Capitolo 1

R come (super-)calcolatrice

1.1 Operazioni elementari

Lanciando il programma R compare sul *desktop* un'interfaccia grafica con al suo interno una *console*, ovvero la finestra dalla quale si danno i comandi, come mostrato in figura 1.1.[1] Il simbolo '>' nell'ultima riga della console è il cosiddetto *prompt*, ovvero il carattere che R visualizza per far capire che è pronto a ricevere comandi. Ora sta a noi. Cominciamo con qualcosa di banale, come '$2+2$'. Basta digitare tale espressione, dare `Invio` e il programma ci fornisce l'atteso risultato:

```
> 2+2
[1] 4
```

(sul significato dell'1 fra parentesi quadre torneremo nel seguito).

A questo punto abbiamo a disposizione, tanto per cominciare, una 'calcolatrice' assai potente e versatile. Seguitiamo quindi con una carrellata di operazioni autoesplicative:

```
> 3*4
[1] 12
> 12-3
[1] 9
> 20/4
[1] 5
> 20/7
[1] 2.857143
> 3^2
[1] 9
> 3^3
[1] 27
> 2^10
[1] 1024
> 2*pi
[1] 6.283145
```

Dall'ultima operazione impariamo che R conosce *pi greco* e questo ci potrà far comodo. E, ovviamente, conosce tale fondamentale costante un po' meglio delle 6 cifre decimali con cui il risultato ci è stato mostrato. Approfittiamo quindi per imparare l'opzione che ci permette di cambiare il nu-

[1] Per cominciare e tenendo conto del target di questo testo, assumiamo che il lettore stia utilizzando Windows. Per quanto riguarda l'installazione, vedi ad esempio sul sito `http://www.roma1.infn.it/~dagos/RaScuola/`, che contiene anche ulteriori informazioni riguardanti questo testo, inclusi link, esempi e figure a colori (ove i colori sono importanti).

