di troncamento a destra identificato dal più grande dei valori inferiori a $Q_{0.75}+1.5DI$, a sinistra identificato dal più piccolo dei valori superiori a $Q_{0.25}-1.5DI$ (dove $DI=Q_{0.75}-Q_{0.25}$);
c) eventuali punti esterni ai punti di troncamento sono potenziali valori anomali e devono essere rappresentati come singoli punti.

```
> par(mfrow=c(1,2))
> boxplot(s, col="orange")
> boxplot(c(s,150), col="light blue")
```



Nel capitolo finale, dedicato alla “Selezione di esercizi svolti”, si affronterà il problema di come implementare autonomamente la funzione `boxplot()`.

Le funzioni `mean()`, `median()` e `quantile()` si applicano indistintamente a vettori o matrici. In particolare, come visto in precedenza, nel caso di matrici o dataframe, l’impiego combinato di tali funzioni con la funzione `apply()` consente di computare l’indice (o gli indici) di posizione desiderato/i anche lungo una particolare dimensione dell’oggetto considerato:

```
> apply(m.1,2,mean)
[1] 2.0 3.0 6.5 6.5 9.5

> apply(m.1,1,median)
[1]  5  6

> x=matrix(sample(1:200,20),ncol=5)
> x
     [,1] [,2] [,3] [,4] [,5]
[1,]   15  189  111  106  158
[2,]  146    8   88   37  174
[3,]   32   83   61  115   57
[4,]  143  165   69   99  117

> apply(x,1,quantile,probs=c(0,.10,.25,.75,.90,1))
      [,1]  [,2]  [,3]  [,4]
0%    15.0   8.0  32.0  69.0
10%   51.4  19.6  42.0  81.0
25%  106.0  37.0  57.0  99.0
75%  158.0 146.0  83.0 143.0
90%  176.6 162.8 102.2 156.2
100% 189.0 174.0 115.0 165.0
```

Per quanto riguarda il calcolo degli usuali indici di variabilità è necessario citare le funzioni `var()` e `sd()`, da utilizzare rispettivamente per il calcolo della varianza e della deviazione standard (sebbene la seconda sia semplicemente la radice quadrata della prima). Notare che la formula di calcolo della varianza utilizzata da **R** è

$$s^2 = \frac{\sum_{i=1}^{n} \left(x_i - \overline{x} \right)^2}{n-1}$$

che produce uno stimatore non distorto della varianza per variabili casuali I.I.D. (indipendenti, identicamente distribuite):

```
> var(v)
[1] 9.166667
> sum((v-mean(v))^2)/(length(v)-1)
[1] 9.166667
> sd(v)
[1] 3.027650
> sqrt(var(v))
[1] 3.027650
```

Ovviamente la presenza di valori mancanti nelle serie di dati alle quali le funzioni per il calcolo degli indici di posizione e di variabilità illustrate in precedenza devono essere applicate costituisce un naturale impedimento nel calcolo dell'indice richiesto. Ad esempio, se si aggiunge un valore mancante al vettore `v` la media aritmetica non è più calcolabile, a meno che non venga settato a `TRUE` il parametro logico `na.rm=T`; in tal caso, la funzione provvede al computo dell'indice richiesto per i soli valori diversi da NA.

```
> w=c(v,NA)
> w
 [1]  1  3  4  2  7  6  5  8  9 10 NA


> mean(w)
[1] NA
> mean(w,na.rm=T)
[1] 5.5
> var(w)
[1] NA
> var(w,na.rm=T)
[1] 9.166667
```

Concludiamo la rassegna degli indici di variabilità citando le funzioni `cov()` e `cor()` per il calcolo rispettivamente della covarianza e della correlazione tra serie di dati multiple e di eguale lunghezza:

```
> cor(1:10,2:10)
Error in cor(1:10, 2:10) : incompatible dimensions
> cor(1:10,2:11)
[1] 1
```

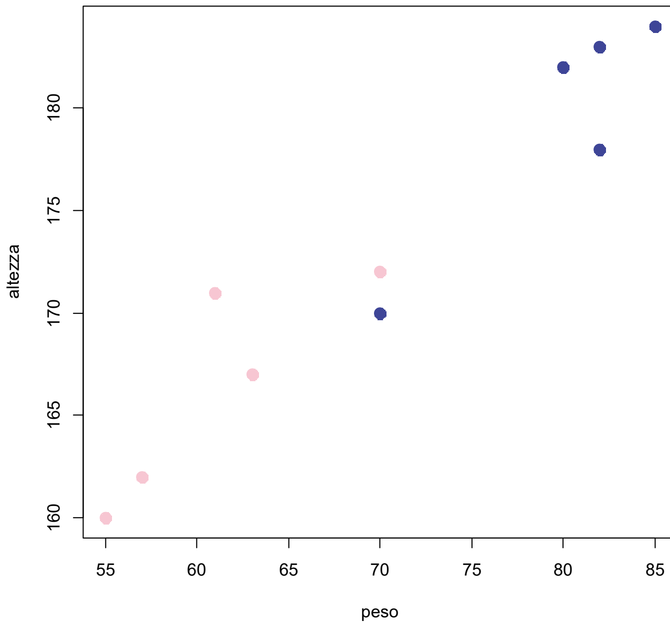
Riconsiderando l'oggetto `DF3`, per come è stato modificato nel capitolo 12:

```
> DF3
   sesso peso altezza ratio classi.ratio
1      M   80     182  2.28        basso
2      F   70     172  2.46        medio
3      M   70     170  2.43        medio
4      F   55     160  2.91         alto
5      M   85     184  2.16        basso
6      F   57     162  2.84         alto
7      M   82     183  2.23        basso
8      F   63     167  2.65        medio
9      F   61     171  2.80         alto
10     M   82     178  2.17        basso


> cor(DF3[,2:4])
              peso    altezza      ratio
peso     1.0000000  0.9582371 -0.9881213
altezza  0.9582371  1.0000000 -0.9132343
ratio   -0.9881213 -0.9132343  1.0000000
```

Le variabili `peso` e `altezza` si dimostrano altamente correlate. Nel caso di due variabili, l'analisi grafica della distribuzione congiunta dei valori osservati può certamente essere un valido aiuto nella comprensione e/o documentazione del fenomeno. La funzione generalizzata `plot()` di **R**, nello specifico, produce il grafico a dispersione (*scatter-plot*) delle coppie di valori generate dall'abbinamento delle variabili `peso` e `altezza`. Indicizzando i colori in base alla variabile sesso, si ottiene[17]:

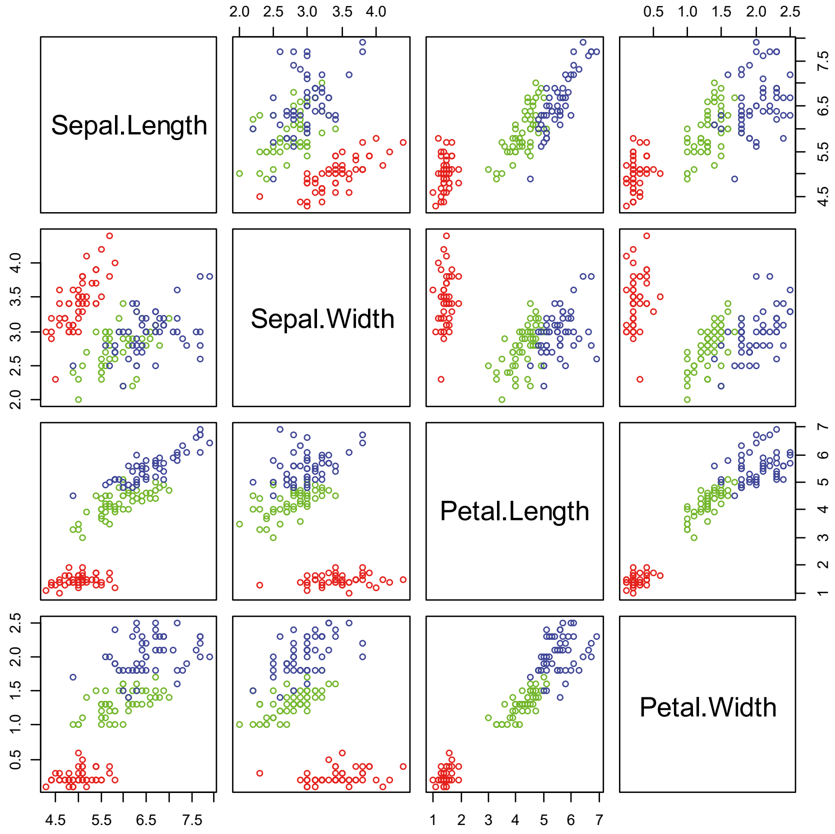
```
> plot(DF3[,2:3], col=c("blue","pink")[DF3$sesso], pch=19, cex=1.5)
```

[17] I parametri grafici `pch=` e `cex=` sono utilizzati rispettivamente per definire il tipo di tratteggio grafico identificante la singola coppia di osservazioni (`pch=21` corrisponde al pallino pieno) e la dimensione di tale tratteggio.



In generale, nel caso di più di due variabili, l'analisi grafica della distribuzione congiunta può risultare complicata se non impossibile. La funzione `plot()` di **R**, qualora le variabili da analizzare congiuntamente siano più di due, produce la cosiddetta matrice dei grafici a dispersione (*scatter-plot matrix*), composta dagli *scatter-plot* relativi a tutte le possibili coppie di variabili. Il codice successivo illustra la *scatter-plot matrix* relativa alle 4 variabili quantitative presenti nel dataset d'esempio `iris` (descritto nel capitolo 11):

```
> data(iris)
> head(iris)
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa


> plot(iris[,1:4],col=c("red","green","blue")[iris$Species])
```


Sepal.Length
Sepal.Width
Petal.Length
Petal.Width
2.0 2.5 3.0 3.5 4.0
0.5 1.0 1.5 2.0 2.5
4.5 5.5 6.5 7.5
1 2 3 4 5 6 7

Capitolo XVII

Generazione di numeri (pseudo) casuali

In statistica (computazionale) si ha sovente la necessità di calcolare per via numerica il valore atteso di una variabile casuale, magari legata alla conduzione di un esperimento casuale complesso, oppure di derivare quantili e forma di una certa distribuzione empirica al fine di valutarne la corrispondenza con un modello matematico noto. Per realizzare questi obiettivi occorre prendere dimestichezza con i cosiddetti Metodi di simulazione Monte Carlo. Questi metodi, approfonditi nel capitolo 20, richiedono la conoscenza delle funzioni messe a disposizione da **R** per la generazione randomizzata di osservazioni dalle principali distribuzioni di variabili casuali. Tali funzioni si basano tutte su algoritmi più o meno complessi che producono, in realtà, sequenze di numeri pseudo-casuali (**PRNG-A**, *Pseudo-Random Numbers Generator Algorithms*) nell'intervallo [0, 1][18]. Una sequenza di numeri pseudo-casuali è una sequenza generata da un algoritmo deterministico che ha, approssimativamente, le stesse proprietà statistiche di una sequenza generata da un vero processo casuale.

---

[18] Gli algoritmi proposti in letteratura differiscono essenzialmente per l'efficienza computazionale e la lunghezza (periodo) della sequenza generata. Per approfondimenti:

- Knuth, D. E. (2002). *The Art of Computer Programming*. Volume 2, (3rd edition, ninth printing). Addison-Wesley, Reading (Ma);
- Marsaglia, G. (1997). *A random number generator for C*. Discussion paper, posting on Usenet newsgroup sci.stat.math on September 29, 1997;
- Marsaglia, G. and Zaman, A. (1994). *Some portable very-long-period random number generators.* Computers in Physics, 8, 117–121;
- Reeds, J., Hubert, S. and Abrahams, M. (1982–4). *C implementation of SuperDuper*. University of California at Berkeley. (Personal communication from Jim Reeds to Ross Ihaka);
- Wichmann, B. A. and Hill, I. D. (1982). *Algorithm AS 183: An Efficient and Portable Pseudo-random Number Generator*, Applied Statistics, 31, 188–190; Remarks: 34, 198 and 35, 89.

B. Bertaccini, *Introduzione alla Statistica Computazionale con R*, ISBN (print) 978-88-6453-673-6, ISBN (online) 978-88-6453-674-3, CC BY 4.0, 2018 Firenze University Press.

Il primo PRNG-A presentato in questo capitolo è una delle più semplici tecniche di generazione di numeri pseudo-casuali nell'intervallo [0, 1]:

1) si scelgono due interi m e a con $0 < a \ll m$ (generalmente si pone $a \simeq \sqrt{m}$);
2) si sceglie il cosiddetto "seme di generazione", ovvero un intero $x_0 \in [1, m)$;
3) si calcolano i semi successivi: $x_{i+1} = (a \cdot x_i) \operatorname{mod} m$, con $i = 0,1,2,...$;
4) la sequenza di numeri pseudo-casuali tra [0, 1] è data da: $u_{i+1} = x_{i+1} / m$.

```
prng.a= function(n, seed, m, a){
if(a>=m) stop("a must less than m")
if(seed>m) stop("seed must be less than or equal m")
seq=NULL
x=seed
for (i in 1:n){
      x[i+1]=(a*x[i])%%m
      seq=c(seq, x[i+1]/m)
      }
return(seq)
}
```

La sequenza restituita è certamente predeterminata, ma i numeri appariranno generati casualmente ovvero non prevedibili né riproducibili da chi non conosce l'algoritmo e i parametri di partenza utilizzati:

```
> prng.a(14,4,12,sqrt(12))
[1] 0.1547005 0.5358984 0.8564065 0.9666790 0.3486743 0.2078432 0.7199899
[8] 0.4941183 0.7116761 0.4653182 0.6119095 0.1197167 0.4147107 0.4366001
```

Differenti valori di m e a generano infatti sequenze diverse e di differente lunghezza. Occorre prestare particolare attenzione alla scelta di m, perché spesso le sequenze tendono a degenerare producendo una serie di zeri:

```
> prng.a(6,3,12,sqrt(12))
[1] 0.8660254 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
```

In questo caso, infatti:

$$\begin{cases} x_1 = \left(\sqrt{12} \cdot 3\right) \operatorname{mod} 12 = \sqrt{12} \cdot 3 \\ u_1 = \sqrt{12} \cdot 3 / 12 = 3 / \sqrt{12} = 0.8660254 \end{cases}$$

$$\begin{cases} x_2 = \left(\sqrt{12} \cdot \left(\sqrt{12} \cdot 3\right)\right) \operatorname{mod} 12 = 0 \\ u_2 = 0/12 = 0 \end{cases}$$

e così via per tutti i valori generati successivamente.

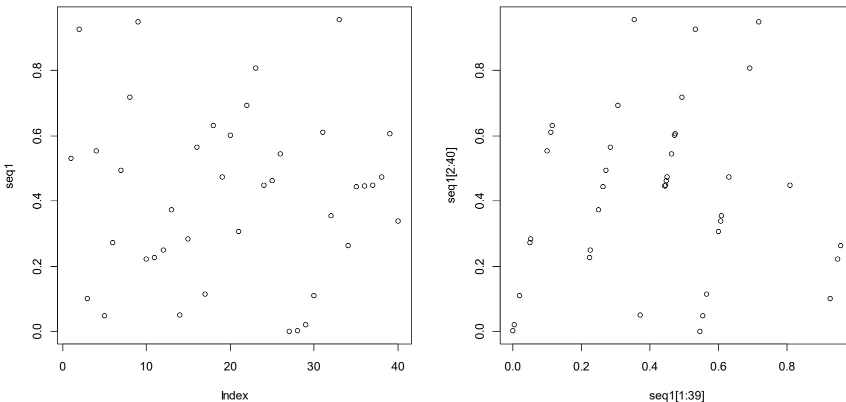
Per ovviare a questo problema si suggerisce di utilizzare valori di m non divisibili per a (numeri primi per m sono da preferirsi):

```
> seq1=prng.a(40,3,31,5.5)
> seq1
```

```
 [1] 0.5322580645 0.9274193548 0.1008064516 0.5544354839 0.0493951613
 [6] 0.2716733871 0.4942036290 0.7181199597 0.9496597782 0.2231287802
[11] 0.2272082913 0.2496456023 0.3730508128 0.0517794701 0.2847870857
[16] 0.5663289716 0.1148093439 0.6314513914 0.4729826527 0.6014045900
[21] 0.3077252450 0.6924888472 0.8086886598 0.4477876290 0.4628319596
[26] 0.5455757778 0.0006667782 0.0036672800 0.0201700398 0.1109352187
[31] 0.6101437030 0.3557903664 0.9568470149 0.2626585821 0.4446222017
[36] 0.4454221091 0.4498216002 0.4740188009 0.6071034049 0.3390687268
```

Il primo pannello della figura successiva illustra la disposizione dei punti generati. Si nota la totale dispersione casuale della sequenza.

```
> par(mfrow=c(1,2))
> plot(seq1)
> plot(seq1[1:39], seq1[2:40])
```



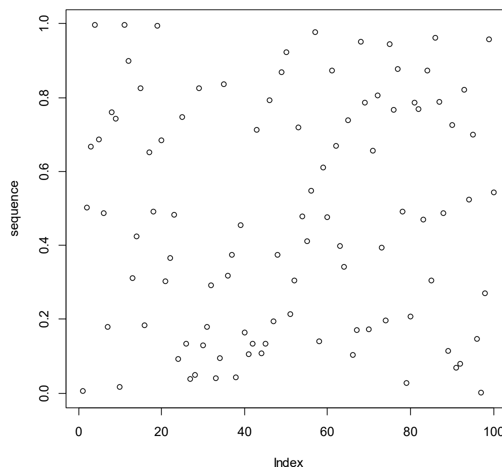
Il secondo pannello mostra però uno dei difetti dell'algoritmo precedente, ovvero la correlazione tra valori successivi della sequenza, che rende questo metodo di generazione non indicato per simulazioni Monte Carlo di esperimenti che presuppongono indipendenza dei valori osservati.

Tale algoritmo (detto Generatore Moltiplicativo di Lehemer o di Parker-Miller) è una configurazione particolare del più generale Generatore Lineare Congruenziale (***Linear Congruential Generator*** - **LCG**), definito ricorsivamente dalla formula:

$$x_{i+1} = \left(a \cdot x_i + b\right) \bmod m, \quad \text{con } 0 \le b < m.$$

Eccettuato il problema della correlazione tra valori successivi della sequenza, gli LCG sono generalmente in grado di produrre buone sequenze di numeri pseudocasuali; a conferma di quanto detto in precedenza per il caso particolare con $b = 0$, la qualità della sequenza è comunque funzione della scelta operata per i coefficienti m, a e b. Gli algoritmi LCG più efficienti hanno m uguale a una potenza del 2, spesso $m = 2^{32}$ o $m = 2^{64}$, perché questo consente di calcolare l'operazione modulo troncando i 32 o i 64 bit meno significativi. Ad esempio, la funzione *rand()* del Borland C e C++ utilizza $m = 2^{32},\ a = 22695477,\ b = 1$:

```
LCG.a = function(n, seed, m, a, b=0){
if(a>=m) stop("a must less than m")
if(b>=m) stop("b must less than m")
if(seed>m) stop("seed must be less than or equal m")

seq=NULL
x=seed
for (i in 1:n){
      x[i+1]=(a*x[i]+b)%%m
      seq=c(seq, x[i+1]/m)
      }
return(seq)
}
> LCG.a(n=40,seed=1,m=2^32,a=22695477,b=1)

 [1] 0.005284203 0.501993488 0.667579932 0.995506888 0.685743749 0.486539066
 [7] 0.178324640 0.761050880 0.741855085 0.016592205 0.997215212 0.899468970
[13] 0.310719904 0.424653988 0.825137105 0.184722077 0.652248617 0.491343539
[19] 0.993197706 0.683822345 0.302774485 0.365058046 0.482897367 0.093562376
[25] 0.746992942 0.133093040 0.038086798 0.048967194 0.826201984 0.129610591
[31] 0.178752325 0.291519277 0.041620456 0.094666653 0.837073718 0.318659104
[37] 0.374905892 0.042615868 0.455460318 0.163994839

> plot(LCG.a(n=100,seed=1,m=2^32,a=22695477,b=1),ylab="sequence")
```



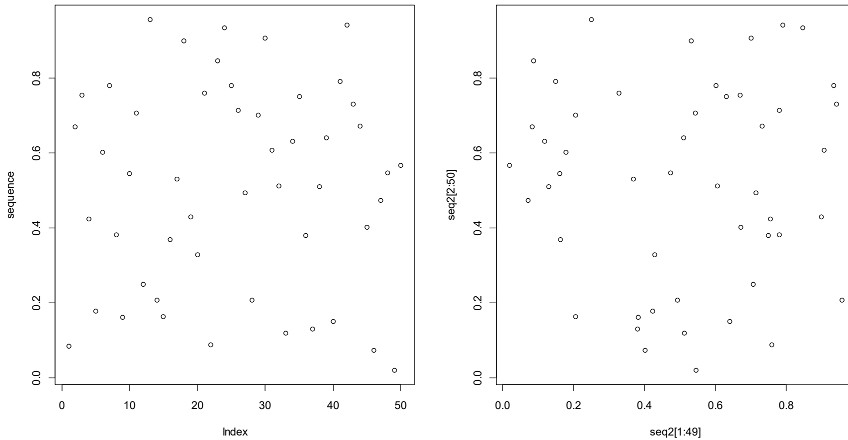
Per risolvere le criticità degli algoritmi LCG è stato proposto l'impiego della successione di Fibonacci $S_n = S_{n-1} + S_{n-2}$. L'algoritmo di generazione dei numeri pseudocasuali di Fibonacci "ritardato" (***Lagged Fibonacci Generator*** - **LFB**) si basa su una generalizzazione di tale successione:

$$x_n = \left(x_{n-j} - x_{n-k}\right) \bmod m, \quad \text{con } 0 < j < k.$$

La versione presentata in questo testo si basa sull'addizione dei termini ritardati di Fibonacci. In letteratura sono stati comunque proposti LFB che contemplano una qualsiasi operazione tra tali termini (quindi anche sottrazione, moltiplicazione o divisione). Anche per questo algoritmo si tende a scegliere un valore di m uguale ad una potenza del 2, (usualmente $m = 2^{32}$ o $m = 2^{64}$)[19].

```
LFG = function(n,j,k,m){

if(j>k) stop("second parameter must be lower than the third")

# generazione della sequenza di Fibonacci di n+k elementi
fib=c(0,1)
for(i in 3:(n+k))
      fib[i]=fib[i-1]+fib[i-2]

# numeri pseudo-casuali generati secondo logica vettoriale
seq= ((fib[(1+k-j):(n+k-j)]-fib[1:n]) %% m) /m
return(seq)
}

> seq2=LFG(50,50,107,2^32)
> seq2
 [1] 0.08453522 0.66966990 0.75420512 0.42387502 0.17808014 0.60195517
 [7] 0.78003531 0.38199048 0.16202579 0.54401627 0.70604206 0.25005833
[13] 0.95610039 0.20615873 0.16225912 0.36841784 0.53067696 0.89909481
[19] 0.42977177 0.32886658 0.75863835 0.08750493 0.84614329 0.93364822
[25] 0.77979150 0.71343972 0.49323123 0.20667095 0.69990218 0.90657313
[31] 0.60647529 0.51304844 0.11952370 0.63257217 0.75209594 0.38466811
[37] 0.13676405 0.52143288 0.65819645 0.17962837 0.83782578 0.01745605
[43] 0.85528564 0.87275696 0.72804260 0.60079956 0.32879639 0.92956543
[49] 0.25842285 0.18798828

> par(mfrow=c(1,2))
> plot(seq2,ylab="sequence")
> plot(seq2[1:49],seq2[2:50])
```

[19] Per una lista delle migliori combinazioni di valori j e k si consulti la pagina 29 di:
- Knuth, D. E. (2002). *The Art of Computer Programming*. Vol. 2, (3rd ed.). Addison-Wesley, Reading (Ma).