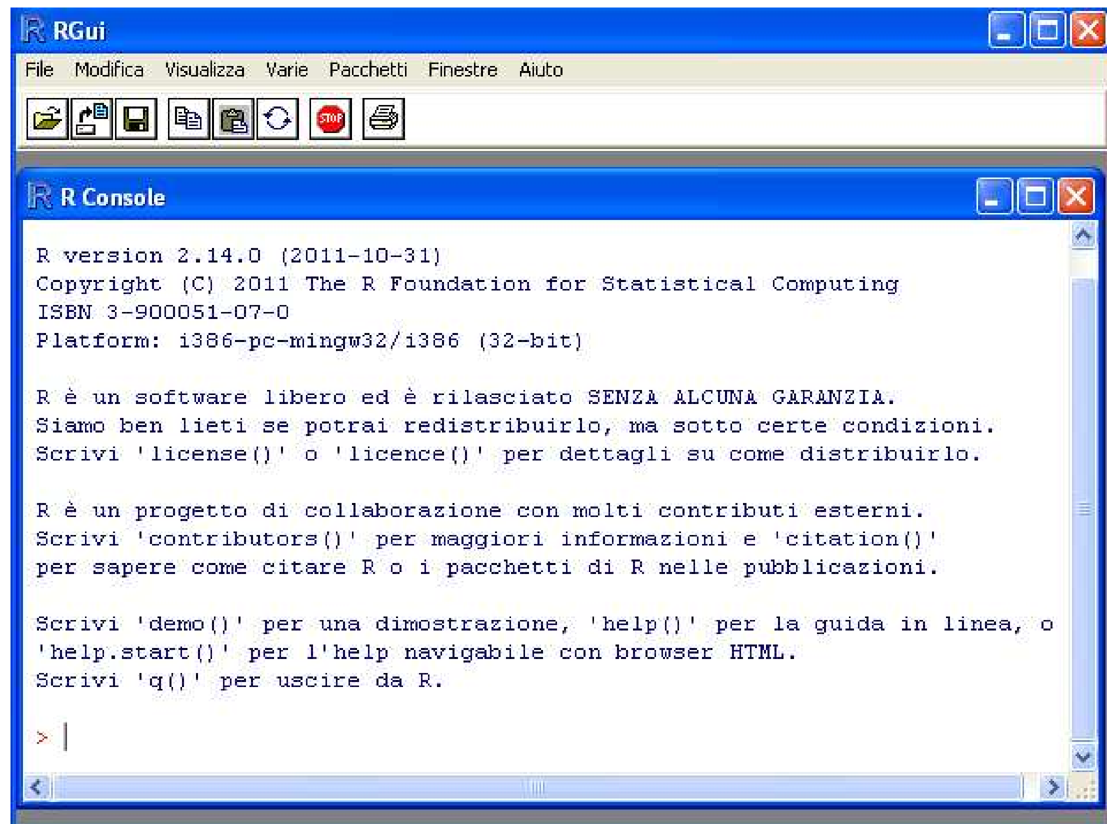

Figura 1.1: Interfaccia grafica ('GUI', ovvero *graphical user interface*, con all'interno la *console*.

mero di cifre con cui i risultati vengono *visualizzati*. Ad esempio se vogliamo π a 10 cifre (totali), diamo

```
> options(digits=10)
```

e controlliamo:

```
> pi
[1] 3.141592654
```

Ri*settiamo* quindi il valore iniziale:[2]

```
> options(digits=7)
```

Approfittiamo per imparare un piccolo e utile trucco per non dover riscrivere l'intero comando. Mediante le frecce ↑ e ↓ della tastiera possiamo ripercorrere i comandi precedenti e riproporli, eventualmente modificati.

Un'altra importante operazione è l'estrazione di radice quadrata, per la quale non c'è un sim-

[2]Le opzioni vengono resettate quando si esce da R. In ogni caso, prima di cambiare un'opzione è bene conoscere, ed eventualmente memorizzare, l'opzione standard ('*default*').
Per avere la 'lista' (nel senso del paragrafo **??**) delle opzioni:

```
> options()
```

Quindi, anticipando alcuni concetti che vedremo nel seguito, questi sono i comandi per salvare il valore di default dell'opzione `digits` e successivamente ripristinarlo:

```
> cifre.default <- options()$digits
.........
> options(digit=cifre.default)
```

bolo particolare (visto che le tastiere non hanno niente di mnemonicamente equivalente) ma dobbiamo usare un'apposita *funzione*:

```
> sqrt(25)
 [1] 5
> sqrt(30)
 [1] 5.477226
```

Gli stessi risultati potevano essere ottenuti elevando i numeri alla $1/2$ (o alla 0.5). Usando questa ben nota proprietà delle potenze, possiamo ottenere le altre radici di interesse, ad esempio

```
> 25^(1/2)
 [1] 5
> 30^0.5
 [1] 5.477226
> 8^(1/3)
 [1] 2.
```

Le parentesi dopo il simbolo di potenza sono importanti per far capire a R che vogliamo fare, ad esempio, 25 alla 1/2 e <u>non</u> 25 alla 1 e dividere il risultato per due, operazione che darebbe invece

```
> 25^1/2
 [1] 12.5
```

Anche per il fattoriale,[3] usualmente indicato con $n!$ bisogna far ricorso a una apposita funzione. Ad esempio,[4]

```
> factorial(4)
 [1] 24
```

La lista delle funzioni matematiche elementari[5] conosciute da R è mostrata in tabella 1.1.[6]

---

[3]Il fattoriale del numero naturale (ovvero intero positivo) n, è pari al prodotto di tutti i naturali da 1 a n. Ad esempio $4! = 1\times 2\times 3\times 4 = 24$. Da cui segue la proprietà $n! = n\times(n-1)!$, estesa a tutti gli interi definendo per convenzione $0! = 1$, come possiamo verificare con R:

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

[4]Qualcuno si stupirà nello scoprire che R accetta anche fattoriali di numeri non interi:

```
> factorial(pi)
 [1] 7.188083
```

In realtà R usa la funzione speciale *gamma* ('Γ'), per la quale sussiste, per n intero, la relazione $\Gamma(n+1) = n!$, come si può verificare con qualche esempio, come

```
> gamma(4 + 1)
 [1] 24
```

o ancora più chiaramente mediante

```
> factorial
function (x)
gamma(x + 1)
```

da cui impariamo che `factorial(x)` non è altro che `gamma(x+1)`. Da questo comando abbiamo pure imparato che se in R se immettiamo una funzione senza parentesi otteniamo la funzione stessa (se è scritta in R) o comunque informazioni sulla funzione.

[5]Si presti attenzione al fatto che l'aggettivo 'significativo' è usato in questa funzione con un'accezione diversa dal quella che si intende in fisica e nelle altre scienze sperimentali, la quale peraltro non sarebbe implementabile in una funzione senza ulteriori informazioni. Questa funzione considera soltanto se le cifre sono 'matematicamente' rilevanti e quindi ad esempio `signif(pi,3)` dà `3.14` e `signif(5.01,3)` dà `5.01`, mentre `signif(5.00,3)` dà `5`, benché nelle scienze sperimentali scrivere $r = 5\,\text{cm}$ sia ben diverso da scrivere $r = 5.00\,\text{cm}$ in quanto nel secondo caso l'informazione è molto più precisa del primo caso. Insomma, **nel preparare una relazione nei futuri laboratori di Fisica, non 'filtrate' i risultati con `signif`, pretendendo poi che le cifre ottenute siano quelle veramente 'significative' perché "l'ha detto R"**.

[6]Il lettore non si spaventi se qualcuna non gli è nota. Come detto nella prefazione, questo è un testo *multilevel* ed è quindi normale che alcune cose non siano ancora state incontrate. Si prenda questa tabella come quella in cui vengono descritte le funzioni di una calcolatrice scientifica.

© G. D'Agostini 2013

+, -, *, /	operazioni aritmetiche elementari
x^y	elevamento a potenza, x^y
%/%	parte intera della divisione (es. 16 %/% 3 = 5)
n%%m	n modulo m (abbr. ‘n mod m’) (es. 17 %% 5 = 2)
%*%	prodotto fra matrici
>, <, >=, <=, ==, !, &, &&, |, ||	operatori logici
sqrt(x)	radice quadrata
factorial(x)	fattoriale
abs(x)	valore assoluto (‘modulo’)
sin(x), cos(x), tan(x)	funzioni trigonometriche (x in radianti)
asin(x), acos(x), atan(x)	funzioni trigonometriche inverse
atan(x,y)	angolo tra il vettore (x, y) e l’asse delle ordinate
sinh(x), cosh(x), tanh(x)	funzioni iperboliche
asinh(x), acosh(x), atanh(x)	funzioni iperboliche inverse
exp(x)	esponenziale (e^x, o $\exp(x)$)
log(x)	logaritmo in base e
log10(x), log2(x), log(x,b)	logaritmi in base 10, 2 e b
trunc(x)	tronca la parte decimale
round(x,n)	arrotonda a n cifre decimali
signif(x,n)	arrotonda a n cifre ‘significative’
floor(x,n)	arrotonda all’intero più basso
ceiling(x,n)	arrotonda all’intero più alto

Tabella 1.1: Operatori e funzioni matematiche elementari di R (si veda la nota 5 riguardo il significato di ’cifre significative’ nel contesto di fisica, diverso da quello che si intende in Fisica).