A differenza di quanto evidenziava il Generatore Moltiplicativo di Lehemer, adesso i valori successivi della sequenza non sono tra loro correlati.

La funzione `runif()` è il metodo base con cui **R** genera numeri pseudo-casuali. Come vedremo meglio nel capitolo 19 dedicato al "Trattamento delle distribuzioni di variabili casuali", `runif(n)` è deputata all'estrazione di un campione casuale di n osservazioni da una distribuzione Uniforme nell'intervallo [min, max] (con valori di default `min=0` e `max=1`):

```
> runif(10)
 [1] 0.7319681 0.8340068 0.2330951 0.5709822 0.6332733 0.3282053 0.1161438
 [8] 0.1956791 0.6620506 0.3329764

> runif(10,min=5,max=10)
 [1] 8.180326 6.030583 7.374389 9.344640 7.964468 5.240585 8.683944 6.677406
 [9] 9.617148 5.527916
```

Il seme di generazione è impostato internamente sulla base del "clock" dell'elaboratore; tuttavia è possibile specificare un differente seme ricorrendo alla funzione `set.seed()`. Il codice sottostante mostra l'impiego in successione di funzioni (`runif()` e `sample()`) basate su algoritmi di generazione di numeri pseudo-casuali; tuttavia la successiva chiamata di `set.seed()` consente di riprodurre esattamente i valori generati inizialmente da `runif()`:

```
> set.seed(14)
> runif(10,min=0,max=100)
 [1] 25.40337 63.78273 95.71886 55.25467 98.30671 51.14739 93.28234 42.84277
 [9] 48.55851 38.16522

> sample(1000,5)
[1] 892 164 474 849 855

> sample(10,5,rep=T)
[1] 8 4 7 9 6
```

```
> set.seed(14)
> runif(10,min=0,max=100)
 [1] 25.40337 63.78273 95.71886 55.25467 98.30671 51.14739 93.28234 42.84277
 [9] 48.55851 38.16522
```

Come detto in precedenza, la possibilità di impostare il seme di generazione dei numeri pseudo-casuali consente al ricercatore di garantirsi, in ogni momento, la replicabilità di esperimenti casuali basati su schemi simulativi più o meno complessi che richiamano algoritmi simili a quelli descritti in questo capitolo. A tale proposito, la funzione `RNGkind()` può essere convenientemente impiegata per interrogare **R** sul tipo di algoritmo RNG (*Random Number Generator*) in uso o per impostarne uno diverso (tale specifica può essere effettuata anche con la funzione `set.seed()` tramite il parametro `kind`). Di default **R** utilizza l'algoritmo "Mersenne-Twister"[20]. È possibile prendere nota degli altri RNG disponibili digitando `?RNG` al *prompt* di comando.

Esercizio: ricorrendo esclusivamente alla funzione `runif()` implementare una funzione analoga alla funzione `sample()`, ovvero deputata all'estrazione (con e senza reimmissione) di un campione di dimensione k da un generico vettore di n osservazioni.

```
my.sample = function(v,k,repl=TRUE){

n=length(v)
if (!repl & k>n) stop("wrong sample size")

if (repl)
      sample.indices = floor(runif(k,min=0,max=n))+1
  else{
      sample.indices=NULL
      indexes=1:n
      for(i in 1:k){
            s.index = floor(runif(1,min=0,max=length(indexes)))+1
            sample.indices=c(sample.indices,indexes[s.index])
            indexes=indexes[-s.index]
            }
      }

return(v[sample.indices])
}


> my.sample(1:10,5)
[1]  3  9 10  9  4
```

[20] Matsumoto, M. and Nishimura, T. (1998). Mersenne Twister: A 623-dimensionally equidistributed uniform pseudo-random number generator, *ACM Transactions on Modeling and Computer Simulation*, 8, 3–30.

```
> my.sample(LETTERS,5,rep=F)
[1] "S" "A" "P" "W" "V"

> my.sample(LETTERS,26,rep=F)
 [1] "D" "I" "W" "H" "T" "V" "L" "Z" "M" "Y" "X"
[12] "F" "Q" "N" "O" "B" "S" "U" "R" "A" "K" "G"
[23] "P" "J" "C" "E"
```

Capitolo XVIII

Elementi di calcolo combinatorio

"Quasi tutta la matematica classica, dall'algebra elementare alla teoria delle equazioni differenziali, è applicabile al mondo reale solo nell'ipotesi che questo sia costituito di oggetti e di eventi a carattere continuo. Però, in molte situazioni comuni in fisica e in chimica ed in altre scienze, si può parlare realisticamente solo di collezione di oggetti a carattere discreto, i quali agiscono in combinazione, un passo per volta; la matematica applicata a tali situazioni si chiama analisi combinatoria."

G. C. Rota, 1973. *Analisi combinatoria*. UMI – Zanichelli

Il calcolo combinatorio è un po' "*l'arte di saper contare… senza dover contare*". In altre parole è quella branca della matematica che mette a disposizione una serie di regole di calcolo finalizzate al conteggio degli elementi di un insieme o dei casi che si possono verificare a seguito di un esperimento, che consentono di evitare una loro enumerazione esplicita.

Per le finalità di questo testo e i programmi dei corsi di studio cui è rivolto ci si limiterà ad illustrare il modo in cui **R** calcola le permutazioni, le disposizioni e le combinazioni.

Una **permutazione** su un insieme A costituito da *n* elementi (indicizzati {1, 2, . . . , *n* }) è una regola che a ogni elemento di A ne associa un altro (eventualmente l'elemento stesso) in maniera che a due indici diversi non venga mai associato la stesso indice; in altre parole la permutazione è una funzione biiettiva su A: $P_n : A \to A$.

B. Bertaccini, *Introduzione alla Statistica Computazionale con R*, ISBN (print) 978-88-6453-673-6, ISBN (online) 978-88-6453-674-3, CC BY 4.0, 2018 Firenze University Press.

Ad esempio: se $A = \{1,2,3,4,5\}$, allora $\begin{bmatrix} 1,2,3,5,4 \\ 2,3,1,4,5 \end{bmatrix}$ sono due sue permutazioni.

Poiché ci sono *n* modi di scegliere l'oggetto che occupa la prima posizione, e per ciascuno di essi ci sono *n* - 1 modi di scegliere l'oggetto che occupa la seconda posizione, e poi fissate le prime due posizioni ci sono *n* - 2 modi di scegliere l'oggetto nella terza posizione, e così via… è immediato comprendere che il numero delle permutazioni di *n* oggetti è pari al fattoriale di *n*:

$$P_n = n! = n \cdot (n-1) \cdot (n-2) \cdot (n-3) \cdot \ldots \cdot 1$$

La funzione di libreria per il calcolo del fattoriale (`factorial()`) è già stata presentata nel capitolo 5. Nel capitolo 15 è stato inoltre affrontato il problema di riscrivere tale funzione secondo la logica ricorsiva. **R**, con l'ausilio del pacchetto aggiuntivo `combinat`, consente tramite la funzione `permn()` inoltre di visualizzare, sotto forma di lista, tutte le differenti permutazioni di un certo insieme:

```
> library(combinat)
> permn(LETTERS[1:3])


[[1]]
[1] "A" "B" "C"

[[2]]
[1] "A" "C" "B"

[[3]]
[1] "C" "A" "B"

[[4]]
[1] "C" "B" "A"

[[5]]
[1] "B" "C" "A"

[[6]]
[1] "B" "A" "C"
```

Notare come la funzione `permn()` restituisca lo stesso identico risultato della funzione ricorsiva `anagramma()` presentata come esercitazione in chiusura del capitolo 15:

```
> anagramma(LETTERS[1:4])
> perm
      [,1] [,2] [,3] [,4]
 [1,] "A"  "B"  "C"  "D"
 [2,] "A"  "B"  "D"  "C"
 [3,] "A"  "C"  "B"  "D"
 [4,] "A"  "C"  "D"  "B"
 [5,] "A"  "D"  "B"  "C"
```

```
 [6,] "A"  "D"  "C"  "B"
 [7,] "B"  "A"  "C"  "D"
 [8,] "B"  "A"  "D"  "C"
 [9,] "B"  "C"  "A"  "D"
[10,] "B"  "C"  "D"  "A"
[11,] "B"  "D"  "A"  "C"
[12,] "B"  "D"  "C"  "A"
[13,] "C"  "A"  "B"  "D"
[14,] "C"  "A"  "D"  "B"
[15,] "C"  "B"  "A"  "D"
[16,] "C"  "B"  "D"  "A"
[17,] "C"  "D"  "A"  "B"
[18,] "C"  "D"  "B"  "A"
[19,] "D"  "A"  "B"  "C"
[20,] "D"  "A"  "C"  "B"
[21,] "D"  "B"  "A"  "C"
[22,] "D"  "B"  "C"  "A"
[23,] "D"  "C"  "A"  "B"
[24,] "D"  "C"  "B"  "A"
```

Una **disposizione (semplice)** di k di elementi di un insieme A di n elementi, con $k \leq n$, è una permutazione di k elementi di A. Per avere il numero di tutte queste permutazioni ridotte si considera che il primo componente di una tale sequenza può essere scelto in n modi diversi, il secondo in n - 1 modi e così via, sino al k-esimo che può essere scelto in n - k +1 modi diversi. Pertanto il numero di disposizioni semplici di k oggetti estratti da un insieme di n oggetti è dato da:

$$D_{n,k} = n \cdot (n-1) \cdot \ldots \cdot (n-k+1) = \frac{n!}{(n-k)!}$$

Una **combinazione (semplice)** è una sequenza di elementi di un insieme distinti per la natura (ovvero nella quale non ha importanza l'ordine dei componenti e non si può ripetere lo stesso elemento più volte). La collezione delle combinazioni di k elementi estratti da un insieme A di n oggetti distinti si ottiene dalla collezione delle disposizioni semplici di lunghezza k degli elementi di A ripartendo però tali sequenze nelle *classi* delle sequenze che presentano lo stesso sottoinsieme di A e scegliendo una sola sequenza da ciascuna di queste classi. Infatti, ciascuna delle suddette classi di lunghezza k contiene k! diverse sequenze, in quanto accanto a una sequenza s si hanno tutte e sole quelle ottenibili permutando i componenti di s. Quindi il numero di combinazioni semplici di k oggetti estratti da un insieme di n oggetti è dato da:

$$C_{n,k} = \frac{n!}{k!(n-k)!}$$

La funzione di libreria per il calcolo del coefficiente combinatorio (`choose()`) è già stata presentata nel capitolo 5. Analogamente alla `permn()`, la funzione `combn()` consente di visualizzare tutte le differenti permutazioni di un certo insieme:

```
> choose(5,3)
[1] 10

> t( combn(LETTERS[1:5],3) )
      [,1] [,2] [,3]
 [1,] "A"  "B"  "C"
 [2,] "A"  "B"  "D"
 [3,] "A"  "B"  "E"
 [4,] "A"  "C"  "D"
 [5,] "A"  "C"  "E"
 [6,] "A"  "D"  "E"
 [7,] "B"  "C"  "D"
 [8,] "B"  "C"  "E"
 [9,] "B"  "D"  "E"
[10,] "C"  "D"  "E"
```

La libreria `combinat` non contiene però un'analoga funzione per la visualizzazione di tutte le differenti disposizioni semplici di *k* oggetti estratti da un insieme di *n* oggetti. Le disposizioni si ottengono però considerando tutte le permutazioni di ciascuna delle combinazioni ottenute, pertanto:

Esercizio: ricorrendo alle funzioni `permn()` e `combn()` illustrate in questo capitolo, implementare una funzione in grado di restituire l'elenco delle differenti disposizioni semplici di *k* oggetti estratti da un insieme di *n* oggetti.

```
my.disp = function(v, k){

a=t( combn(v,k) )
disp=NULL
for(i in 1:choose(length(v),k))
      disp=rbind(disp, do.call(rbind, permn(a[i,])) )
return (disp)
}

> my.disp(LETTERS[1:5],3)

      [,1] [,2] [,3]
 [1,] "A"  "B"  "C"
 [2,] "A"  "C"  "B"
 [3,] "C"  "A"  "B"
 [4,] "C"  "B"  "A"
 [5,] "B"  "C"  "A"
 [6,] "B"  "A"  "C"
 [7,] "A"  "B"  "D"
 [8,] "A"  "D"  "B"
 [9,] "D"  "A"  "B"
[10,] "D"  "B"  "A"
[11,] "B"  "D"  "A"
[12,] "B"  "A"  "D"
[13,] "A"  "B"  "E"
[14,] "A"  "E"  "B"
[15,] "E"  "A"  "B"
[16,] "E"  "B"  "A"
[17,] "B"  "E"  "A"
[18,] "B"  "A"  "E"
[19,] "A"  "C"  "D"
[20,] "A"  "D"  "C"
[21,] "D"  "A"  "C"
[22,] "D"  "C"  "A"
[23,] "C"  "D"  "A"
[24,] "C"  "A"  "D"
[25,] "A"  "C"  "E"
[26,] "A"  "E"  "C"
[27,] "E"  "A"  "C"
[28,] "E"  "C"  "A"
```

```
[29,] "C"  "E"  "A"
[30,] "C"  "A"  "E"
[31,] "A"  "D"  "E"
[32,] "A"  "E"  "D"
[33,] "E"  "A"  "D"
[34,] "E"  "D"  "A"
[35,] "D"  "E"  "A"
[36,] "D"  "A"  "E"
[37,] "B"  "C"  "D"
[38,] "B"  "D"  "C"
[39,] "D"  "B"  "C"
[40,] "D"  "C"  "B"
[41,] "C"  "D"  "B"
[42,] "C"  "B"  "D"
[43,] "B"  "C"  "E"
[44,] "B"  "E"  "C"
[45,] "E"  "B"  "C"
[46,] "E"  "C"  "B"
[47,] "C"  "E"  "B"
[48,] "C"  "B"  "E"
[49,] "B"  "D"  "E"
[50,] "B"  "E"  "D"
[51,] "E"  "B"  "D"
[52,] "E"  "D"  "B"
[53,] "D"  "E"  "B"
[54,] "D"  "B"  "E"
[55,] "C"  "D"  "E"
[56,] "C"  "E"  "D"
[57,] "E"  "C"  "D"
[58,] "E"  "D"  "C"
[59,] "D"  "E"  "C"
[60,] "D"  "C"  "E"
```

Notare il funzionamento della funzione di sistema `do.call()`, utilizzata in questo frangente per applicare la funzione `rbind()` a tutte le possibili permutazioni restituite come lista dalla funzione `permn()`.

Capitolo XIX

Funzioni per il trattamento delle distribuzioni di Variabili Casuali

R dispone di tutta una serie di funzioni di libreria (metodi) per il controllo e la gestione delle variabili casuali sia discrete che continue.

Per ogni variabili casuale sono generalmente disponibili quattro funzioni per:

- il computo dei valori della funzione di massa di probabilità o densità di probabilità,
- il computo dei valori della funzione di ripartizione,
- il computo dei quantili,
- la generazione di numeri casuali sulla base del modello probabilistico desiderato.

In generale, le funzioni che iniziano con la lettera " *d* " sono dedicate al calcolo dei valori della funzione di massa di probabilità o densità di probabilità, con argomento necessario un vettore di valori sul dominio di definizione della funzione stessa; le funzioni che iniziano con la lettera " *p* " sono dedicate al calcolo dei valori della funzione di distribuzione o ripartizione, con argomento necessario un vettore di valori (quantili) sul dominio di definizione della variabile casuale; le funzioni che iniziano con la lettera " *q* " sono dedicate al computo dei quantili associati ad un vettore di valori di probabilità; infine, le funzioni che iniziano con la lettera " *r* " sono dedicate alla replicazione di *n* (argomento necessario) esperimenti casuali indipendenti secondo lo schema probabilistico tipico della variabile casuale considerata.

Variabili Casuali Discrete

Questo sotto paragrafo sarà esclusivamente dedicato alla presentazione dei principali metodi per il trattamento delle variabili casuali discrete.

B. Bertaccini, *Introduzione alla Statistica Computazionale con R*, ISBN (print) 978-88-6453-673-6, ISBN (online) 978-88-6453-674-3, CC BY 4.0, 2018 Firenze University Press.

Nelle due tabelle riportate in successione sono illustrate le funzioni richiamate in precedenza relativamente ai modelli probabilistici di tipo Binomiale, di Poisson e Binomiale Negativo. Come noto, il modello probabilistico di Bernoulli è un caso particolare del modello di Binomiale per cui con le funzioni per la Binomiale si ottengono anche i valori caratteristici della variabile casuale di Bernoulli.

	Binomiale	**Bernoulli**
Funzione di massa di probabilità	`dbinom(x, size, prob)`	`dbinom(x, size=1, prob)`
Funzione di distribuzione	`pbinom(q, size, prob)`	`pbinom(q, size=1, prob)`
Quantili	`qbinom(p, size, prob)`	`qbinom(p, size=1, prob)`
Generazione di numeri casuali	`rbinom(n, size, prob)`	`rbinom(n, size=1, prob)`

	Poisson	**Binomiale Negativa**
Funzione di massa di probabilità	`dpois(x, lambda)`	`dnbinom(x, size, prob)`
Funzione di distribuzione	`ppois(q, lambda)`	`pnbinom(q, size, prob)`
Quantili	`qpois(p, lambda)`	`qnbinom(p, size, prob)`
Generazione di numeri casuali	`rpois(n, lambda)`	`rnbinom(n, size, prob)`

Alcuni esempi.

Per la variabile casuale di Bernoulli:

```
> dbinom(c(0,1),size=1,prob=.5)
[1] 0.5 0.5

> dbinom(c(0,1),size=1,prob=.2)
[1] 0.8 0.2

> pbinom(c(0,1),size=1,prob=.5)
[1] 0.5 1.0

> pbinom(c(0,1),size=1,prob=.2)
[1] 0.8 1.0

> qbinom(c(0,.5,.7,.8,.81,1),size=1,prob=.2)
[1] 0 0 0 0 1 1
```

In particolare, se si desidera simulare un esperimento che prevede la registrazione del numero di teste (successi) a seguito del lancio ripetuto di una moneta bilanciata per 10 volte:

```
> rbinom(10,size=1,p=.5)
 [1] 0 0 0 1 1 1 0 1 0 0
```

Per lo stesso esperimento, se la moneta risultasse sbilanciata con una probabilità di testa (successo) pari a 0.2:

```
> rbinom(10,size=1,p=.2)
 [1] 1 0 0 0 0 0 0 0 0 1
```

Per la variabile casuale Binomiale:

```
> dbinom(0:10,size=10,p=.5)
 [1] 0.0009765625 0.0097656250 0.0439453125 0.1171875000 0.2050781250
 [6] 0.2460937500 0.2050781250 0.1171875000 0.0439453125 0.0097656250
[11] 0.0009765625

> sum(dbinom(0:10,size=10,p=.5))
[1] 1

> pbinom(0:10,size=10,p=.5)
 [1] 0.0009765625 0.0107421875 0.0546875000 0.1718750000 0.3769531250
 [6] 0.6230468750 0.8281250000 0.9453125000 0.9892578125 0.9990234375
[11] 1.0000000000
```

In particolare, se si desidera ripetere 20 volte l'esperimento precedente, che prevedeva la registrazione del numero di teste (successi) a seguito del lancio ripetuto di una moneta bilanciata per 10 volte:

```
> rbinom(20,size=10,p=.5)
 [1] 4 4 3 5 6 2 3 5 5 5 5 2 6 5 5 3 5 6 4 4
```

Per la variabile casuale di Poisson:

```
> dpois(0:10,lambda=1)
 [1] 3.678794e-01 3.678794e-01 1.839397e-01 6.131324e-02 1.532831e-02
 [6] 3.065662e-03 5.109437e-04 7.299195e-05 9.123994e-06 1.013777e-06
[11] 1.013777e-07
```

Sebbene, il dominio della variabile casuale di Poisson sia identificabile con l'insieme dei Naturali, se il parametro caratteristico della distribuzione è 1, le probabilità associate a valori della variabile casuale più grandi di 10 sono di fatto trascurabili:

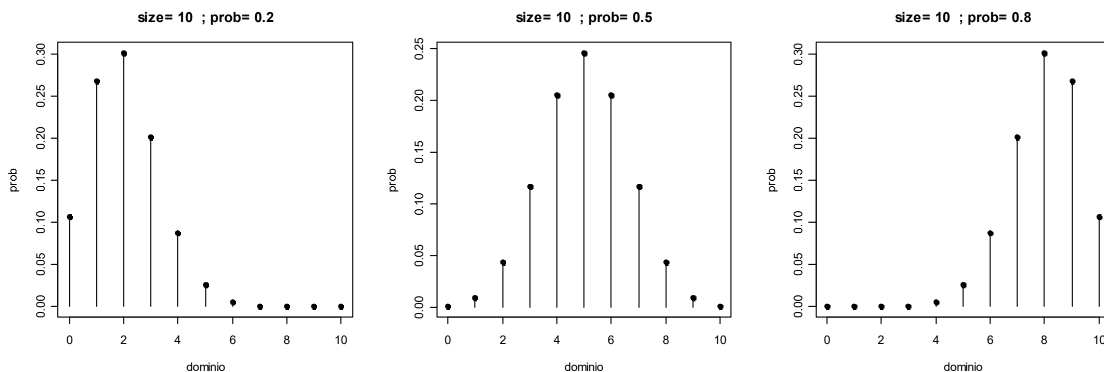
```
> dpois(11,lambda=1)
[1] 9.216156e-09

> ppois(0:10,lambda=1)
 [1] 0.3678794 0.7357589 0.9196986 0.9810118 0.9963402 0.9994058
 [7] 0.9999168 0.9999898 0.9999989 0.9999999 1.0000000
```

```
> rpois(15,lambda=1)
 [1] 0 2 1 0 4 1 1 2 1 0 1 2 1 0 1
```

Esercizio: costruire una funzione in grado di tracciare il grafico relativo alla funzione di massa di probabilità di una variabile casuale di tipo Binomiale.