1.2 Variabili

Ora che abbiamo imparato ad usare R come calcolatrice, vediamo come si fa, per usare il gergo delle calcolatrici tascabili, a “mettere un numero in memoria”. Ad esempio, immaginiamo di essere interessati ad usare nel seguito i risultati di due operazioni, ad esempio ‘3+4*2’ e ‘(3+4)*2’. Potremmo chiamarli quindi ‘ris1’ e ’ris2’ ed *assegnare* ad essi i risultati nel modo seguente:

```
> ris1 <- 3 + 4 * 2
> ris2 <- (3 + 4) * 2
```

Per mostrare il contenuto di una *variabile* è sufficiente dare come *comando* il suo nome, ad esempio

```
> ris1
[1] 11
> ris2
[1] 12
```

In realtà, R accetta per le assegnazioni anche il simbolo di uguale, anche se nel seguito l’operatore standard di R, ‘<-’, ha un significato più intuitivo ed inoltre può essere usato anche ‘verso destra’, con ‘->’ con significato immutato: “metti il risultato dell’operazione in questa variabile”.

© G. D’Agostini 2013

Il nome ‘variabile’ indica che il contenuto può cambiare. Ad esempio a `ris1` possiamo aggiungere 7, sottrarre al risultato 2 ed infine moltiplicare il tutto per 3. Lo facciamo usando i tre modi possibili per ridefinire la variabile:

```
> ris1 <- ris1 + 7
> ris1 = ris1 - 2
> ris1 * 3 -> ris1
> ris1
[1] 48
```

Convinti del risultato?

Si noti come l’assegnazione con le frecce risulti quella più vicina al significato dell’operazione che si sta effettuando, ovvero “metti nella variabile `ris1` il risultato dell’operazione” mentre “`ris1 = ris1 - 2`” *potrebbe sembrare* a prima vista un’equazione senza soluzione.

Una volta che delle variabili sono state definite, le possiamo usare per operazioni successive, come abbiamo iniziato a fare. Ad esempio, immaginiamo di voler calcolare area di base e volume di un cilindro di raggio 5 cm e altezza 8 cm. Le variabili in gioco sono quindi

```
> r <- 5
> h <- 8
```

Le operazioni da effettuare sono allora

```
> area.base <- pi * r^2
> volume <- area.base * h
> area.base
[1] 78.53982
> volume
[1] 628.3185
```

Infine, c’è un trucco per assegnare il valore ad una variabile e vederlo istantaneamente visualizzato sulla console: bisogna racchiudere l’intera istruzione entro una coppia di parentesi tonde. Naturalmente tale opzione può essere utile quando si assegna ad una variabile il risultato di un’operazione, ad esempio

```
> ( r <- sqrt(1225) / 7 )
[1] 5
```

1.3 Funzioni

Supponiamo ora di essere interessati a calcolare spesso area di base e volume di cilindri. Può essere quindi comodo creare una nostra funzione che, chiamata con gli opportuni *argomenti* (solo raggio nel caso dell’area di base; raggio e altezza nel caso del volume), ci restituisce il valore di interesse. Ovviamente l’esempio è un po’ artificioso, essendo le formule per il calcolo di area e volume molto semplici. Ma ci vuole poco ad immaginare casi in cui le funzioni sono effettivamente utili, come vedremo nel seguito.

In R la definizione delle semplici funzioni è veramente immediata e nemmeno lontanamente confrontabile con quella di altri linguaggi di programmazione. Vediamo come scriverle nei due casi di interesse (ovviamente i nomi li scegliamo noi in modo che siano mnemonici):

```
> cilindro.Ab <- function(r) pi * r^2
> cilindro.V <- function(r, h) pi * r^ 2 * h
```

Se ora digitiamo i nomi delle due funzioni seguiti dal tasto `Invio` otteniamo la loro definizione

```
> cilindro.Ab
function(r) pi * r^ 2
```

© G. D’Agostini 2013

```
> cilindro.V
function(r, h) pi * r^ 2 * h
```

Per usarle, basta *chiamarle* assegnando dei valori numerici agli argomenti. Ad esempio:

```
> cilindro.Ab(5)
 [1] 78.53982
> cilindro.V(5, 8)
 [1] 628.3185
```

Naturalmente possiamo memorizzare i risultati delle funzioni in opportune variabili. Ma possiamo altresì usare le funzioni nelle espressioni, ad esempio

```
> 2 * cilindro.Ab(5)
 [1] 157.0796
> cilindro.V(5, 8) / cilindro.Ab(5)
 [1] 8
> cilindro.V(5, 8) * sqrt(2) / 2
 [1] 444.2883
```

Va da sé che l'ordine degli argomenti è importante, in quanto la nostra funzione `cilindro.V` si aspetta che il primo valore sia il raggio e il secondo l'altezza e se li scambiamo otteniamo un altro risultato (errato) in quanto nella formula del volume raggio e altezza *non commutano*:

```
> cilindro.V(8, 5)
 [1] 1005.31
```

Ricordarsi l'ordine degli argomenti può diventare un vero problema quando essi sono molti. Fortunatamente R permette di indicare gli argomenti con il loro nome (anche abbreviato se non si generano ambiguità!) e quindi il loro significato diventa indipendente dalla posizione. Possiamo quindi scrivere

```
> cilindro.V(h=8, r=5)
 [1] 628.3185
```

[Si noti come nell'assegnazione dei parametri R si preferisce usare l'uguale, anche se R accetta pure l'operatore di assegnazione '`<-`'. Ci atterremo a questa convenzione.]

Le nostre due funzioni erano talmente elementari che consistevano di una sola semplice espressione. In questo caso R capisce automaticamente che deve 'rimandare indietro' (in inglese *return*, da cui il pessimo uso transitivo di 'ritornare' del gergo dei programmatori) il risultato dell'unica espressione che trova. Ma una funzione può assolvere a compiti ben più complicati, per i quali servono diverse istruzioni. Mimiamo una situazione del genere con una nuova funzione, la quale prima calcola l'area di base e poi il volume. Quando una funzione consiste di più istruzioni, queste vanno racchiuse fra parentesi graffe e

- separate da punto e virgola;
- oppure scritte su più righe.

Cominciamo con la prima opzione, utilizzabile in pratica solo quando abbiamo soltanto un paio di istruzioni. Possiamo digitare quindi:[7]

```
> cilindro.V1 <- function(r, h) { Ab <- pi * r^2 ;  Ab * h }
```

Ovviamente la funzionalità di `cilindro.V1()` è esattamente identica a quella di `cilindro.V()`. In particolare si noti come la funzione ci rimandi il risultato dell'ultima espressione, se esso non è

[7] Oramai la maggior parte delle tastiere, specialmente sui portatili, riporta esplicitamente i caratteri { e }, ma nel caso non fossero disponibili si possono ottenere dalla combinazione dei tasti `Ctrl + Alt + Shift` che permette di inserirle usando i tasti di "è" (e accentata) e di "*" (asterisco)

© G. D'Agostini 2013

assegnato a nessuna variabile. Per capire questo concetto, basta confrontare `cilindro.V1()` con la seguente variante

```
> cilindro.V1a <- function(r, h) { Ab <- pi * r^2 ; V <- Ab * h }
```

Se ora si chiama la funzione non succede niente! Ci sono allora due modi per farsi rimandare il risultato immagazzinato in `V`:

- o aggiungere una terza istruzione che contiene semplicemente `V`;
- oppure usare la funzione `return()`, con la quale si indica esplicitamente cosa rimandare indietro. Quest'ultimo è in pratica il modo standard di passare gli *eventuali* risultati a *chi* ha chiamato la funzione, anche per questioni di leggibilità del programma. Il motivo per cui si parla di 'eventuali' risultati è che una funzione può compiere diverse operazioni e quindi, in termini generici di causa ed effetto, essa produce dei *risultati*, ma può comunque non rimandare indietro degli *oggetti*, come vedremo in alcuni degli esempi seguenti.