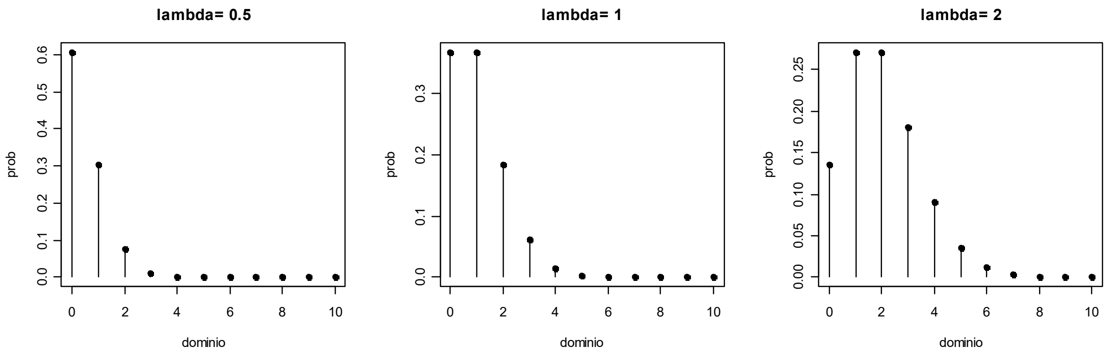
```
pmf.binom = function(size, prob){
      y=dbinom(0:size, size, prob)
      plot(0:size, y, type="h", xlab="dominio", ylab="prob")
      title(paste("size=", size," ; prob=", prob))
      points(0:size,y,pch=19)
}
```

```
> par(mfrow=c(1,3))
> pmf.binom(10,.2)
> pmf.binom(10,.5)
> pmf.binom(10,.8)
```



La stessa funzione può essere facilmente modificata per tracciare il grafico relativo alla funzione di massa di probabilità di una Poisson:

```
pmf.pois = function(lambda,max=10){

      y=dpois(0:max, lambda)
      plot(0:max, y, type="h", xlab="dominio", ylab="prob")
      title(paste("lambda=", lambda))
      points(0:max,y,pch=19)
}

> par(mfrow=c(1,3))
> pmf.pois(.5)
> pmf.pois(1)
> pmf.pois(2)
```



Variabili Casuali Continue

Questo sotto paragrafo sarà esclusivamente dedicato alla presentazione dei principali metodi per il trattamento delle variabili casuali continue.

Nelle due tabelle riportate in successione sono illustrate le funzioni richiamate in precedenza e relative ai modelli probabilistici di tipo Uniforme, Normale, Chi-Quadrato, *t* di Student, *F* di Fisher, Gamma e Beta. Tra le parentesi che definiscono la *call* della funzione sono indicati i parametri necessari e quelli facoltativi con gli eventuali valori di default.

	Uniforme	**Normale**
Funzione di densità di probabilità	`dunif(x, min=0, max=1)`	`dnorm(x, mean=0, sd=1)`
Funzione di distribuzione	`punif(q, min=0, max=1)`	`pnorm(q, mean=0, sd=1)`
Quantili	`qunif(p, min=0, max=1)`	`qnorm(p, mean=0, sd=1)`
Generazione di numeri casuali	`runif(n, min=0, max=1)`	`rnorm(n, mean=0, sd=1)`

	Chi-Quadrato	***t* di Student**
Funzione di densità di probabilità	`dchisq(x, df)`	`dt(x, df)`
Funzione di distribuzione	`pchisq(q, df)`	`pt(q, df)`
Quantili	`qchisq(p, df)`	`qt(p, df)`
Generazione di numeri casuali	`rchisq(n, df)`	`rt(n, df)`

	F di Fisher	Gamma
Funzione di densità di probabilità	`df(x, df1, df2)`	`dgamma(x, shape, scale)`
Funzione di distribuzione	`pf(q, df1, df2)`	`pgamma(q, shape, scale)`
Quantili	`qf(p, df1, df2)`	`qgamma(p, shape, scale)`
Generazione di numeri casuali	`rf(n, df1, df2)`	`rgamma(n, shape, scale)`

	Beta
Funzione di densità di probabilità	`dbeta(x, shape1, shape2)`
Funzione di distribuzione	`pbeta(q, shape1, shape2)`
Quantili	`qbeta(p, shape1, shape2)`
Generazione di numeri casuali	`rbeta(n, shape1, shape2)`

Alcuni esempi.

Per la variabile casuale Uniforme continua (la funzione `runif()` è stata presentata nel capitolo 17):

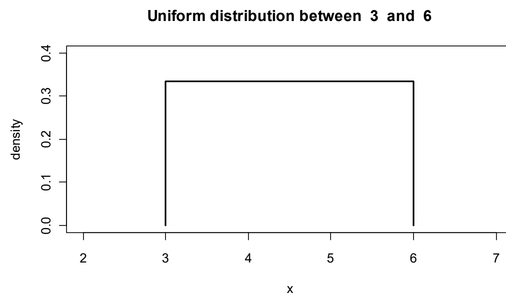
```
> dunif(1,0,2)
[1] 0.5
```

```
> runif(5,1,10)
 [1] 1.614000 5.818778 6.309254 5.880768 9.739542
```

Esercizio: costruire una funzione in grado di tracciare il grafico relativo alla funzione di densità di probabilità di una variabile casuale Uniforme continua, definita nell'intervallo [a,b].

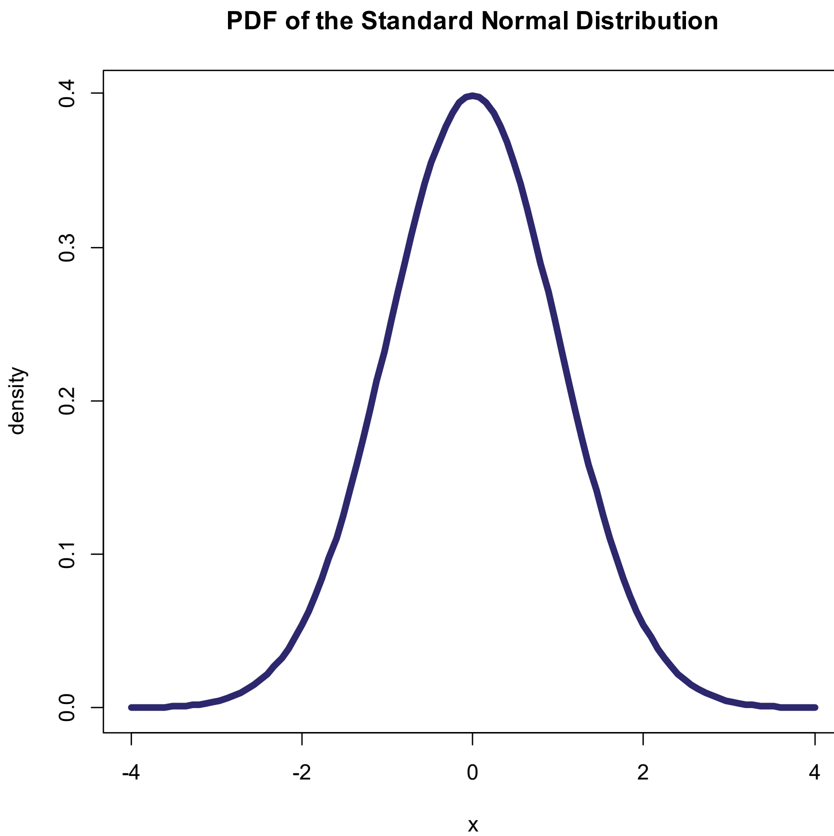
```
pdf.unif=function(a,b, xlim=c(-5,5)){
v=c(a,0)
v=rbind(v, c(a,dunif(a,a,b)))
v=rbind(v, c(b,dunif(b,a,b)))
v=rbind(v, c(b,0))
ylim=c(0, 1.2/(b-a))
plot(v, type="l",lwd=2,xlab="x",ylab="density",xlim=xlim,ylim=ylim)
title(paste("Uniform distribution between ",a," and ",b))
}
```

```
> pdf.unif(3,6, xlim=c(2,7))
```



Per la variabile casuale Normale:

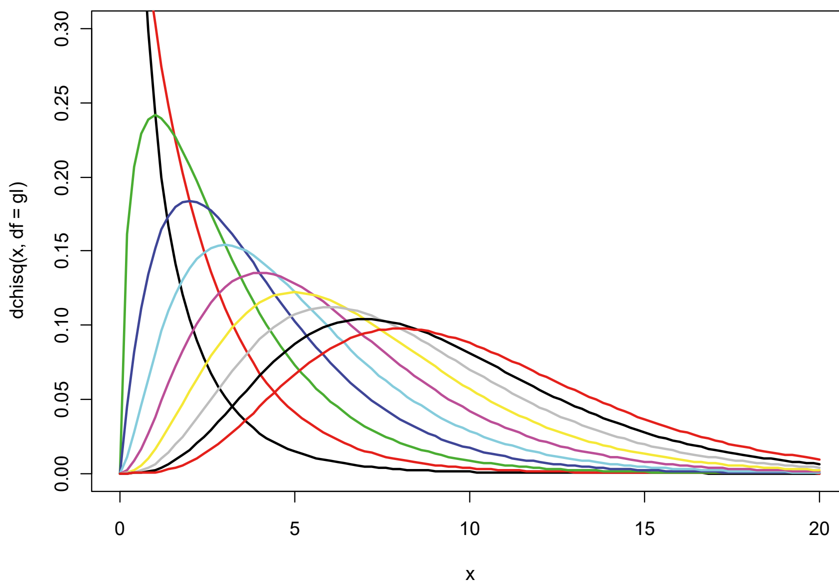
```
> curve(dnorm(x,0,1),from=-4,to=4,ylab="density", lwd=5,col="navy")
> title("PDF of the Standard Normal Distribution")
```



```
> pnorm(1.96,mean=0,sd=1)-pnorm(-1.96,mean=0,sd=1)
[1] 0.9500042

> rnorm(10,mean=1,sd=2)
 [1]  0.72984668  0.72645173 -0.35997524 -0.76682359  3.55793676
 [6] -1.04520934 -0.05679271  3.33588000  0.29886852  4.09574856
```

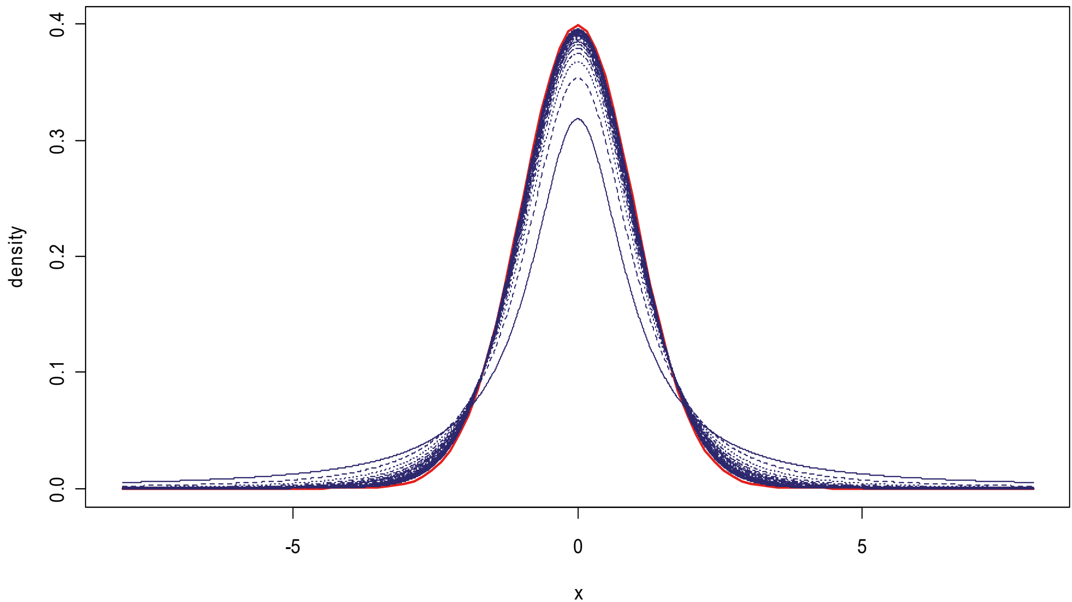
Esercizio: costruire una funzione che illustri la variazione delle forme distributive della variabile casuale Chi-Quadro al crescere dei suoi gradi di libertà.

```
chisq.dens = function(dfmax=10){

flag=F
for (gl in 1:dfmax){
      if (gl>1) flag=T
      curve(dchisq(x,df=gl),from=0,to=20,ylim=c(0,0.3),add=flag,lwd=2,
      col=gl)
      }
title("Densità della variabile Chi-Quadro al crescere dei g.l.")
}

> chisq.dens()
```



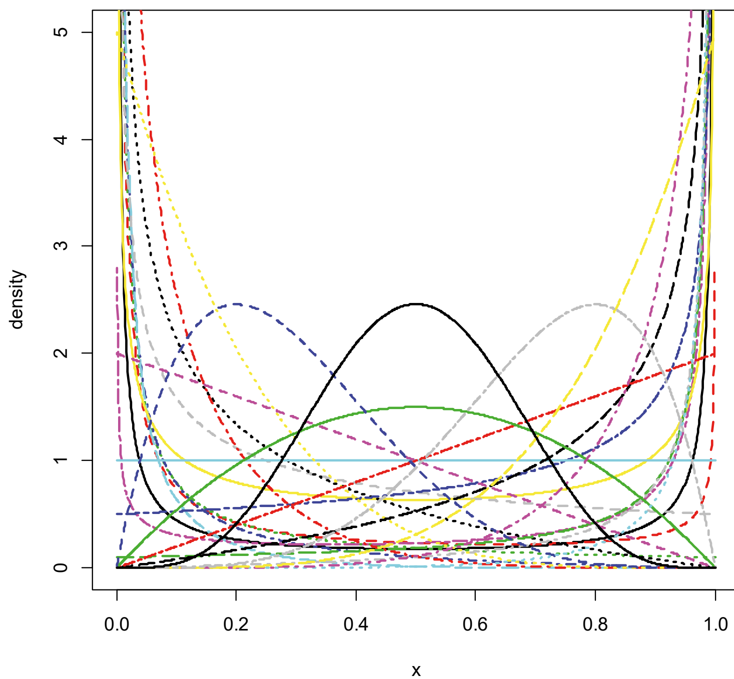
Esercizio: costruire una funzione che illustri la convergenza della *t* di Student, al crescere dei suoi gradi di libertà, alla Normale Standard.

```
conver.t = function(){

for(i in 1:30)
      curve(dt(x,df=i),from=-8,to=8,ylim=c(0,.4),col="navy",lty=i,
           add=(i>2),ylab="density")
title("Convergenza della t di Student alla Normale Standard")

curve(dnorm(x),from=-8,to=8,col="red",lwd=2,add=T)
}

> conver.t()
```



Esercizio: costruire una funzione che illustri le diverse configurazioni assumibili dalla funzione di densità di una variabile casuale Beta al variare dei suoi parametri.

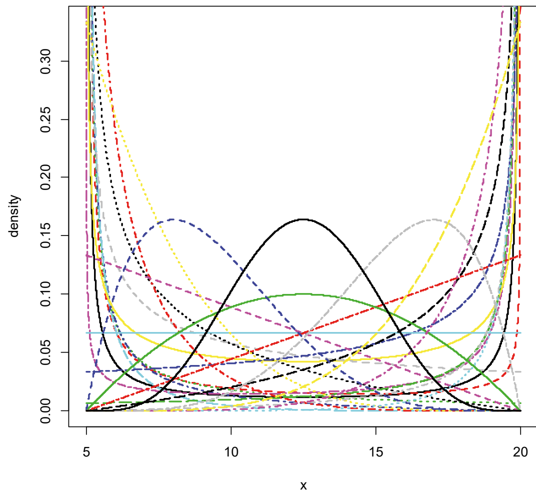
```
beta.den <- function(a,b){

if (!is.vector(a)) stop("first parameter must be a vector")
if (!is.vector(b)) stop("second parameter must be a vector")

plot(NA,xlim=c(0,1),ylim=c(0,5),xlab="x",ylab="density")
x=seq(0,1,0.001)
i=1
for (aa in a)
  for(bb in b){
      lines(x,dbeta(x,aa,bb), lty=i, col=i, lwd=2)
      i=i+1
      }
}

> alpha=c(0.1,.5,1,2,5)
> beta=c(0.1,.5,1,2,5)
> beta.den(alpha,beta)
```



Esercizio: ripetere l'esercizio precedente con una variabile Beta generalizzata al campo di variazione [l1, l2], la cui funzione di densità è:

$$f(x;\alpha,\beta)=\frac{1}{B(\alpha,\beta)}\left(\frac{x-l_1}{l_2-l_1}\right)^{a-1}\left(\frac{l_2-x}{l_2-l_1}\right)^{b-1}\frac{1}{l_2-l_1}$$

```
betagen.den <- function(a,b,l1,l2){

if (l2<l1) {aux=l2; l2=l1; l1=aux}
if (!is.vector(a)) stop("first parameter must be a vector")
if (!is.vector(b)) stop("second parameter must be a vector")

dbeta.gen <- function(x,a,b,l1,l2){
  return(1/beta(a,b)  *  ((x-l1)/(l2-l1))^(a-1)  *  ((l2-x)/(l2-l1))^(b-1)
          * 1/(l2-l1))
  }
# codominio riscalato sulla base del campo di variazione di x
plot(NA,xlim=c(l1,l2),ylim=c(0,5/(l2-l1)),xlab="x",ylab="density")
x=seq(l1,l2,0.001)
i=1
for (aa in a)
  for(bb in b){
      lines(x,dbeta.gen(x,aa,bb,l1,l2), lty=i, col=i, lwd=2)
      i=i+1
      }
}

> alpha=c(0.1,.5,1,2,5)
> beta=c(0.1,.5,1,2,5)
> betagen.den(alpha,beta,20,5)
```



Capitolo XX

Simulazioni Monte Carlo

Le simulazioni Monte Carlo sono il sale della Statistica computazionale. Le simulazioni Monte Carlo sono tecniche di simulazione stocastica che, assieme ai metodi di simulazione dinamico-deterministica, sono oggi uno dei principali ambiti di applicazione delle scienze informatiche[21].

I metodi Monte Carlo sono metodi improntati all'ottenimento di stime a fronte di simulazioni basate su un elevato numero di replicazioni, tra loro indipendenti, di un esperimento casuale. Sono in generale utilizzati per prevedere il comportamento di una realtà complessa caratterizzata da un certo margine di incertezza, quando è comunque noto ciò che ne stabilisce il funzionamento.

Le origini della tecnica risalgono agli anni 40 e sono da attribuire a scienziati di spicco del Progetto Manhattan come Enrico Fermi, John von Neumann e Stanislaw Marcin Ulam. Il nome Monte Carlo lo si deve invece al fisico matematico Nicholas Constantine Metropolis che, qualche anno più tardi, associò la caratteristica di aleatorietà delle replicazioni al famoso casinò del Principato di Monaco.

L'inferenza statistica è ovviamente uno dei principali ambiti di applicazione di questo tipo di simulazioni. Si pensi, ad esempio, ad un gioco d'azzardo che ha regole prestabilite ma il cui esito è governato dal caso. È possibile a priori valutarne la vincita attesa per comprendere se conviene o meno tentare la sorte? Se le regole sono semplici, la valutazione è spesso intuitiva o al più ricavabile tramite dimostrazione analitica. Quando le regole si complicano,

[21] Le tecniche di simulazione stocastica si differenziano da quelle deterministiche per la presenza di parametri in ingresso di tipo casuale e funzioni che effettuano chiamate agli algoritmi di generazione dei numeri (pseudo) casuali. Le tecniche di simulazione dinamica si differenziano da quelle di tipo statico per la gestione della variabile tempo, in quanto presuppongono la necessità di studiare il comportamento di un certo risultato al variare del tempo.

B. Bertaccini, *Introduzione alla Statistica Computazionale con R*, ISBN (print) 978-88-6453-673-6, ISBN (online) 978-88-6453-674-3, CC BY 4.0, 2018 Firenze University Press.

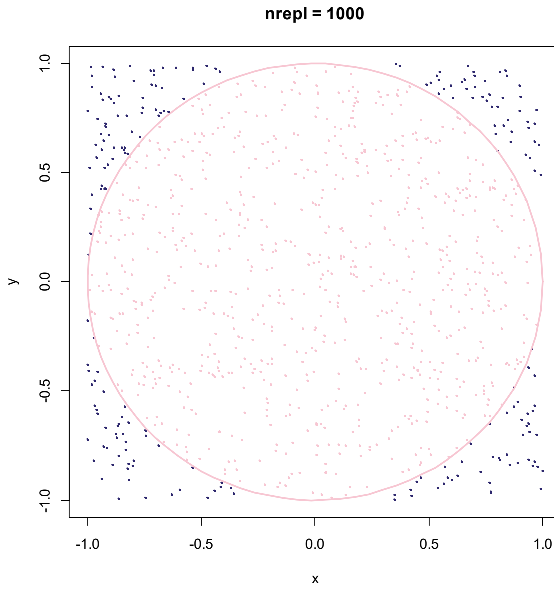
procedere per via analitica potrebbe però rivelarsi molto arduo. In altre parole, se volessimo valutare la probabilità che esca un "sei" su un dado truccato, procedere per via frequentista sembra essere la soluzione più saggia, nella piena consapevolezza che il risultato ottenuto è solo un'approssimazione (più o meno buona a seconda del numero di replicazioni effettuate) della vera probabilità di fare "sei". Ma oggi, grazie ai moderni calcolatori, queste approssimazioni possono considerarsi davvero molto buone.

Per acquisire familiarità con i metodi Monte Carlo, si provi a risolvere il seguente problema.

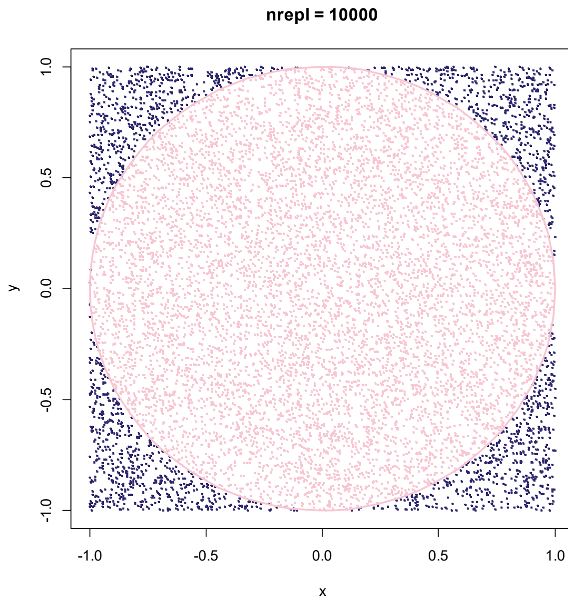
Esercizio: ottenere una approssimazione del π (pi-greco) tramite metodo Monte Carlo.

Per risolvere l'esercizio conviene ricordare che l'area di un cerchio di raggio *r* è pari a $A = \pi \cdot r^2$, e che il cerchio unitario (di raggio 1 e con area $A = \pi$), centrato sull'origine, può essere inscritto in un quadrato il cui lato è pari a 2 (da -1 ad 1) e la cui area è 4. Pertanto, generando punti in maniera casuale all'interno di questo quadrato, è possibile calcolare la proporzione di questi che cadono all'interno della circonferenza (descritta dall'equazione $x^2 + y^2 = 1$), proporzione che a livello teorico deve risultare pari a $\frac{\pi}{4}$. Ovviamente, maggiore sarà il numero dei punti generati, più precisa sarà la stima di questo rapporto, dal quale è poi banale ricavare la stima di π.

```
pi.greco=function(nrepl=10000){

xy=runif(nrepl*2, min=-1, max=1)
dim(xy)=c(nrepl,2)

inside=apply(xy^2,1,sum)<1
plot(xy, pch=20, cex=.5, col=c("navy","pink")[inside+1],
      xlab="x", ylab="y", main=paste("nrepl =",nrepl))

a=seq(0,2*pi,length.out=100)
xy.circle=cbind(cos(a),sin(a))
lines(xy.circle,col="pink",lwd=2)

return( sum(inside)/nrepl *4)
}

> pi.greco(1000)
[1] 3.144
```




```
> pi.greco(10000)
[1] 3.1428
```



Sulla base dello stesso principio, con i metodi Monte Carlo è possibile stimare anche l'area di un lago dai bordi comunque frastagliati, se questo si trova all'interno di un'area rettangolare dai lati di lunghezza nota.

Ovviamente, la bontà dei risultati prodotti dai questi metodi è fortemente correlata con il peso assunto dalla componente aleatoria nel processo simulativo e, di conseguenza, con la qualità degli algoritmi di generazione dei numeri casuali richiamati.

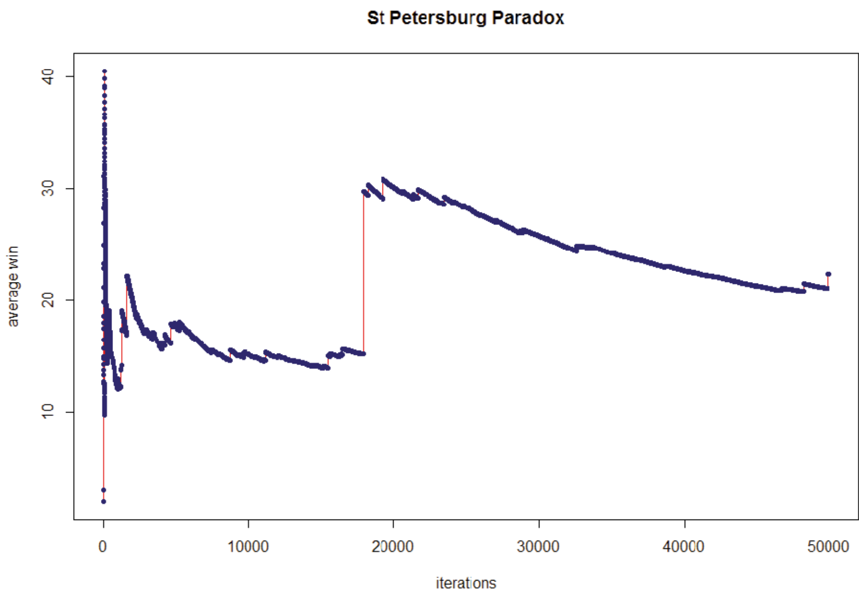
Come anticipato, la replicabilità di un esperimento simulato per via computazionale abbinata alle velocità di calcolo degli odierni processori costituiscono il presupposto ideale per stimare la probabilità di vincita o il valore atteso di un certo gioco d'azzardo.

Si pensi ad esempio al famoso **Paradosso di San Pietroburgo**. Nella teoria del calcolo delle probabilità, il paradosso di San Pietroburgo descrive un particolare gioco d'azzardo basato su una variabile casuale con valore atteso infinito. Ciononostante, stante le sue regole, pare ragionevole considerare adeguato solo il pagamento di una minima somma per poter partecipare al gioco. Il gioco consiste infatti nel lanciare ripetutamente una moneta (non truccata) finché non esce *Croce*, che dà luogo alla vincita. L'entità della vincita quindi dipende dal numero di lanci complessivi: se esce *Croce* al primo lancio, si vincono 2 unità monetarie. Se esce *Testa*, si raddoppia la vincita ad ogni lancio successivo. In breve, si vincono 2^k unità monetarie se la moneta è stata lanciata k volte per ottenere la prima *Croce*.

Esercizio*:* si implementi una funzione, tramite metodo Monte Carlo, che simuli il gioco d'azzardo denominato ***paradosso di San Pietroburgo*** e ne calcoli la vincita media (attesa) su un numero sufficientemente grande di replicazioni. Si provveda anche a disegnare l'andamento della vincita media all'aumentare del numero delle replicazioni.

```
sanpietroburgo <- function(nrepl=10000){

vett.win=NULL

for(i in 1:nrepl){
      print (i)
      nlanci=1
      while( sample(0:1,1)==0 ){
            nlanci=nlanci+1
            }
      win=2^nlanci

      vett.win=c(vett.win, win)
}

return( vett.win )
}


> v=sanpietroburgo(50000)
```

```
> table(v)
v
    2     4     8    16    32    64   128   256   512  1024  2048  4096
25017 12554  6286  3139  1485   782   367   168   110    44    19    12

  8192  16384  32768  65536 262144
     8      5      2      1      1

> mean(v)
[1] 22.36852


> plot(mean.w,type="l",lwd=1,col="red",xlab="iterations",
       ylab="average win")
> points(mean.w,col="navy",cex=.7,pch=19)
> title(main="St Petersburg Paradox")
```



Il grafico evidenzia il momento del processo simulativo durante il quale si è verificata la vincita di 262144 unità monetarie. Nonostante ciò, la vincita attesa si attesta sull'ordine delle 22 unità, valore questo certamente condizionato dalla vincita eccezionale verificatasi dopo oltre 18000 replicazioni (prima di questa la vincita media era vicina a 16) e comunque ben lontana dal valore atteso teorico del gioco.

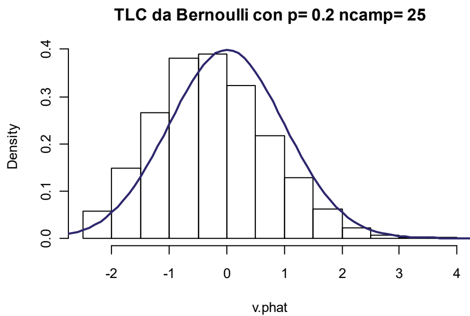
Infine, le caratteristiche del metodo Monte Carlo rendono l'approccio simulativo particolarmente appropriato nella dimostrazione empirica di importanti teoremi propri dell'Inferenza Statistica.

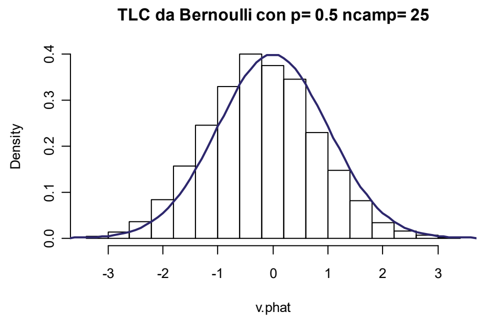
Esercizio: costruire una funzione per dimostrare empiricamente la tesi del **Teorema Limite Centrale**, nel caso di estrazioni da una popolazione caratterizzata da distribuzione di Bernoulli.

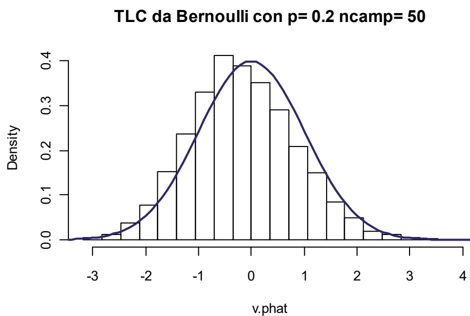
```
tlc = function(nrepl,n,p){

v.phat=NULL
for(i in 1:nrepl){
  sam=rbinom(n,1,p)
  phat=mean(sam)
  v.phat=c(v.phat, (phat-p)/sqrt(p*(1-p)/n) )
  }
br=sort(unique(v.phat))
hist(v.phat, breaks=br, prob=T, main="")
curve(dnorm(x),from=-5,to=5,add=T,lwd=2,col="navy")
title(paste("TLC da Bernoulli con p=",p,"ncamp=",n))
}
```

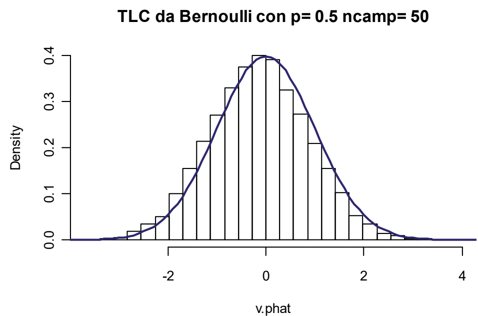
```
> par(mfrow=c(2,2))
```

```
> tlc(10000,25,.2)
> tlc(10000,25,.5)
> tlc(10000,50,.2)
> tlc(10000,50,.5)
```









Capitolo XXI

Selezione di esercizi svolti

Esercizio: in riferimento al dataset d'esempio `iris` (richiamato nel capitolo 11), implementare una funzione che restituisca un oggetto di tipo lista contenente elementi bidimensionali (uno per ogni specie di iris censito nel dataset) composto dai quantili: 0, 0.10, 0.25, 0.50, 0.75, 0.90 e 1 di tutte le variabili quantitative presenti nel dataset.

```
descript.iris = function(){

data(iris)
s=split(iris[,-5],iris$Species)
specie=unique(iris$Species)
l=list()
for (i in specie)
    l[[i]]=sapply(s[[i]],quantile,probs=c(0,.1,.25,.5,.75,.90,1))

return(l)
}

> descript.iris()
$setosa
     Sepal.Length Sepal.Width Petal.Length Petal.Width
0%           4.30       2.300        1.000        0.10
10%          4.59       3.000        1.300        0.19
25%          4.80       3.200        1.400        0.20
50%          5.00       3.400        1.500        0.20
75%          5.20       3.675        1.575        0.30
90%          5.41       3.900        1.700        0.40
100%         5.80       4.400        1.900        0.60
```

B. Bertaccini, *Introduzione alla Statistica Computazionale con R*, ISBN (print) 978-88-6453-673-6, ISBN (online) 978-88-6453-674-3, CC BY 4.0, 2018 Firenze University Press.

```
$versicolor
     Sepal.Length Sepal.Width Petal.Length Petal.Width
0%           4.90       2.000         3.00        1.00
10%          5.38       2.300         3.59        1.00
25%          5.60       2.525         4.00        1.20
50%          5.90       2.800         4.35        1.30
75%          6.30       3.000         4.60        1.50
90%          6.70       3.110         4.80        1.51
100%         7.00       3.400         5.10        1.80

$virginica
     Sepal.Length Sepal.Width Petal.Length Petal.Width
0%          4.900       2.200        4.500        1.40
10%         5.800       2.590        4.900        1.79
25%         6.225       2.800        5.100        1.80
50%         6.500       3.000        5.550        2.00
75%         6.900       3.175        5.875        2.30
90%         7.610       3.310        6.310        2.40
100%        7.900       3.800        6.900        2.50
```

Esercizio: implementare l'algoritmo di ordinamento denominato **Selection Sort** (per ordinare un vettore di lunghezza *n*, si fa scorrere l'indice *i* da 1 a *n*-1 cercando il più piccolo elemento della sottosequenza *i* … *n*, per poi scambiarlo con l'elemento *i*-esimo).

```
sel.sort <- function(v){

if (!is.vector(v)) stop("Input parameter must be a vector")

n=length(v)
for (i in 1:(n-1)){
      min.index=i

#cerco il più piccolo elemento nella sottosequenza
#e memorizzo la posizione
      for (j in i:n)
            if(v[j]<v[min.index]) min.index=j

      aux=v[i]
      v[i]=v[min.index]
      v[min.index]=aux

      }

return(v)
}
```

```
> a=sample(1:15)
> a
 [1] 15  2  4  1 13  9  7 14 11  5  8  6  3 10 12

> sel.sort(a)
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
```

Esercizio: implementare l'algoritmo di ordinamento denominato **Bubble Sort** (ogni coppia di elementi adiacenti della lista viene comparata e se sono nell'ordine sbagliato vengono invertiti. L'algoritmo scorre poi tutta la lista finché non saranno più eseguiti scambi). L'algoritmo deve il suo nome al modo in cui gli elementi vengono ordinati, con quelli più piccoli che "risalgono" verso le loro posizioni corrette all'interno del vettore così come fanno le bollicine in un bicchiere d'acqua gassata.

```
bubble <- function(v){
if (!is.vector(v)) stop("Input parameter must be a vector")

n=length(v)
scambio=T
while (scambio==T){
      scambio=F
      for (i in 1:(n-1))
            if(v[i]>v[i+1]){
                  aux=v[i]
                  v[i]=v[i+1]
                  v[i+1]=aux
                  scambio=T
                  }
      }
return(v)
}
```

```
> a=sample(1:15)
> a
 [1] 11  8  2  7  4  6  1 15  3 10  9 14 13 12  5

> bubble(a)
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
```

```
> a=sample(150)

> t0=Sys.time()
> v1=sort(a)
> Sys.time()-t0
Time difference of 0 secs

> t0=Sys.time()
> v1=sel.sort(a)
> Sys.time()-t0
Time difference of 0.01563096 secs

> t0=Sys.time()
> v1=bubble(a)
> Sys.time()-t0
Time difference of 0.03124905 secs
```

Performance degli algoritmi Selection Sort e Bubble Sort rispetto all'algoritmo di ordinamento interno di **R.**

Esercizio: costruire una funzione che riproduca **la distribuzione di frequenza** delle realizzazioni di un certo fenomeno collettivo: in output devono essere riprodotte in relazione a ciascun valore osservato, le loro frequenze assolute, relative e cumulate.

```
distr.freq = function(v){

if (!is.vector(v)) stop("Input parameter must be a vector")

mod=sort(unique(v))

fr=NULL
for (i in mod)
      fr=c(fr, sum(v==i))

num.mod=length(mod)
l.t=matrix(0,nrow=num.mod,ncol=num.mod)
l.t[lower.tri(l.t,diag=T)]=1

m=cbind(mod, fr, l.t%*%fr, fr/sum(fr), l.t%*%(fr/sum(fr)))
colnames(m)=c("mod","fa","fa.cum","fr","fr.cum")

return (m)
}
```

```
> v=c(1,3,6,7,2,2,4,2,5,6,7,9)
> distr.freq(v)
     mod fa fa.cum         fr     fr.cum
[1,]   1  1      1 0.08333333 0.08333333
[2,]   2  3      4 0.25000000 0.33333333
[3,]   3  1      5 0.08333333 0.41666667
[4,]   4  1      6 0.08333333 0.50000000
[5,]   5  1      7 0.08333333 0.58333333
[6,]   6  2      9 0.16666667 0.75000000
[7,]   7  2     11 0.16666667 0.91666667
[8,]   9  1     12 0.08333333 1.00000000
```

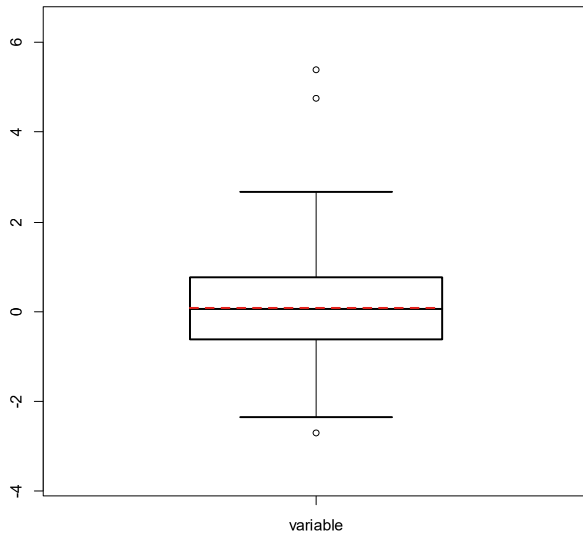
Esercizio: costruire una funzione che riproduca il **box-plot** delle realizzazioni di un certo fenomeno collettivo.

Regole di costruzione di un box-plot:

a) il rettangolo indica il grado di dispersione del 50% dei valori centrali della distribuzione (ovvero dei valori compresi tra il primo e terzo quartile);
b) i segmenti che partono dai lati minori del rettangolo indicano l'estensione delle code di sinistra e di destra della distribuzione. Tali segmenti hanno un punto di troncamento a destra identificato dal più grande dei valori inferiori a $Q_{0.75}+1.5DI$, a sinistra identificato dal più piccolo dei valori superiori a $Q_{0.25}-1.5DI$ (dove $DI=Q_{0.75}-Q_{0.25}$);
c) eventuali punti esterni ai punti di troncamento sono potenziali valori anomali e devono essere rappresentati come singoli punti.

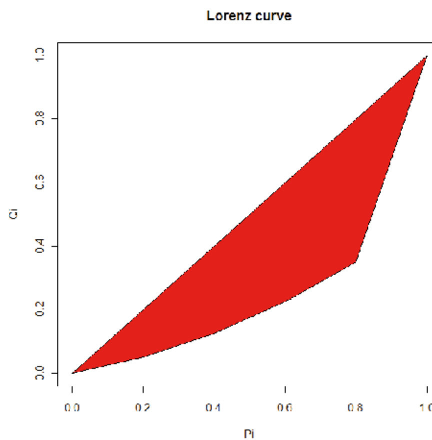
```
my.boxpl = function(v,wdt=.5,wdt2=.3){

if (!is.vector(v)) stop("Input parameter must be a vector")
q=quantile(v)

y.lim=c(q[1]-1,q[5]+1)
DI=q[4]-q[2]

y.box=c(q[2],q[2],q[4],q[4],q[2])
x.box=c(1-wdt,1+wdt,1+wdt,1-wdt,1-wdt)
x=x.box[1:2]
x2=c(1-wdt2,1+wdt2)

plot (x.box, y.box, type="l", ylim=y.lim, xlim=c(0,2), lwd=2, xlab="",
ylab="", xaxt="n")
trsup=max(v[ v <= (q[4]+1.5*DI) ])
y.trsup=c(trsup,trsup)
trinf=min(v[ v >= (q[2]-1.5*DI) ])
y.trinf=c(trinf,trinf)
y.med=c(q[3],q[3])
y.avg=c(mean(v),mean(v))
out= v[ (v > (q[4]+1.5*DI)) | (v < (q[2]-1.5*DI))]

lines(x2,y.trinf,lwd=2)
lines(x2,y.trsup,lwd=2)
lines(x,y.med,lwd=2)
lines(x,y.avg,lwd=2, lty=2, col="red")
lines(c(1,1),c(trinf,q[2]))
lines(c(1,1),c(q[4],trsup))
if (length(out)>0) points(1,out)

var.name=names(v)
if( is.null(var.name) ) var.name= "variable"
axis(1,at=1,labels=var.name)
}
```

```
> set.seed(32)
> vett = c(rnorm(100), rt(50,df=4))
> my.boxpl(vett)
```



Esercizio: (***concentrazione***) si implementi una funzione in grado di produrre, a partire da un vettore di realizzazioni di un carattere trasferibile, la curva di concentrazione di Lorenz ed il coefficiente di concentrazione di Gini (per richiami e approfondimenti cfr. Leti, 1983).

```
LorenzGini = function(m){
if (!is.vector(m)) stop("input parameter must be a vector")

lm=length(m)
m=sort(m)
M=cbind(1:lm,m)
Idown=matrix(0,ncol=lm,nrow=lm)
Idown[lower.tri(Idown,diag=T)]=1

# calcolo Qi
Qi=(Idown%*%m)/sum(m)
# calcolo Pi
Pi=(Idown%*%rep(1/lm,lm))

L=rbind(c(0,0),cbind(Pi,Qi))
plot(L, type="l", xlab="Pi", ylab="Qi", main="Lorenz curve")
polygon(L,col="red")

Gini.Index=sum(Pi-Qi)
maxGini.Index=(lm-1)/2
Gini.relIndex=Gini.Index/maxGini.Index
print(paste("Gini reltive index of concentration:",Gini.relIndex))
}

> vett=c(10,15,20,25,130)
> LorenzGini(vett)
[1] "Gini reltive index of concentration: 0.625"
```



Esercizio: (***modello di Ehrenfest***[22]) si implementi una funzione i cui parametri in ingresso sono due numeri interi indicanti rispettivamente il numero di valori '1' e di valori '2' da memorizzare in un vettore. Si scelga a caso un elemento del vettore e se ne cambi lo stato (il valore 1 diventa 2 e il valore 2 diventa 1). Verificare che continuando con le estrazioni e i conseguenti cambiamenti di stato, prima o poi il vettore finirà per contenere i valori 1 e 2 nella medesima proporzione, indipendentemente dalla loro composizione iniziale.

```
Ehrenfest = function(m1,m2,eps=0.000001){

urna=c(rep(1,m1),rep(2,m2))
perc.1=m1/(m1+m2)

c=0
while (abs(perc.1-0.5)>eps){

      i=sample(1:length(urna),1)
      urna[i]=3-urna[i]
      perc.1=sum(urna==1)/(m1+m2)
      c=c+1
      }

print (paste("numero di iterazioni: ",c))
}
```

```
> Ehrenfest(2,500)
[1] "numero di iterazioni:  879"
> Ehrenfest(2,5000)
[1] "numero di iterazioni:  7807"
```

[22] Questo modello fu proposto Paul Ehrenfest per spiegare il secondo principio della termodinamica.

Esercizio: si lanciano tre dadi regolari. In ciascun lancio si elimina il dado con il punteggio più alto (se più di un dado mostra il punteggio più alto se ne elimina uno a caso). Quindi:

a) se la somma dei dadi rimanenti è inferiore a 7 oppure se è 8, 9 o 10, non si vince niente;
b) se la somma dei dadi rimanenti è 7, si lancia una moneta regolare e solo se viene TESTA si vincono 50 euro;
c) se la somma dei dadi rimanenti è maggiore di 10, si vincono 100 euro.

Calcolare, tramite adeguato processo simulativo, il valore atteso del gioco (ovvero quanto il banco dovrebbe chiedere per accettare la partecipazione al gioco, affinché il gioco sia equo).

```
treDadi = function(nrepl=10000){

win=NULL
for (i in 1:nrepl){
       vincita=0
       dadi=sort(sample(1:6,3,replace=T))
       dadi=dadi[1:2]
       punteggio=sum(dadi)
       if (punteggio>10) vincita=100
       else if (punteggio==7 & runif(1)<.5) vincita=50
       win=c(win,vincita)
       }

return(mean(win))
}
```

```
> treDadi()
[1] 5.215

> treDadi(1e5)
[1] 5.0905
```

Esercizio: (**Craps**): Craps è un famoso gioco da tavolo americano proposto nei più rinomati casinò. Se ne vedano le regole sul sito del Casinò di Monte Carlo:
http://fr.casinomontecarlo.com/wp-content/uploads/2016/01/regole-del-gioco-craps.pdf

Nella versione semplificata, il gioco prevede il lancio di una coppia di dadi regolari di cui si sommano le facce. Il lancio è considerato:

a. vincente se il punteggio è pari a 7 o 11;
b. perdente se il risultato è un *craps (una schifezza)*, vale a dire se il punteggio è pari a 2, 3 o 12.
c. negli altri casi (4, 5, 6, 8, 9, 10), ai fini della vincita, l'esito diventa il punteggio che il giocatore, continuando a tirare i dadi, dovrà cercare di ottenere di nuovo prima di un 7.

Calcolare, tramite opportuno processo simulativo la probabilità di vittoria al gioco.

```
craps = function(nrepl=10000){

win=0
for(i in 1:nrepl){

      vincita=0
      dadi=sample(1:6,2,repl=T)
      punti=sum(dadi)

      if(sum(c(7,11)==punti)>0) vincita=1
      else if(sum(c(4:6,8:10)==punti)>0){
            punti.new=0
            while (punti.new!=7 & punti.new!=punti){
                  dadi.new=sample(1:6,2,repl=T)
                  punti.new=sum(dadi.new)
                  }
            if (punti.new==punti) vincita=1
            }
      win=win+vincita
      }
return(win/nrepl)
}

> craps()
[1] 0.4926

> craps(1e5)
[1] 0.49246
```

Come si può osservare, la probabilità di vincere a questo gioco è leggermente inferiore a 0,5. La quota che manca al raggiungimento del 50% è la percentuale di guadagno che il banco si assicura. Un po' come quando si decide di puntare sul Rosso o sul Nero sul tavolo della roulette. La percentuale di guadagno del banco è pari alla probabilità che esca il numero zero (di colore verde).

Esercizio: (**il *Solitario dell'un-due-tre***) si implementi una funzione in grado di calcolare, tramite adeguato processo simulativo, la probabilità di vincere al "*Solitario dell'un-due-tre"* giocando con un mazzo da 40 carte romagnole ben mescolate. Il solitario consiste nello scoprire le carte una alla volta mentre si dice a voce alta 1, 2, 3, 1, 2, 3, … (un numero per ogni carta scoperta in sequenza). Si vince se <u>nessuna</u> carta estratta coincide con il numero detto.

```
sol.123=function(nrepl=10000){

mazzo=rep(1:10,4)

win=0
for(i in 1:nrepl){
      mazzo=sample(mazzo)
      if (sum( mazzo==c(rep(1:3,13),1) )==0) win=win+1
      }

return(win/nrepl)
}

> sol.123()
[1] 0.008
> sol.123(1e5)
[1] 0.00825
```

Esercizio: (***"denari e monete" alla ricerca del poker d'assi***) si estraggano 5 carte da un mazzo di 40 carte romagnole ben mescolate. Per ogni "denari" estratta si ha diritto a lanciare una moneta bilanciata. Se il risultato del lancio evidenzierà una "testa" si avrà diritto ad estrarre una carta aggiuntiva. Si vince se tra le carte estratte ci sono 4 "Assi". Calcolare, tramite adeguato processo simulativo, la probabilità di vincere a questo gioco.

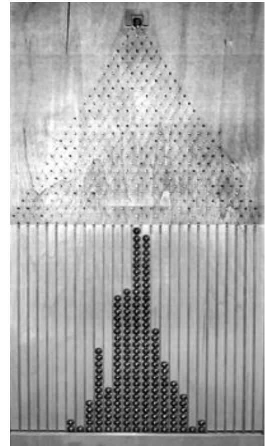
```
es3=function(nrepl=10000){

mazzo=sort(rep(1:10,4))
names(mazzo)=rep(c("C","D","F","P"),10)

win=0
for(i in 1:nrepl){
  indici=1:40
  ind.estr=sample(indici,5)
  indici=indici[-ind.estr]
  estr=mazzo[ind.estr]
  num.d=sum(names(estr)=="D")
  monete=rbinom(num.d,1,.5)
  if(sum(monete)>0){
                  new.ind.estr=sample(indici,sum(monete))
                  estr=c(estr, sample(mazzo,new.ind.estr))
                  }
  if((sum(estr==1)==4)) win=win+1
  }

return(win/nrepl)
}

> es3(1e6)
[1] 0.100928
```

Esercizio: (***primiera di sette***) si estraggano 10 carte da un mazzo di 40 carte romagnole ben mescolate. Per ogni “denari” estratta si ha diritto ad estrarre due carte aggiuntive, anche quando tra le carte estratte sono presenti delle “denari”. Si vince se tra tutte le carte estratte ci sono almeno tre 7. Calcolare, tramite adeguato processo simulativo, la probabilità di vincere a questo gioco.

```
primiera <- function(nrepl=10000){

win=0
for (i in 1:nrepl){
    mazzo=sort(sample(rep(1:10,4)))
    names(mazzo)=rep(c("D","C","B","S"),10)

    ind.estr=sample(1:40,10)
    estr=mazzo[ind.estr]
    n.den=sum(names(estr)=="D")
    mazzo.res=mazzo[-ind.estr]
    while(n.den>0){
       ind.estr=sample(1:length(mazzo.res),2*n.den)
       new.estr=mazzo.res[ind.estr]
       n.den=sum(names(new.estr)=="D")
       estr=c(estr,new.estr)
       mazzo.res=mazzo.res[-ind.estr]
       }
    win =win +(sum(estr==7)>2)
    }

return(win/nrepl)
}
```

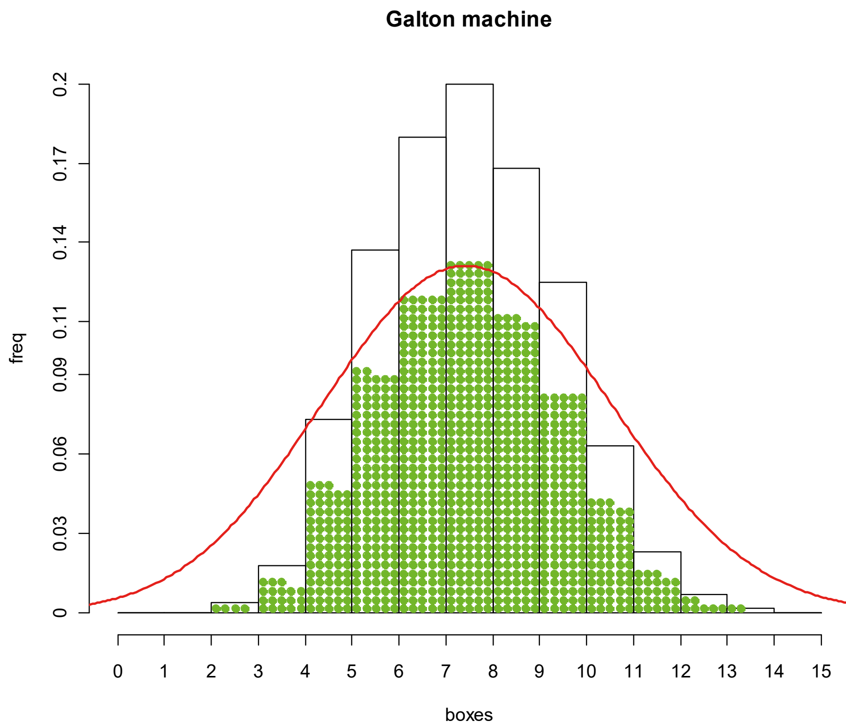
```
> primiera()
[1] 0.2879

> primiera(1e5)
[1] 0.28094
```

Esercizio: costruire una funzione che simuli il funzionamento della **macchina di Galton**. Tale macchina (o scatola) fu inventata da da Sir Francis Galton (1822 – 1911) per produrre una dimostrazione empirica del Teorema del Limite Centrale.



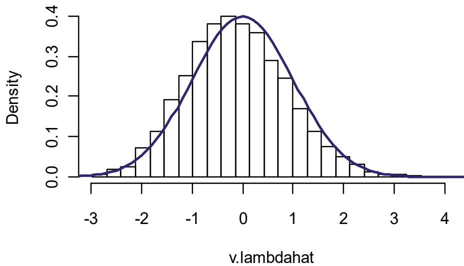
```
Galton = function(npalle, nbox){

if (nbox<2) stop("almeno due contenitori")
boxes=rep(0,nbox)
for (i in 1:npalle){
      spilli=rbinom(1,nbox-1,.5)+1
      boxes[spilli]=boxes[spilli]+1
      }
# calcola le frequenze relative
boxes.freq=boxes/npalle

plot.new()
plot.window( xlim=c(0,nbox), ylim=c(0,max(boxes.freq)))
axis(side=1,at=0:nbox,label=0:nbox)
axis(side=2,at=round(seq(0,max(boxes.freq),length=8),2)
            ,label=round(seq(0,max(boxes.freq),length=8),2))
title(main="Galton machine", xlab="boxes", ylab="freq")

# calcola lo step di salita delle palline nel box in proporzione a una
finestra di 600 pixel
ystep=max(boxes.freq)*10/600

# alloca le palline nei box in file di 5
for (i in 1:nbox){
      polygon(x=c(i-1,i,i,i-1), y=c(0,0,boxes.freq[i],boxes.freq[i]))
  if (boxes[i]>0){
      xx=i-1+0.1
      yy=ystep/2
      for (j in 1:boxes[i]){
          points(xx,yy,pch=19, col="green")
          xx=xx+0.2
          if(xx > i){
              xx=i-1+0.1
              yy=yy+ystep
              }
          }
      }
  }
d=density(1:nbox-0.5,weight=boxes.freq)
lines(d, col="red", lwd=2)
return(boxes)
}
```

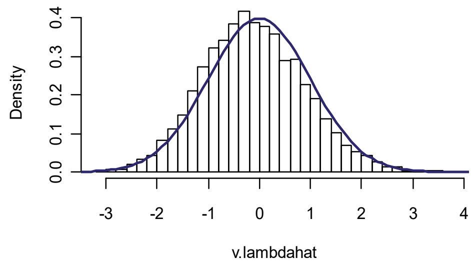
```
> galton(1000,15)
  [1]    0    0    4   18   73 137 180 200 168 125   63   23    7    2    0
```

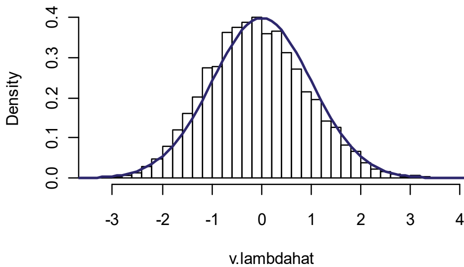


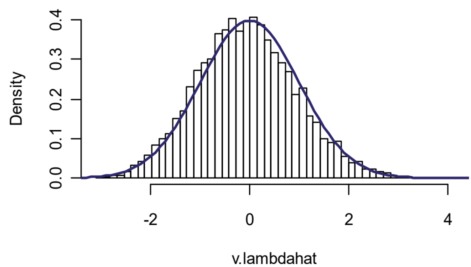
Esercizio: costruire una funzione per dimostrare empiricamente la tesi del **Teorema Limite Centrale**, nel caso di estrazioni da una popolazione caratterizzata da distribuzione di Poisson.

```
tlc2 = function(nrepl,n,l){

v.lambdahat=NULL
for(i in 1:nrepl){
  sam=rpois(n,lambda=l)
  lambdahat=mean(sam)
  v.lambdahat=c(v.lambdahat, (lambdahat-l)/sqrt(l/n) )
  }
br=sort(unique(v.lambdahat))
hist(v.lambdahat, breaks=br, prob=T, main="")
curve(dnorm(x),from=-5,to=5,add=T,lwd=2,col="navy")
title(paste("TLC da Poisson con lambda=",l))
}

> par(mfrow=c(2,2))
> tlc2(10000,25,.5)
> tlc2(10000,25,1)
> tlc2(10000,50,.5)
> tlc2(10000,50,1)
```









Esercizio: nel caso di un campionamento casuale semplice da una popolazione Normale a varianza non nota, verificare empiricamente che il livello di confidenza di un intervallo di stima per la media corrisponde effettivamente alla probabilità che l'intervallo estratto contenga la media.

```
confInt=function(media,sd.pop,alpha=0.05,dim.camp=100,num.int=10000){

int=NULL
for(i in 1:num.int){

      sam=rnorm(dim.camp, mean=media, sd=sd.pop)
      xmedio=mean(sam)
      s.camp=sd(sam)
      ME=qt(1-alpha/2,df=dim.camp-1)*s.camp/sqrt(dim.camp)

      int=rbind(int, c(xmedio-ME,xmedio+ME,
              (media>(xmedio-ME) & media<(xmedio+ME))*1))
      }
colnames(int)=c("L1","L2","chk")

return(int)
}
```

```
> a=confInt(3,2)
> head(a)
           L1       L2 chk
[1,] 2.379093 3.239344   1
[2,] 2.884979 3.678855   1
[3,] 2.251198 2.991450   0
[4,] 2.779672 3.563075   1
[5,] 2.812281 3.570642   1
[6,] 2.573958 3.437747   1
> colMeans(a)['chk']
   chk
0.9503
```

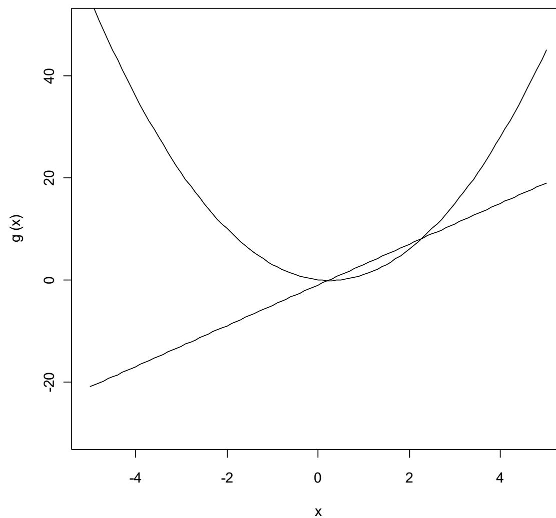
Esercizio: si implementi una generica funzione per il calcolo delle derivate per via numerica.

In analisi matematica, la derivata di una funzione (reale) $f(x)$ di una variabile reale nel punto x_0 è definita come il limite, al tendere a 0 dell'incremento h, del rapporto incrementale della funzione, sotto l'ipotesi che tale limite esista e sia finito.
Pertanto:

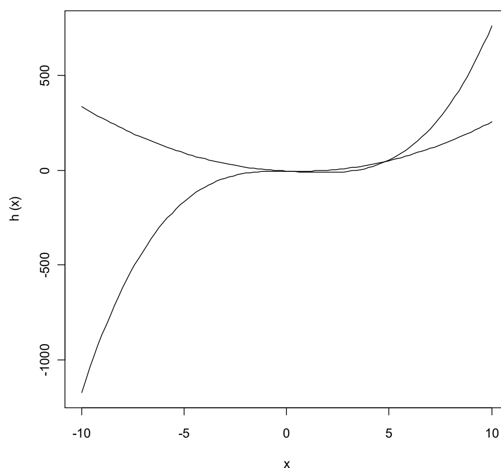
```
deriva <- function(f,x0,eps=1e-10){
limite= (f(x0+eps)-f(x0))/(eps)
return (limite)
}
```

Verifichiamo il funzionamento della funzione con qualche esempio:

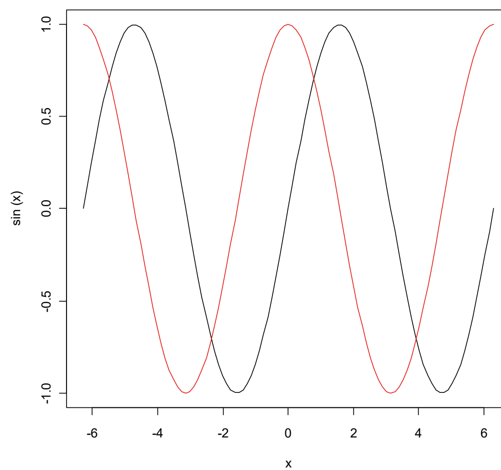
```
> g = function(x) 2*x^2-x

> g(.25)
[1] -0.125

> deriva(g,.25)
[1] 0

> curve(g,from=-5,to=5,ylim=c(-30,50))

> curve(deriva(g,x),from=-5,to=5,add=T)
```



```
> h=function(x) x^3-2*x^2-3*x-5
> curve(h,from=-10,to=10)
> curve(deriva(h,x), from=-10, to=10, add=T)
```




```
> curve(sin,from=-2*pi,to=2*pi)
> curve(deriva(sin,x), from=-2*pi, to=2*pi, add=T, col="red")
```



Esercizio: (***integrazione di Riemann***)
Si implementi una generica funzione per il calcolo degli integrali per via numerica.

In analisi matematica, l'integrale di una funzione (reale) $f(x)$ tra a e b può essere interpretato come l'area sottesa dalla funzione nello stesso intervallo.

Si consideri la partizione di un intervallo chiuso $[a,b]$ in n sottointervalli $[x_{i,}x_{i+1}]$ di uguale ampiezza e si consideri una funzione limitata $f(x)$ definita su $[a,b]$. L'integrale secondo Riemann di $f(x)$ nell'intervallo chiuso e limitato $[a,b]$ è definito come il limite per n che tende ad infinito della somma:

$$S_n = \frac{(b-a)}{n} \sum_{i=1}^{n-1} f(t_i)$$

dove:

- se $t_i = x_i$, S_n si dice somma sinistra di Riemann;
- se $t_i = x_{i+1}$, S_n si dice somma destra di Riemann;
- se $t_i = \left(x_i + x_{i+1}\right)/2$, S_n si dice somma media di Riemann.

```
Riemann=function(fun,a,b,num.int=10000,method="avg",pl=F){

if (sum(method == c("sx","dx","avg"))==0){
  warning(paste("method",method,"not supported; 'avg' will be used."))
  method="avg"
  }

step=(b-a)/num.int
x=seq(from=a, to=b, length=num.int+1)
if (method=="sx")
      y=fun(x[-length(x)])
      else if(method=="dx")
            y=fun(x[-1])
            else y=( fun(x[-1]) + fun(x[-length(x)]) )/2
aree=step*y

if(pl){
  plot.new()
  plot.window(xlim=c(a,b),ylim=c(0,max(y)))
  box()
  r=rainbow(num.int)
  for(i in 1:num.int)
      polygon(c(x[i:(i+1)],x[(i+1):i]), c(0,0, y[i], y[i]),col=r[i])
  curve(fun, from=a, to=b, col="red", lwd=2,add=T)
  title(main=paste("method '",method, "'"))
  }
sum(aree)
}
```

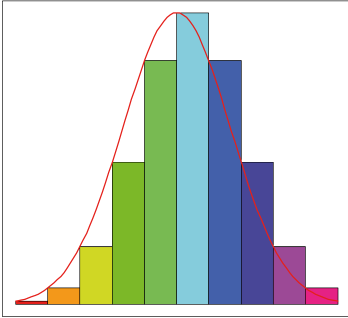
```
> Riemann(dnorm,-5,5)
[1] 0.9999994

> Riemann(dnorm, -qnorm(.975), qnorm(.975))
[1] 0.95

> Riemann(dnorm, -qnorm(.95), qnorm(.95),method="dx")
[1] 0.9
```

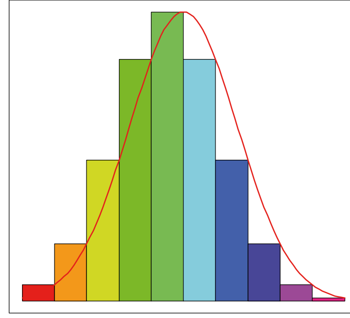
Riprendiamo ora la densità della Beta generalizzata illustrata alla fine del capitolo 19:

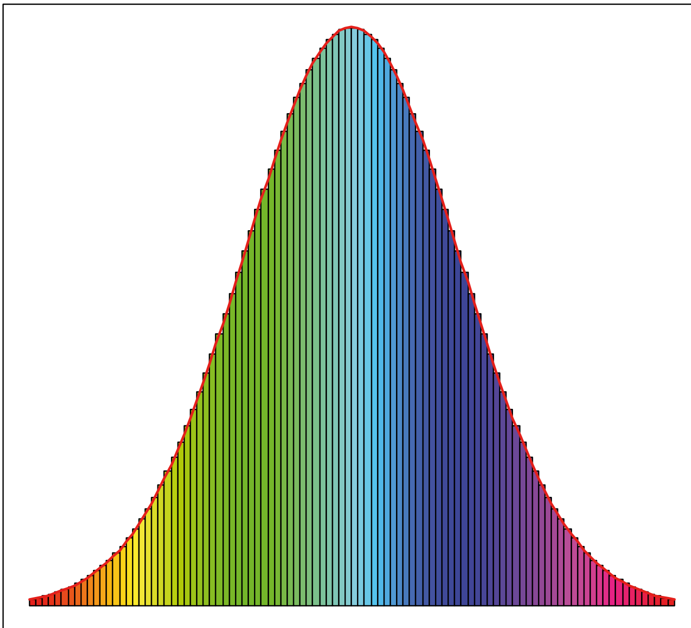
```
dbeta.gen <- function(x,a,b,l1,l2)
      return(1/beta(a,b) * ((x-l1)/(l2-l1))^(a-1)
                         * ((l2-x)/(l2-l1))^(b-1) * 1/(l2-l1))

f=function(x){
      dbeta.gen(x,a=3,b=2,l1=5,l2=20)
      }
> Riemann (f, 6, 8)
[1] 0.02607407

> Riemann (f, 6, 8, method="sx")
[1] 0.0260763
```

Le seguenti chiamate consentono di esaminare il risultato della funzione `rainbow()` (già utilizzata nel pie-chart illustrato nel capitolo 16) che restituisce il vettore dei codici esadecimali dei colori contigui, necessari a generare le sfumature proprie dell'arcobaleno nel numero richiesto dal parametro in ingresso.

```
> Riemann(dnorm,-3,3,num.int=10,method="sx",pl=T)
[1] 0.9965309

> Riemann(dnorm,-3,3,num.int=10,method="dx",pl=T)
[1] 0.9965309

> Riemann(dnorm,-3,3,num.int=100,pl=T)
[1] 0.9972922
```


method ' sx '


method ' dx '


method ' avg '

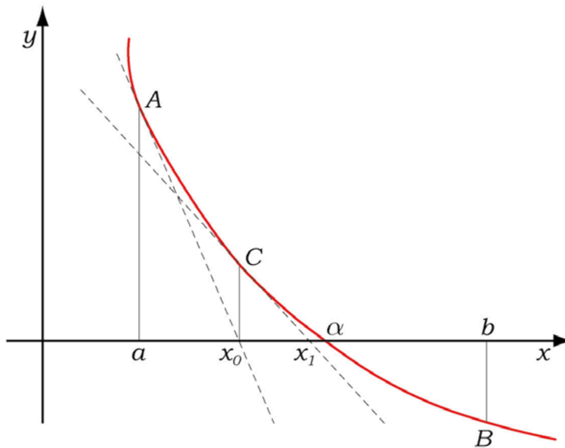
Esercizio: (***algoritmo di Newton - Raphson***) si implementi una funzione in grado di calcolare approssimativamente la soluzione di un'equazione della forma $f(x)=0$ in un intervallo $[a,b]$ in cui la funzione $f(x)$ è definita, assolutamente continua ed ha almeno una radice.

La funzione deve prevedere i seguenti passi:

a) si approssimi la funzione $f(x)$ con la tangente in uno degli estremi dell'intervallo $[a,b]$ e si assuma, come primo approssimativo valore soluzione dell'equazione $f(x)=0$, l'ascissa x_0 del punto in cui la tangente interseca l'asse delle x internamente all'intervallo $[a,b]$;

b) si ripeta il procedimento restringendosi all'intervallo $[x_0,b]$ al fine di individuare con il valore di ascissa x_1 il secondo approssimativo valore soluzione dell'equazione $f(x)=0$;

c) si iteri il procedimento fino ad ottenere una buona approssimazione del valore α reale soluzione dell'equazione $f(x)=0$.



Si osservi che la retta tangente alla funzione $f(x)$ nel punto A ha equazione:

$f(a)=\gamma+f'(a)\cdot a$ con $f'(x)$ derivata prima della funzione $f(x)$.

Pertanto $\gamma=f(a)-f'(a)\cdot a$.

L'equazione della retta tangente nel punto A quindi è: $y=f(a)-f'(a)\cdot a+f'(a)\cdot x$.

Ponendo $y=0$, si ricava il valore di x_0 pari a: $x_0=a-\dfrac{f(a)}{f'(a)}$.

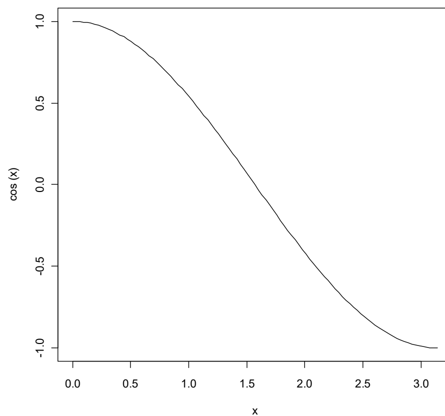
Ripetendo il ragionamento, il valore di x_1 risulterà pari a: $x_1=x_0-\dfrac{f(x_0)}{f'(x_0)}$.

Iterando il procedimento, per approssimazioni successive ci si avvicina al valore α. Il processo iterativo si dovrà interrompere quando l'approssimazione di tale valore sarà ritenuta soddisfacente.

```
deriva <- function(f,x0,eps=1e-10){
      return ( (f(x0+eps)-f(x0))/(eps) )
}


NR <- function(f,xstart,bounds,eps=1e-10){
x0=xstart
cnt=1
while(abs(f(x0))>eps){
        x0=x0-f(x0)/deriva(f,x0)

        if(x0<min(bounds))
                  x0=min(bounds)+eps
        if(x0>max(bounds))
                  x0=max(bounds)-eps

        print(paste("Iterazione",cnt,": x =",x0,", f =",f(x0)))
        cnt=cnt+1
        }
return(x0)
}
```

Verifichiamo il funzionamento della funzione con qualche esempio:

```
> curve(cos, from=0, to=pi)
```

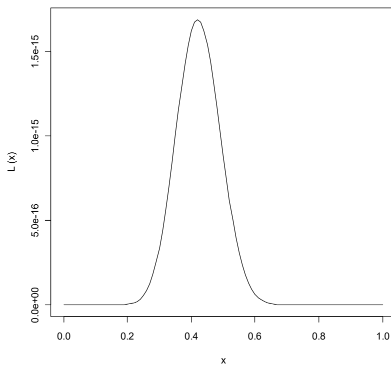


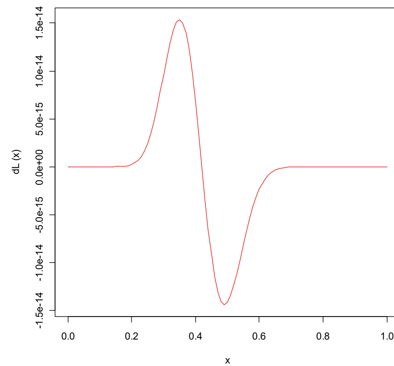

```
> NR(cos, .5, bounds=c(0,pi))
```

```
[1] "Iterazione 1 : x = 2.33048405675225 , f = -0.688695066952988"
[1] "Iterazione 2 : x = 1.38062681465759 , f = 0.189025353899075"
[1] "Iterazione 3 : x = 1.57312242528244 , f = -0.00232609638989258"
[1] "Iterazione 4 : x = 1.57079632279018 , f = 4.00471351097594e-09"
[1] "Iterazione 5 : x = 1.57079632679490 , f = 2.8327492261615e-16"
[1] 1.570796
```

Si supponga ora di lanciare 50 volte una moneta e di osservare 21 volte TESTA. Si vuol utilizzare l'algoritmo di NR per individuare il valore più probabile dell'evento TESTA.

```
# si scrive la funzione di verosimiglianza
L = function(p){
return ( (p)^x*(1-p)^(n-x) )
}
# si scrive la funzione derivata prima della verosimiglianza
dL = function(p){
return ( x*(p)^(x-1)*(1-p)^(n-x) - (p)^x*(n-x)*(1-p)^(n-x-1) )
}

> x=21
> n=50

> curve(L, from=0, to=1)
> x11()
> curve(dL, from=0, to=1,col="red")
```

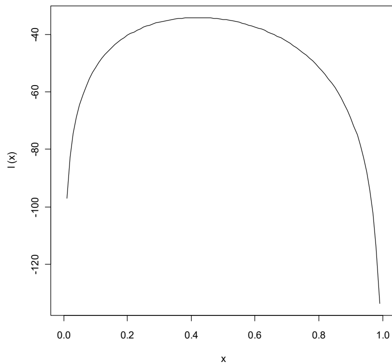


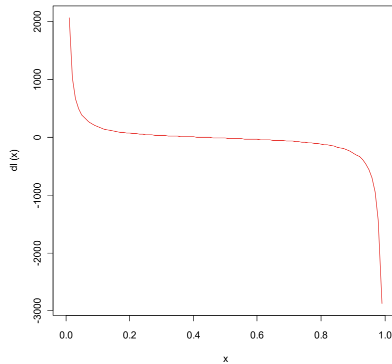



```
> NR(dL, .37, bounds=c(.3,.6), eps=.0000000000000000001)
[1] "Iterazione 1 : x = 0.46624270628071 , f = -1.26439072018149e-14"
[1] "Iterazione 2 : x = 0.383343470630840 , f = 1.13787909320257e-14"
[1] "Iterazione 3 : x = 0.432090893250659 , f = -4.09956688822050e-15"
[1] "Iterazione 4 : x = 0.41954223640583 , f = 1.58745897930128e-16"
[1] "Iterazione 5 : x = 0.419999881386301 , f = 4.11220272515272e-20"
[1] "Iterazione 6 : x = 0.419999999999989 , f = 3.96323718783036e-27"
[1] 0.42
```

Visti i valori molto piccoli assunti dalla funzione, conviene effettuare la trasformazione logaritmica della verosimiglianza, ottenendo la cosiddetta funzione di log-verosimiglianza:

```
l = function(p){
return ( log(p)*x+log(1-p)*(n-x) )
}

dl = function(p){
return ( x/p-(n-x)/(1-p) )
}
```

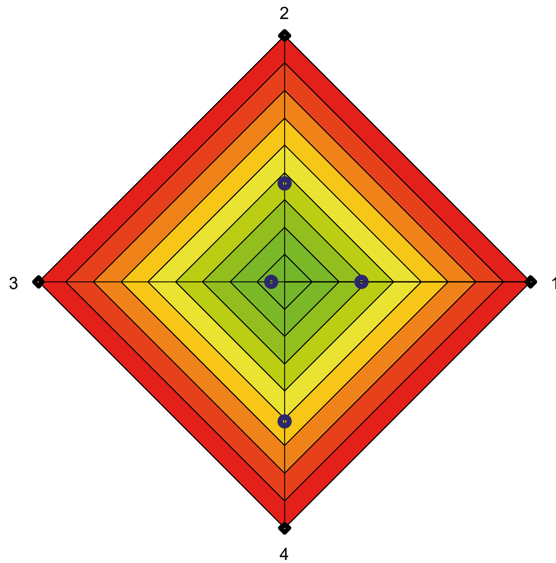




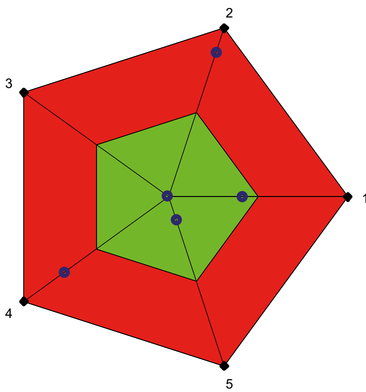

```
> NR(dl, 0.1, bounds=c(0,1))
[1] "Iterazione 1 : x = 0.183237003947457 , f = 79.099656156585"
[1] "Iterazione 2 : x = 0.301486427929206 , f = 28.1381459213474"
[1] "Iterazione 3 : x = 0.398356164494802 , f = 4.51536867223248"
[1] "Iterazione 4 : x = 0.419609809438925 , f = 0.0801089552197638"
[1] "Iterazione 5 : x = 0.419999899531795 , f = 2.06215542135624e-05"
[1] "Iterazione 6 : x = 0.419999999999989 , f = 2.20978790821391e-12"
[1] 0.42
```

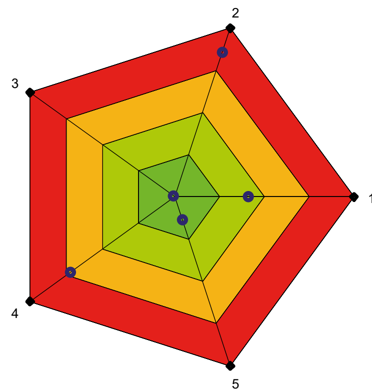
Esercizio: (*grafica avanzata*) si implementi una funzione in grado implementare un **grafico poligonale a bersaglio**, ovvero un grafico capace di illustrare, con unico colpo d'occhio, i punteggi ottenuti (su scala da 1 a 10) in relazione ad una certa batteria di item.

```
bersaglio <- function(nodi, set, val, bordi=T){

color=rainbow(set, start=0, end=.35)
xy.nodi=array(NA,dim=c(nodi,2,set))
xy.nodi.per=array(NA,dim=c(nodi+1,2,set+1))

par(mar=c(.5, .5, .5, .5))
plot.new()
plot.window(xlim=c(-1.1,1.1), ylim=c(-1.1,1.1))

for (j in 1:set){
  xy.text=NULL
  xy.shots=NULL
  xy.nodi.per[,,1]=matrix(0,nrow=nodi+1,ncol=2)
  for (i in 0:(nodi-1)){
     xy.nodi[i+1,,j]=c(cos(2*pi*i/nodi)*j/set, sin(2*pi*i/nodi)*j/set)
     if (j==set)
          xy.text=rbind(xy.text,c(1.1*cos(2*pi*i/nodi),1.1*sin(2*pi*i/n
          odi)))
     xy.shots=rbind(xy.shots, c(cos(2*pi*i/nodi)*(10-val[i+1])/9,
     sin(2*pi*i/nodi)*(10-val[i+1])/9))
     }
  xy.nodi.per[,,j+1]=rbind(xy.nodi[,,j],xy.nodi[1,,j])
  }

for (j in 1:set){
  polygon(c(xy.nodi.per[,1,j+1],xy.nodi.per[,1,j]),c(xy.nodi.per[,2,j+
       1],xy.nodi.per[,2,j]), col=color[set-j+1], border=bordi)
  if (bordi)
      lines(xy.coords(xy.nodi.per[,1,j+1],xy.nodi.per[,2,j+1]))
  if (j==set){
     for (i in 1:nodi)
         lines(c(0,xy.nodi[i,1,j]),c(0,xy.nodi[i,2,j]))
     points(xy.nodi[,,j], col = "black", lwd=5, pch=23)
     text(xy.text, paste("",1:nodi,sep=""),cex=1.2)
     points(xy.shots, col = "navy", lwd=10)
     }
  }
}


> settori=9 #su scala da 1 a 10, gli intervalli sono 9
> vertici=4
> shots=round(runif(vertici,min=1,max=10),1)
> bersaglio(vertici,settori,shots)
```

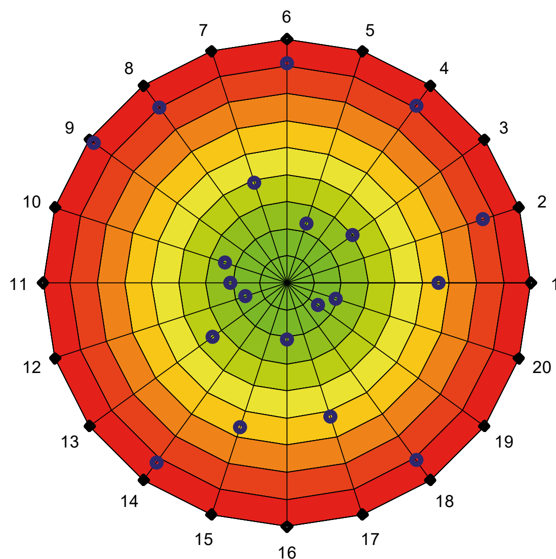



```
> par(mfrow=c(1,2))
> vertici=5
> settori=2
> shots=round(runif(vertici,min=1,max=10),1)
> shots
[1] 6.3 2.3 9.9 3.5 8.8
> bersaglio(vertici,settori,shots)
> settori=4
> bersaglio(vertici,settori,shots)
```

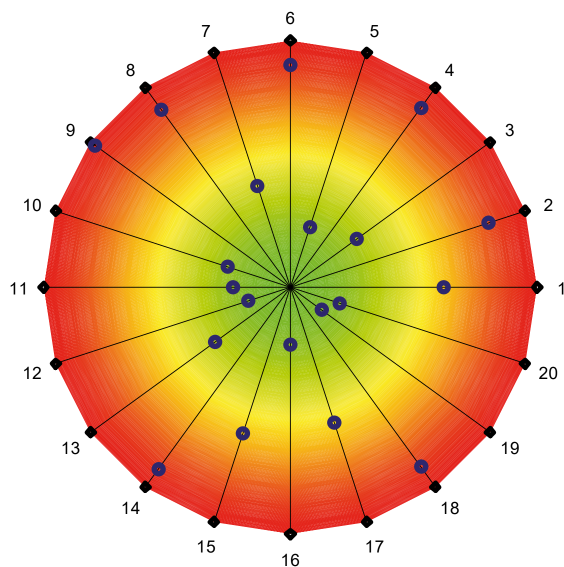




```
> vertici=20
> settori=9
> shots=round(runif(vertici,min=1,max=10),1)
> shots
[1] 4.4 2.4 7.0 1.9 7.7 1.9 6.1 2.0 1.2 7.6 7.9 8.4 6.6 1.8 4.4 7.9 4.8 1.9 8.6 8.1
> bersaglio(vertici,settori,shots)
```




```
> bersaglio(vertici,999,shots,bordi=F)
```



Esercizio: (“**gratta e vinci”**): si implementi una funzione in grado di simulare una serie di giocate successive alle lotterie istantanee stile Gratta e Vinci, ipotizzando una certa disponibilità economica del giocatore (borsellino iniziale) ed un’ipotetica soglia di uscita (ovvero l’importo del borsellino raggiunto il quale il giocatore smette di giocare).

Tutte le lotterie nazionali ad estrazione istantanea del tipo “gratta e vinci” sono indette tramite un decreto istitutivo dell’Agenzia delle Dogane e dei Monopoli che riporta: il prezzo del biglietto, la numerosità del lotto di biglietti, l’ammontare dei premi in relazione ai biglietti che verranno stampati, la distribuzione dei biglietti vincenti per ammontare del premio, le regole del gioco, le modalità di pagamento dei premi, ecc...

Per effettuare una simulazione realistica, consultiamo il decreto istitutivo de “Il Milionario”[23] il quale ci informa che il primo lotto è fissato nel numero complessivo di 108.000.000 biglietti e che la distribuzione dei biglietti vincenti è la seguente:

Importo vincita €	**biglietti vincenti**	**Probabilità di Vincita**
500.000	18	1 ogni 6.000.000,00 biglietti
300.000	18	1 ogni 6.000.000,00 biglietti
100.000	18	1 ogni 6.000.000,00 biglietti
10.000	324	1 ogni 333.333,33 biglietti
1.000	13.500	1 ogni 8.000,00 biglietti
500	36.000	1 ogni 3.000,00 biglietti
200	45.000	1 ogni 2400,00 biglietti
100	720.000	1 ogni 150,00 biglietti
50	900.000	1 ogni 120,00 biglietti
20	1.800.000	1 ogni 60,00 biglietti
15	3.060.000	1 ogni 35,29 biglietti
10	7.200.000	1 ogni 15,00 biglietti
5	13.140.000	1 ogni 8,22 biglietti
TOTALI	**26.914.878**	**1 biglietto ogni 4,013 è vincente**

Ciò che tendenzialmente cattura l’attenzione dei giocatori è il fatto che 1 biglietto ogni 4 risulti vincente. Si può però anche osservare che i biglietti da 500mila euro sono solo 18 e che sono solo le prime 4 righe della tabella precedente che contengono premi significativi. Inoltre, considerando che i biglietti di queste lotterie sono generalmente realizzati su un cartoncino lungo circa 15 cm, si può calcolare quanto sia lungo l’ipotetico rotolo di biglietti del primo lotto:

$$108 \cdot 10^6 \times 15 \text{ cm} = 16.200 \text{ Km}$$

[23] https://www.lottomaticaitalia.it/it/prodotti/gratta-e-vinci/classico/il-miliardario

16.200 Km sono una distanza ragguardevole, sono tanti quanti ne segnala Google MAPS volendo tentare di raggiungere Cape Town (Sud Africa) partendo da Murmansk (città russa sul Mare di Barents, all'interno del Circolo Polare Artico). Di tutti i biglietti della lotteria che, uno dopo l'altro, possiamo ipoteticamente immaginare di disporre sulla strada indicata da Google MAPS, solo 54 consentono di vincere un premio di almeno 100.000 euro. Volendo tentare di vincere almeno 10.000 euro, il numero dei biglietti vincenti sale a 378. Da questa prospettiva, la lotteria perde molto del suo interesse.

Vediamo se la simulazione conferma questa valutazione.

Volendo realizzare una funzione generalizzabile ad una qualsiasi lotteria istantanea, inseriamo in un file di testo, a due colonne (la prima indicante il valore del premio, la seconda indicante la frequenza dei biglietti emessi che danno diritto a ricevere quel premio), la struttura dei premi che provvederemo a leggere con la seguente istruzione:

```
tb=read.table("C:\\...\\tab_premi.csv",header=F,sep=";",dec=",")
```

Nel caso de "Il Milionario", il file di testo avrà questa struttura:

500000	18
300000	18
100000	18
10000	324
1000	13500
500	36000
200	45000
100	720000
50	900000
20	1800000
15	3060000
10	7200000
5	13140000

La seguente funzione prevede, come parametri in ingresso, oltre alla tavola della struttura dei premi, l'ammontare iniziale del borsellino (di default impostato a 10 euro), il costo di un singolo biglietto (impostato a 5 euro, in analogia con "Il Milionario"), la soglia di uscita (di default impostata a 100 euro, che come richiesto dall'esercizio è l'importo del borsellino raggiunto il quale il giocatore smette di giocare) ed il numero dei biglietti stampati (108e6).

La funzione è stata quindi costruita ipotizzando che un generico giocatore intenda giocarsi tutto il borsellino, a meno che il valore di questo (nel susseguirsi delle giocate) non raggiunga la soglia di uscita (100 euro quale valore di default).

```
gratta.vinci = function(borsellino=10, costo.bigl=5, uscita=100,
n.bigl, tab.premi, nrepl=1000){

bigl=rep(0,n.bigl)
premi=NULL
for(i in 1:dim(tab.premi)[1])
   premi=c(premi, rep(tab.premi[i,1],tab.premi[i,2]) )
bigl[1:length(premi)]=premi

b.in=borsellino
bigl.in=bigl
outcome=NULL
for(i in 1:nrepl){

       bigl=sample(bigl.in)
       borsellino=b.in
       nb=0
       while ((borsellino-costo.bigl)>=0 & borsellino<uscita){
              nb=nb+1
              borsellino=borsellino-costo.bigl
              borsellino=borsellino+bigl[1]
              bigl=bigl[-1]
              }

       outcome=rbind(outcome,c(nb,borsellino))
       }
colnames(outcome)=c("n.bigl","borsellino")
return(outcome)
}
```

Quanto si aspetta di vincere un giocatore con un borsellino iniziale pari a 10 €, che intende ritirarsi solo qualora il suo borsellino raggiunga i 100 euro? Grazie alle tecniche Monte Carlo possiamo valutare queste aspettative:

```
> ris100=gratta.vinci(n.bigl=108000000,uscita=100, tab.premi=tb)
> head(ris100)
     n.bigl borsellino
[1,]      4        -10
[2,]      3        -10
[3,]     15        100
[4,]      6        -10
[5,]      2        -10
[6,]      7        -10

> apply(ris100,2,mean)
    n.bigl borsellino
     3.827      5.295
```

Con un borsellino iniziale di 10 €, la vincita attesa (calcolata su 1000 replicazioni) è pari a 5,295 € (sarebbe pertanto meglio parlare di perdita attesa), con un numero medio di biglietti acquistati circa pari a 4. Nel dettaglio:

```
> table(ris100[,'borsellino'])

  0 100 105 115 120 125 145 165 205 225 500 505
965  10  11   2   1   2   3   1   1   1   1   2
```

965 scommettitori su 1000 perderebbero tutto l'intero borsellino, altri 30 riuscirebbero a vincere un premio tra i 100 e i 200 €. Solo 5 supererebbero i 200 € di vincita. L'ammontare dei premi pagati è pari a:

```
> sum(ris100[,'borsellino'])
[1] 5295
```

I risultati precedenti sono stati ottenuti immaginando una soglia di uscita pari a 100 euro. Cosa accade se abbassassimo tale soglia di uscita? Proviamo a simulare il comportamento di uno scommettitore che si avvia a giocare con un borsellino iniziale pari a 10 € ma che intende ritirarsi al raggiungimento dei 50 €:

```
> apply(ris50,2,mean)
    n.bigl borsellino
     3.214      4.825

> table(ris50[,'borsellino'])

   0   50   55   60   65  100  105  120  205 1005
 947   12   18    3    1    7    8    2    1    1

> sum(ris50[,'borsellino'])
[1] 4825
```

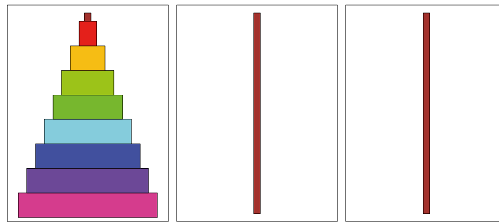
Adesso gli scommettitori che perderebbero l'intero borsellino sono 947, per cui è leggermente maggiore la quota di coloro che interromperebbe il gioco con una vincita almeno uguale alla soglia di uscita desiderata (50 €). Com'era logico attendersi, l'ammontare dei premi pagati è inferiore al caso in cui la soglia di uscita era impostata a 100 €.

Una importante riflessione a margine di questa simulazione: l'algoritmo è stata strutturato immaginando che il giocatore, ad ogni replicazione, stia fronteggiando la lotteria per "primo", ovvero che abbia la possibilità di estrarre a sorte un biglietto dall'intero lotto dei biglietti messi in commercio. Questo in effetti è ciò che succede nella testa di un qualsiasi individuo che decide di acquistare un biglietto di una lotteria istantanea: non sa quali biglietti siano già usciti, l'unica informazione che ha è la tavola dei premi, per cui decide di giocare ipotizzando di avere a disposizione l'intero monte premi. Ma ovviamente questo non è mai

vero. Le giocate successive alterano le probabilità di vincita (sono estrazioni senza reimmissione da una popolazione finita) e potrebbero rendere nel lungo andare (data la dimensione dei lotti generalmente stampati) più o meno allettante il gioco. Per questo motivo le vincite che si susseguono nel tempo non vengono mai comunicate.

Esercizio: (“**la Torre di Hanoi**”) si implementi una funzione in grado di risolvere il famoso rompicapo ludico-matematico (inventato nell'Ottocento dal matematico Edouard Lucas), visualizzandone graficamente tutte le mosse.

La più frequente versione del gioco è formata da otto dischi, di dimensioni decrescenti e con un foro al centro affinché possano essere infilati in una colonnina in maniera sovrapposta a formare un cono (torre) rovesciato. I dischi devono poi essere spostati, uno alla volta, su una delle altre due colonnine libere con la seguente regola: è proibito collocare un disco più grande su uno di raggio inferiore.



Il gioco è anche noto con il nome di “Torre di Brahma”: si narra infatti (secondo una leggenda pare inventata per pubblicizzare il gioco stesso) che in un tempio Indù, i monaci siano, ogni giorno, costantemente impegnati a spostare 64 dischi d’oro di raggio decrescente su tre colonne di diamante, seguendo le regole della Torre di Hanoi. La leggenda afferma che quando i monaci avranno finito di spostare i dischi, ricreando il cono sulla terza colonnina, il Mondo finirà.

Il gioco è estremamente interessante dl punto di vista matematico-induttivo:

- se i dischi fossero due, il gioco è banale e si risolve in tre mosse;
- se i dischi fossero tre, il gioco si risolve in sette mosse: il disco più piccolo sulla terza colonna, il disco intermedio sulla seconda colonna; il disco più piccolo dalla terza alla seconda colonna; questo libera la terza colonna per il disco più grande; quindi il disco più piccolo sulla prima colonna; il disco intermedio sulla terza ed infine il disco più piccolo dalla prima alla terza colonna;
- per spostare n dischi:
  a) occorre spostarne $(n-1)$ sulla seconda colonna;
  b) quindi occorre spostare il disco più grande sulla terza colonna;
  c) e riportare gli $(n-1)$ dischi dalla seconda alla terza colonna.

  Però le regole vietano spostamenti simultanei. Ma spostare $(n-1)$ dischi è un’operazione complessa che si risolve nel modo appena descritto, provvedendo allo spostamento di $(n-2)$ dischi e così via, fino allo spostamento di un solo disco come richiesto dalle regole del gioco.

Quello appena descritto è un algoritmo ricorsivo di complessità esponenziale: il numero minimo di mosse necessarie per ultimare il gioco è $2^n - 1$, dove n è il numero dei dischi presenti all’inizio nella prima colonnina. 2 dischi richiedono almeno 3 mosse, 3 dischi ne richiedono almeno 7, 64 dischi (quelli che muovono i monaci del tempio dedicato a Brahma)

ne richiedono 18.446.744.073.709.551.615. Se occorresse anche un solo secondo per compiere ogni singola mossa, sarebbero necessari oltre 5.849.424.173 secoli per ultimare il gioco. Sebbene nel calcolo non si sia tenuto conto degli anni bisestili, questo ci tranquillizza un po' sulla "prossimità" della fine del Mondo.

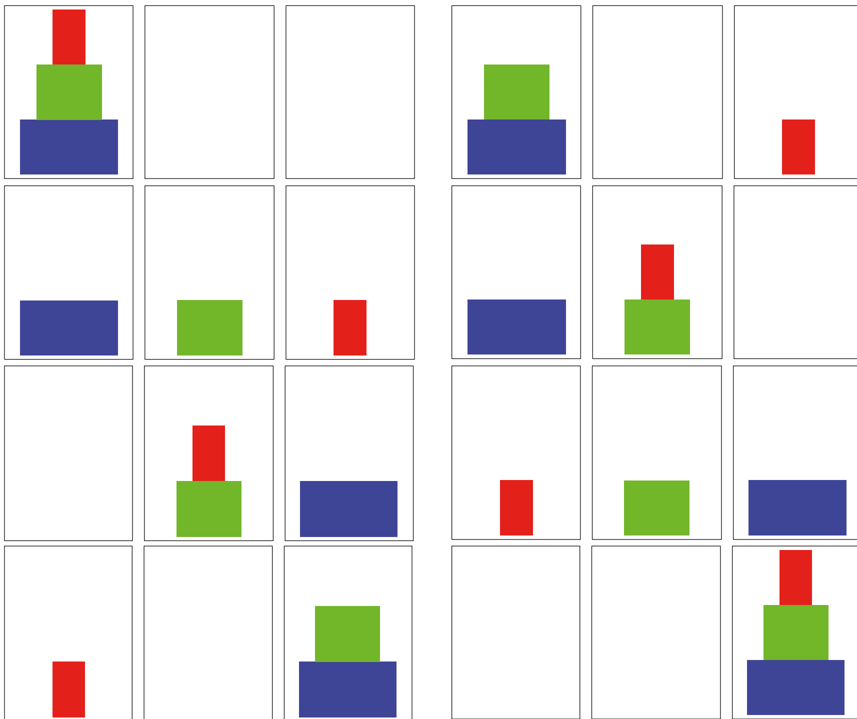
La seguente funzione prevede, come parametri in ingresso, solo il numero dei dischi da utilizzare (per contenere il tempo di esecuzione è stato previsto un massimo di 10 dischi, ma questa configurazione può essere modificata in maniera estremamente semplice) ed il numero di secondi di attesa tra una mossa e l'altra. Nella soluzione proposta sono state utilizzate alcune funzioni che non sono mai state discusse in precedenza (`split.screen()`, `screen()`, `Sys.sleep()`). Il loro funzionamento è estremamente semplice ed intuitivo per cui si invita il lettore a consultare il manuale di **R** per eventuali approfondimenti.

```
Hanoi <- function(nd, sec=1){

if(nd>10){
    warning("previsti massimo 10 dischi: ne verranno utilizzati 10.")
    nd=10
    }
dischi <<- cbind(1:nd,rep(1,length(nd)))
col.d <<- rainbow(nd)

# configurazione iniziale
plot.new()
split.screen(figs = c( 1, 3 ))
for(i in 1:3){
      screen(i)
      par(mar=c(.5, .5, .5, .5))
      plot.window(xlim=c(0-.5,nd+.5), ylim=c(0,nd))
      box()
      }
screen(1)
for(i in nd:1){
     d=dischi[i,1]
     liv.d=nd-which(dischi[,1]==i)+1
     x.p=c(nd/2-d/2,nd/2-d/2,nd/2+d/2,nd/2+d/2)
     y.p=c(liv.d-1,liv.d,liv.d,liv.d-1)
     polygon(x.p,y.p,col=col.d[i],border=NA)
     }

scambio <- function(n,from,to,aux){

        if(n>1) scambio(n-1,from,aux,to)

        # disco più piccolo sulla torre from e suo livello
        min.from=min(dischi[dischi[,2]==from,1])
        liv.from=sum(dischi[,2]==from)-
                       which(dischi[dischi[,2]==from,1]==min.from)+1
        # modifico la torre from
        screen(from)
        x.p=c(0,0,nd,nd)
        y.p=c(liv.from-1,liv.from,liv.from,liv.from-1)
        polygon(x.p,y.p,col="white",border=NA)
```

```
        # sposto il disco sulla torre to
        dischi[min.from,2] <<- to
        min.to=min(dischi[dischi[,2]==to,1])
        liv.to=sum(dischi[,2]==to)-
                        which(dischi[dischi[,2]==to,1]==min.to)+1

        # modifico la torre to
        screen(to)
        x.p=c(nd/2-min.to/2,nd/2-min.to/2,nd/2+min.to/2,nd/2+min.to/2)
        y.p=c(liv.to-1,liv.to,liv.to,liv.to-1)
        polygon(x.p,y.p,col=col.d[min.to],border=NA)

        Sys.sleep(sec)

        if (n>1) scambio(n-1,aux,to,from)
        }

Sys.sleep(2)
scambio(nd,1,3,2)
}
```

Richiamando la funzione con 3 dischi (e 2 secondi di attesa tra ogni mossa):

```
> Hanoi(3,sec=2)
```



Buon divertimento.

Bibliografia essenziale

Albert J.H., Gentle J.E. (2004). "Special Section: Teaching Computational Statistics", *The American Statistician*, 58: 1–1, doi:10.1198/0003130042872

Braun W.J., Murdoch D.J. (2007). A First Course in Statistical Programming with R. Cambridge University Press, ISBN: 978-0-521-87265-2

Burattini E., Chianese A., Picariello A., Moscato V., Sansone C. (2016). *Che C serve? - Per iniziare a programmare*. Maggioli, ISBN 9788891611734

Espa G., Micciolo R. (2008). *Problemi ed esperimenti di statistica con R*. Apogeo, ISBN: 9788850327379

Kabacoff R.I. (2011). *R in action* (*data analysis and graphics with R)*. Manning, ISBN: 9781935182399

Knuth, D.E. (2002). *The Art of Computer Programming*. Volume 2, (3rd edition, ninth printing). Addison-Wesley, Reading (Ma)

Lauro C. (1996). "Computational statistics or statistical computing, is that the question?", *Computational Statistics & Data Analysis*, 23 (1): 191–193, doi:10.1016/0167-9473(96)88920-1

Leti G. (1983). *Statistica Descrittiva.* Il Mulino, Bologna, ISBN: 9788815002785

Muggeo V.M.R., Ferrara G. (2005). *Il Linguaggio R: concetti introduttivi ed esempi* (II edizione). ftp://cran.r-project.org/pub/R/doc/contrib/nozioniR.pdf

Sturges, H.A. (1926). The choice of a class interval. *Journal of the American Statistical Association* 21, 65–66

B. Bertaccini, *Introduzione alla Statistica Computazionale con R*, ISBN (print) 978-88-6453-673-6, ISBN (online) 978-88-6453-674-3, CC BY 4.0, 2018 Firenze University Press.

Venables W.N., Smith D.M. and the R Development Core Team (2016). *An introduction to R*. https://cran.r-project.org/doc/manuals/R-intro.pdf

Wilkinson L. (2008). "The Future of Statistical Computing (with discussion)", *Technometrics*, 50 (4): 418–435, doi:10.1198/004017008000000460

STRUMENTI
PER LA DIDATTICA E LA RICERCA

1. Brunetto Chiarelli, Renzo Bigazzi, Luca Sineo (a cura di), *Alia: Antropologia di una comunità dell'entroterra siciliano*
2. Vincenzo Cavaliere, Dario Rosini, *Da amministratore a manager. Il dirigente pubblico nella gestione del personale: esperienze a confronto*
3. Carlo Biagini, *Information technology ed automazione del progetto*
4. Cosimo Chiarelli, Walter Pasini (a cura di), *Paolo Mantegazza. Medico, antropologo, viaggiatore*
5. Luca Solari, *Topics in Fluvial and Lagoon Morphodynamics*
6. Salvatore Cesario, Chiara Fredianelli, Alessandro Remorini, *Un pacchetto evidence based di tecniche cognitivo-comportamentali sui generis*
7. Marco Masseti, *Uomini e (non solo) topi. Gli animali domestici e la fauna antropocora*
8. Simone Margherini (a cura di), *BIL Bibliografia Informatizzata Leopardiana 1815-1999: manuale d'uso ver. 1.0*
9. Paolo Puma, *Disegno dell'architettura. Appunti per la didattica*
10. Antonio Calvani (a cura di), *Innovazione tecnologica e cambiamento dell'università. Verso l'università virtuale*
11. Leonardo Casini, Enrico Marone, Silvio Menghini, *La riforma della Politica Agricola Comunitaria e la filiera olivicolo-olearia italiana*
12. Salvatore Cesario, *L'ultima a dover morire è la speranza. Tentativi di narrativa autobiografica e di "autobiografia assistita"*
13. Alessandro Bertirotti, *L'uomo, il suono e la musica*
14. Maria Antonietta Rovida, *Palazzi senesi tra '600 e '700. Modelli abitativi e architettura tra tradizione e innovazione*
15. Simone Guercini, Roberto Piovan, *Schemi di negoziato e tecniche di comunicazione per il tessile e abbigliamento*
16. Antonio Calvani, *Technological innovation and change in the university. Moving towards the Virtual University*
17. Paolo Emilio Pecorella, *Tell Barri/Kahat: la campagna del 2000. Relazione preliminare*
18. Marta Chevanne, *Appunti di Patologia Generale. Corso di laurea in Tecniche di Radiologia Medica per Immagini e Radioterapia*
19. Paolo Ventura, *Città e stazione ferroviaria*
20. Nicola Spinosi, *Critica sociale e individuazione*
21. Roberto Ventura (a cura di), *Dalla misurazione dei servizi alla customer satisfaction*
22. Dimitra Babalis (a cura di), *Ecological Design for an Effective Urban Regeneration*
23. Massimo Papini, Debora Tringali (a cura di), *Il pupazzo di garza. L'esperienza della malattia potenzialmente mortale nei bambini e negli adolescenti*
24. Manlio Marchetta, *La progettazione della città portuale. Sperimentazioni didattiche per una nuova Livorno*
25. Fabrizio F.V. Arrigoni, *Note su progetto e metropoli*
26. Leonardo Casini, Enrico Marone, Silvio Menghini, *OCM seminativi: tendenze evolutive e assetto territoriale*
27. Pecorella Paolo Emilio, Raffaella Pierobon Benoit, *Tell Barri/Kahat: la campagna del 2001. Relazione preliminare*
28. Nicola Spinosi, *Wir Kinder. La questione del potere nelle relazioni adulti/bambini*
29. Stefano Cordero di Montezemolo, *I profili finanziari delle società vinicole*
30. Luca Bagnoli, Maurizio Catalano, *Il bilancio sociale degli enti non profit: esperienze toscane*
31. Elena Rotelli, *Il capitolo della cattedrale di Firenze dalle origini al XV secolo*
32. Leonardo Trisciuzzi, Barbara Sandrucci, Tamara Zappaterra, *Il recupero del sé attraverso l'autobiografia*
33. Nicola Spinosi, *Invito alla psicologia sociale*
34. Raffaele Moschillo, *Laboratorio di disegno. Esercitazioni guidate al disegno di arredo*
35. Niccolò Bellanca, *Le emergenze umanitarie complesse. Un'introduzione*
36. Giovanni Allegretti, *Porto Alegre una biografia territoriale. Ricercando la qualità urbana a partire dal patrimonio sociale*
37. Riccardo Passeri, Leonardo Quagliotti, Christian Simoni, *Procedure concorsuali e governo dell'impresa artigiana in Toscana*
38. Nicola Spinosi, *Un soffitto viola. Psicote-*

rapia, formazione, autobiografia
39. Tommaso Urso, *Una biblioteca in divenire. La biblioteca della Facoltà di Lettere dalla penna all'elaboratore. Seconda edizione rivista e accresciuta*
40. Paolo Emilio Pecorella, Raffaella Pierobon Benoit, *Tell Barri/Kahat: la campagna del 2002. Relazione preliminare*
41. Antonio Pellicanò, *Da Galileo Galilei a Cosimo Noferi: verso una nuova scienza. Un inedito trattato galileiano di architettura nella Firenze del 1650*
42. Aldo Burresi (a cura di), *Il marketing della moda. Temi emergenti nel tessile-abbigliamento*
43. Curzio Cipriani, *Appunti di museologia naturalistica*
44. Fabrizio F.V. Arrigoni, *Incipit. Esercizi di composizione architettonica*
45. Roberta Gentile, Stefano Mancuso, Silvia Martelli, Simona Rizzitelli, *Il Giardino di Villa Corsini a Mezzomonte. Descrizione dello stato di fatto e proposta di restauro conservativo*
46. Arnaldo Nesti, Alba Scarpellini (a cura di), *Mondo democristiano, mondo cattolico nel secondo Novecento italiano*
47. Stefano Alessandri, *Sintesi e discussioni su temi di chimica generale*
48. Gianni Galeota (a cura di), *Traslocare, riaggregare, rifondare. Il caso della Biblioteca di Scienze Sociali dell'Università di Firenze*
49. Gianni Cavallina, *Nuove città antichi segni. Tre esperienze didattiche*
50. Bruno Zanoni, *Tecnologia alimentare 1. La classe delle operazioni unitarie di disidratazione per la conservazione dei prodotti alimentari*
51. Gianfranco Martiello, *La tutela penale del capitale sociale nelle società per azioni*
52. Salvatore Cingari (a cura di), *Cultura democratica e istituzioni rappresentative. Due esempi a confronto: Italia e Romania*
53. Laura Leonardi (a cura di), *Il distretto delle donne*
54. Cristina Delogu (a cura di), *Tecnologia per il web learning. Realtà e scenari*
55. Luca Bagnoli (a cura di), *La lettura dei bilanci delle Organizzazioni di Volontariato toscane nel biennio 2004-2005*
56. Lorenzo Grifone Baglioni (a cura di), *Una generazione che cambia. Civismo, solidarietà e nuove incertezze dei giovani della provincia di Firenze*
57. Monica Bolognesi, Laura Donati, Gabriella Granatiero, *Acque e territorio. Progetti e regole per la qualità dell'abitare*
58. Carlo Natali, Daniela Poli (a cura di), *Città e territori da vivere oggi e domani. Il contributo scientifico delle tesi di laurea*
59. Riccardo Passeri, *Valutazioni imprenditoriali per la successione nell'impresa familiare*
60. Brunetto Chiarelli, Alberto Simonetta, *Storia dei musei naturalistici fiorentini*
61. Gianfranco Bettin Lattes, Marco Bontempi (a cura di), *Generazione Erasmus? L'identità europea tra vissuto e istituzioni*
62. Paolo Emilio Pecorella, Raffaella Pierobon Benoit, *Tell Barri / Kahat. La campagna del 2003*
63. Fabrizio F.V. Arrigoni, *Il cervello delle passioni. Dieci tesi di Adolfo Natalini*
64. Saverio Pisaniello, *Esistenza minima. Stanze, spazî della mente, reliquiario*
65. Maria Antonietta Rovida (a cura di), *Fonti per la storia dell'architettura, della città, del territorio*
66. Ornella De Zordo, *Saggi di anglistica e americanistica. Temi e prospettive di ricerca*
67. Chiara Favilli, Maria Paola Monaco, *Materiali per lo studio del diritto antidiscriminatorio*
68. Paolo Emilio Pecorella, Raffaella Pierobon Benoit, *Tell Barri / Kahat. La campagna del 2004*
69. Emanuela Caldognetto Magno, Federica Cavicchio, *Aspetti emotivi e relazionali nell'e-learning*
70. Marco Masseti, *Uomini e (non solo) topi* (2ª edizione)
71. Giovanni Nerli, Marco Pierini, *Costruzione di macchine*
72. Lorenzo Viviani, *L'Europa dei partiti. Per una sociologia dei partiti politici nel processo di integrazione europea*
73 Teresa Crespellani, *Terremoto e ricerca. Un percorso scientifico condiviso per la caratterizzazione del comportamento sismico di alcuni depositi italiani*
74 Fabrizio F.V. Arrigoni, *Cava. Architettura in "ars marmoris"*
75. Ernesto Tavoletti, *Higher Education and Local Economic Development*
76. Carmelo Calabrò, *Liberalismo, democrazia, socialismo. L'itinerario di Carlo Rosselli (1917-1930)*
77. Luca Bagnoli, Massimo Cini (a cura di),

La cooperazione sociale nell'area metropolitana fiorentina. Una lettura dei bilanci d'esercizio delle cooperative sociali di Firenze, Pistoia e Prato nel quadriennio 2004-2007

78. Lamberto Ippolito, *La villa del Novecento*
79. Cosimo Di Bari, *A passo di critica. Il modello di* Media Education *nell'opera di Umberto Eco*
80. Leonardo Chiesi (a cura di), *Identità sociale e territorio. Il Montalbano*
81. Piero Degl'Innocenti, *Cinquant'anni, cento chiese. L'edilizia di culto nelle diocesi di Firenze, Prato e Fiesole (1946-2000)*
82. Giancarlo Paba, Anna Lisa Pecoriello, Camilla Perrone, Francesca Rispoli, *Partecipazione in Toscana: interpretazioni e racconti*
83. Alberto Magnaghi, Sara Giacomozzi (a cura di), *Un fiume per il territorio. Indirizzi progettuali per il parco fluviale del Valdarno empolese*
84. Dino Costantini (a cura di), *Multiculturalismo alla francese?*
85. Alessandro Viviani (a cura di), *Firms and System Competitiveness in Italy*
86. Paolo Fabiani, *The Philosophy of the Imagination in Vico and Malebranche*
87. Carmelo Calabrò, *Liberalismo, democrazia, socialismo. L'itinerario di Carlo Rosselli*
88. David Fanfani (a cura di), *Pianificare tra città e campagna. Scenari, attori e progetti di nuova ruralità per il territorio di Prato*
89. Massimo Papini (a cura di), *L'ultima cura. I vissuti degli operatori in due reparti di oncologia pediatrica*
90. Raffaella Cerica, *Cultura Organizzativa e Performance economico-finanziarie*
91. Alessandra Lorini, Duccio Basosi (a cura di), *Cuba in the World, the World in Cuba*
92. Marco Goldoni, *La dottrina costituzionale di Sieyès*
93. Francesca Di Donato, *La scienza e la rete. L'uso pubblico della ragione nell'età del Web*
94. Serena Vicari Haddock, Marianna D'Ovidio, *Brand-building: the creative city. A critical look at current concepts and practices*
95. Ornella De Zordo (a cura di), *Saggi di Anglistica e Americanistica. Ricerche in corso*
96. Massimo Moneglia, Alessandro Panunzi (edited by), *Bootstrapping Information from Corpora in a Cross-Linguistic Perspective*
97. Alessandro Panunzi, *La variazione semantica del verbo* essere *nell'Italiano parlato*
98. Matteo Gerlini, *Sansone e la Guerra fredda. La capacità nucleare israeliana fra le due superpotenze (1953-1963)*
99. Luca Raffini, *La democrazia in mutamento: dallo Stato-nazione all'Europa*
100. Gianfranco Bandini (a cura di), *noi-loro. Storia e attualità della relazione educativa fra adulti e bambini*
101. Anna Taglioli, *Il mondo degli altri. Territori e orizzonti sociologici del cosmopolitismo*
102. Gianni Angelucci, Luisa Vierucci (a cura di), *Il diritto internazionale umanitario e la guerra aerea. Scritti scelti*
103. Giulia Mascagni, *Salute e disuguaglianze in Europa*
104. Elisabetta Cioni, Alberto Marinelli (a cura di), *Le reti della comunicazione politica. Tra televisioni e social network*
105. Cosimo Chiarelli, Walter Pasini (a cura di), *Paolo Mantegazza e l'Evoluzionismo in Italia*
106. Andrea Simoncini (a cura di), *La semplificazione in Toscana. La legge n. 40 del 2009*
107. Claudio Borri, Claudio Mannini (edited by), *Aeroelastic phenomena and pedestrian-structure dynamic interaction on non-conventional bridges and footbridges*
108. Emiliano Scampoli, *Firenze, archeologia di una città (secoli I a.C. – XIII d.C.)*
109. Emanuela Cresti, Iørn Korzen (a cura di), *Language, Cognition and Identity. Extensions of the endocentric/exocentric language typology*
110. Alberto Parola, Maria Ranieri, *Media Education in Action. A Research Study in Six European Countries*
111. Lorenzo Grifone Baglioni (a cura di), *Scegliere di partecipare. L'impegno dei giovani della provincia di Firenze nelle arene deliberative e nei partiti*
112. Alfonso Lagi, Ranuccio Nuti, Stefano Taddei, *Raccontaci l'ipertensione. Indagine a distanza in Toscana*
113. Lorenzo De Sio, *I partiti cambiano, i valori restano? Una ricerca quantitativa e qualitativa sulla cultura politica in Toscana*
114. Anna Romiti, *Coreografie di stakeholders nel management del turismo sportivo*
115. Guidi Vannini (a cura di), *Archeologia Pubblica in Toscana: un progetto e una proposta*

116. Lucia Varra (a cura di), *Le case per ferie: valori, funzioni e processi per un servizio differenziato e di qualità*
117. Gianfranco Bandini (a cura di), *Manuali, sussidi e didattica della geografia. Una prospettiva storica*
118. Anna Margherita Jasink, Grazia Tucci e Luca Bombardieri (a cura di), *MUSINT. Le Collezioni archeologiche egee e cipriote in Toscana. Ricerche ed esperienze di museologia interattiva*
119. Ilaria Caloi, *Modernità Minoica. L'Arte Egea e l'Art Nouveau: il Caso di Mariano Fortuny y Madrazo*
120. Heliana Mello, Alessandro Panunzi, Tommaso Raso (edited by), *Pragmatics and Prosody. Illocution, Modality, Attitude, Information Patterning and Speech Annotation*
121. Luciana Lazzeretti, *Cluster creativi per i beni culturali. L'esperienza toscana delle tecnologie per la conservazione e la valorizzazione*
122. Maurizio De Vita (a cura di / edited by), *Città storica e sostenibilità / Historic Cities and Sustainability*
123. Eleonora Berti, *Itinerari culturali del consiglio d'Europa tra ricerca di identità e progetto di paesaggio*
124. Stefano Di Blasi (a cura di), *La ricerca applicata ai vini di qualità*
125. Lorenzo Cini, *Società civile e democrazia radicale*
126. Francesco Ciampi, *La consulenza direzionale: interpretazione scientifica in chiave cognitiva*
127. Lucia Varra (a cura di), *Dal dato diffuso alla conoscenza condivisa. Competitività e sostenibilità di Abetone nel progetto dell'Osservatorio Turistico di Destinazione*
128. Riccardo Roni, *Il lavoro della ragione. Dimensioni del soggetto nella* Fenomenologia dello spirito *di Hegel*
129. Vanna Boffo (edited by), *A Glance at Work. Educational Perspectives*
130. Raffaele Donvito, *L'innovazione nei servizi: i percorsi di innovazione nel retailing basati sul vertical branding*
131. Dino Costantini, *La democrazia dei moderni. Storia di una crisi*
132. Thomas Casadei, *I diritti sociali. Un percorso filosofico-giuridico*
133. Maurizio De Vita, *Verso il restauro. Temi, tesi, progetti per la conservazione*
134. Laura Leonardi, *La società europea in costruzione. Sfide e tendenze nella sociologia contemporanea*
135. Antonio Capestro, *Oggi la città. Riflessione sui fenomeni di trasformazione urbana*
136. Antonio Capestro, *Progettando città. Riflessioni sul metodo della Progettazione Urbana*
137. Filippo Bussotti, Mohamed Hazem Kalaji, Rosanna Desotgiu, Martina Pollastrini, Tadeusz Łoboda, Karolina Bosa, *Misurare la vitalità delle piante per mezzo della fluorescenza della clorofilla*
138. Francesco Dini, *Differenziali geografici di sviluppo. Una ricostruzione*
139. Maria Antonietta Esposito, *Poggio al vento la prima casa solare in Toscana - Windy hill the first solar house in Tuscany*
140. Maria Ranieri (a cura di), *Risorse educative aperte e sperimentazione didattica. Le proposte del progetto Innovascuola-AMELIS per la condivisione di risorse e lo sviluppo professionale dei docenti*
141. Andrea Runfola, *Apprendimento e reti nei processi di internazionalizzazione del retail. Il caso del tessile-abbigliamento*
142. Vanna Boffo, Sabina Falconi, Tamara Zappaterra (a cura di), *Per una formazione al lavoro. Le sfide della disabilità adulta*
143. Beatrice Töttössy (a cura di), *Fonti di Weltliteratur. Ungheria*
144. Fiorenzo Fantaccini, Ornella De Zordo (a cura di), *Saggi di Anglistica e Americanistica. Percorsi di ricerca*
145. Enzo Catarsi (a cura di), *The Very Hungry Caterpillar in Tuscany*
146. Daria Sarti, *La gestione delle risorse umane nelle imprese della distribuzione commerciale*
147. Raffaele De Gaudio, Iacopo Lanini, *Vivere e morire in Terapia Intensiva. Quotidianità in Bioetica e Medicina Palliativa*
148. Elisabete Figueiredo, Antonio Raschi (a cura di), *Fertile Links? Connections between tourism activities, socioeconomic contexts and local development in European rural areas*
149. Gioacchino Amato, *L'informazione finanziaria* price-sensitive
150. Nicoletta Setola, *Percorsi, flussi e persone nella progettazione ospedaliera. L'analisi configurazionale, teoria e applicazione*
151. Laura Solito e Letizia Materassi, *DIVERSE eppur VICINE. Associazioni e imprese per la responsabilità sociale*

152. Ioana Both, Ayşe Saraçgil e Angela Tarantino, *Storia, identità e canoni letterari*
153. Barbara Montecchi, *Luoghi per lavorare, pregare, morire. Edifici e maestranze edili negli interessi delle élites micenee*
154. Carlo Orefice, *Relazioni pedagogiche. Materiali di ricerca e formazione*
155. Riccardo Roni (a cura di), *Le competenze del politico. Persone, ricerca, lavoro, comunicazione*
156. Barbara Sibilio (a cura di), *Linee guida per l'utilizzo della Piattaforma Tecnologica PO.MA. Museo*
157. Fortunato Sorrentino, Maria Chiara Pettenati, *Orizzonti di Conoscenza. Strumenti digitali, metodi e prospettive per l'uomo del terzo millenni*
158. Lucia Felici (a cura di), *Alterità. Esperienze e percorsi nell'Europa moderna*
159. Edoardo Gerlini, *The Heian Court Poetry as World Literature. From the Point of View of Early Italian Poetry*
160. Marco Carini, Andrea Minervini, Giuseppe Morgia, Sergio Serni, Augusto Zaninelli, *Progetto Clic-URO. Clinical Cases in Urology*
161. Sonia Lucarelli (a cura di), *Gender and the European Union*
162. Michela Ceccorulli, *Framing irregular immigration in security terms. The case of Libya*
163. Andrea Bellini, *Il puzzle dei ceti medi*
164. Ambra Collino, Mario Biggeri, Lorenzo Murgia (a cura di), *Processi industriali e parti sociali. Una riflessione sulle imprese italiane in Cina (Jiangsu) e sulle imprese cinesi in Italia (Prato)*
165. Anna Margherita Jasink, Luca Bombardieri (a cura di), AKROTHINIA. *Contributi di giovani ricercatori italiani agli studi egei e ciprioti*
166. Pasquale Perrone Filardi, Stefano Urbinati, Augusto Zaninelli, *Progetto ABC. Achieved Best Cholesterol*
167. Iryna Solodovnik, *Repository Istituzionali, Open Access e strategie Linked Open Data. Per una migliore comunicazione dei prodotti della ricerca scientifica*
168. Andrea Arrighetti, *L'archeosismologia in architettura*
169. Lorenza Garrino (a cura di), *Strumenti per una medicina del nostro tempo. Medicina narrativa, Metodologia Pedagogia dei Genitori e International Classification of Functioning (ICF)*
170. Ioana Both, Ayşe Saraçgil e Angela Tarantino (a cura di), *Innesti e ibridazione tra spazi culturali*
171. Alberto Gherardini, *Squarci nell'avorio. Le università italiane e l'innovazione tecnologica*
172. Anthony Jensen, Greg Patmore, Ermanno Tortia (a cura di), *Cooperative Enterprises in Australia and Italy. Comparative analysis and theoretical insights*
173. Raffaello Giannini (a cura di), *Il vino nel legno. La valorizzazione della biomassa legnosa dei boschi del Chianti*
174. Gian Franco Gensini, Augusto Zaninelli (a cura di), *Progetto RIARTE. Raccontaci l'Ipertensione ARTEriosa*
175. Enzo Manzato, Augusto Zaninelli (a cura di), *Racconti 33. Come migliorare la pratica clinica quotidiana partendo dalla Medicina Narrativa*
176. Patrizia Romei, *Territorio e turismo: un lungo dialogo. Il modello di specializzazione turistica di Montecatini Terme*
177. Enrico Bonari, Giampiero Maracchi (a cura di), *Le biomasse lignocellulosiche*
178. Mastroberti C., *Assoggettamento e passioni nel pensiero politico di Judith Butler*
179. Franca Tani, Annalisa Ilari, *La spirale del gioco.Il gioco d'azzardo da attività ludica a patologia*
180. Angelica Degasperi, *Arte nell'arte. Ceramiche medievali lette attraverso gli occhi dei grandi maestri toscani del Trecento e del Quattrocento*
181. Lucilla Conigliello, Chiara Melani (a cura di), *Esperienze di gestione in una biblioteca accademica: la Biblioteca di scienze sociali dell'Ateneo fiorentino (2004-2015)*
182. Anna Margherita Jasink, Giulia Dionisio (a cura di), *Musint 2. Nuove esperienze di ricerca e didattica nella museologia interattiva*
183. Ayşe Saraçgil, Letizia Vezzosi (a cura di), *Lingue, letterature e culture migranti*
184. Gian Luigi Corinto, Roberto Fratini, *Caccia e territorio. Evoluzione della disciplina normativa in Toscana*
185. Riccardo Bruni, Dialogare*: compendio di logica*
186. Daniele Buratta, Dialogare*: compendio di matematica*
187. Manuela Lima, Dialogare*: compendio di fisica*
188. Filippo Frizzi, Dialogare*: compendio di biologia*

189. Riccardo Peruzzini, Dialogare*: compendio di chimica*
190. Guido Vannini (a cura di), *Florentia. Studi di archeologia: vol. 3*
191. Rachele Raus, Gloria Cappelli, Carolina Flinz (édité par), *Le guide touristique: lieu de rencontre entre lexique et images du patrimoine culturel. Vol. II*
192. Lorenzo Corbetta (a cura di), *Hot Topics in pneumologia interventistica*
193. Valeria Zotti, Ana Pano Alamán (a cura di), *Informatica umanistica. Risorse e strumenti per lo studio del lessico dei beni culturali*
194. Sabrina Ballestracci, *Teoria e ricerca sull'apprendimento del tedesco L2. Manuale per insegnanti in formazione*
195. Ginevra Cerrina Feroni, Veronica Federico (a cura di), *Società multiculturali e percorsi di integrazione. Francia, Germania, Regno Unito ed Italia a confronto*
196. Anna Margherita Jasink, Judith Weingarten, Silvia Ferrara (edited by), *Non-scribal Communication Media in the Bronze Age Aegean and Surrounding Areas : the semantics of a-literate and proto-literate media (seals, potmarks, mason's marks, seal-impressed pottery, ideograms and logograms, and related systems)*
197. Nicola Antonello Vittiglio, *Il lessico miceneo riferito ai cereali*
198. Rosario D'Auria, *Recall Map. Imparare e Ricordare attraverso Immagini, Colori, Forme e Font*
199. Bruno Bertaccini, *Introduzione alla Statistica Computazionale con R*