Ecco quindi le due nuove funzioni riscritte nei due modi e aventi identica funzionalità:

```
> cilindro.V1b <- function(r, h) { Ab <- pi * r^2; V <- Ab * h; V }
> cilindro.V1c <- function(r, h) { Ab <- pi * r^2 ; V <- Ab * h; return(V) }
```

Infine ecco come si presenta una schermata di una funzione scritta su più righe:

```
> cilindro.Vc <- function(r, h) {
+ Ab <- pi * r^2
+ V <- Ab * h
+ return(V)
+ }
```

I '+' all'inizio delle righe successive alla prima non hanno il significato dei normali operatori aritmetici di addizione. Sono invece dei 'prompt' indicanti che le istruzioni immesse non sono completate in quanto ci sono delle parentesi aperte e non ancora chiuse (si provi ad esempio a scrivere '`sqrt(9`' e dare l'invio). Quando il bilancio delle parentesi è completato (vedremo che ce ne possono essere diverse coppie) riappare il prompt normale e la funzione è accettata (a meno di errori di sintassi, ma questa è un'altra storia...). Comunque, non è una buona abitudine scrivere le funzioni su più righe direttamente sulla console, anche perché se si sbaglia qualcosa bisogna ricominciare tutto da capo. Quando abbiamo funzioni lunghe è preferibile scriverle su un file ed eseguirle mediante un copia/incolla, oppure, come vedremo, dando un apposito comando che fa eseguire tutte le istruzioni di un file (file di questo tipo si chiamano *script*).

Approfittiamo per chiarire che la possibilità di avere più istruzioni sulla stessa linea *comando* non è legata alle funzioni. Possiamo ad esempio scrivere

```
> a<-2; b<-3; c<-5; d<-10; a^b * c / d
 [1] 4
```

(Chiaro il risultato?)

Possiamo allora far uso di questa accortezza per *chiamare* più volte le funzioni cilindro.Ab e cilindro.V cambiando i parametri:

```
> r<-5; h<-8;  cilindro.Ab(r); cilindro.V(r,h)
 [1] 78.53982
 [1] 628.3185
> r<-1; h<-1; cilindro.Ab(r); cilindro.V(r,h)
 [1] 3.141593
 [1] 3.141593
```

Chiaramente per non dover riscrivere l'intera riga usiamo il 'trucco' di ritornare al comando appena inviato usando la freccia in alto della tastiera (↑) e cambiando poi solo i valori numerici che ci

© G. D'Agostini 2013

interessano. A tal proposito: un modo per andare rapidamente da un estremo all'altro della riga di comando selezionata, o che si sta comunque editando, è quello di usare i comandi Ctrl-a e Ctrl-e, che portano il cursore rispettivamente all'inizio e alla fine della riga.

1.4 Scrivere ed eseguire degli script

Come anticipato, uno script è un file contenente una sequenza di istruzioni che R esegue nell'ordine in cui sono date. Si tratta quindi di imparare come scrivere e salvare, per poi successivamente rileggere ed eventualmente modificare, un file contenente istruzioni di un linguaggio di programmazione. Vedremo poi come dire a R di eseguirlo.

La prima cosa da capire è che i programmi di videoscrittura non vanno bene allo scopo, in quanto oltre al testo che vediamo contengono una grande quantità di informazione circa la *formattazione* del testo stesso (tipo, dimensione e colore dei caratteri, margini, interlinea, etc.). Vogliamo invece un file che contenga solo ed esclusivamente i caratteri che scriviamo, in gergo un file *ascii*, ove con 'ascii' si intende la codifica dei caratteri della tastiera. Una possibilità è di utilizzare il 'Blocco Note' di Windows. Meglio ancora un editor adatto alla programmazione in quanto tali strumenti hanno la possibilità di 'capire' in qualche modo il linguaggio di programmazione che si sta usando e di colorare opportunamente i caratteri in modo da facilitare la lettura del *codice*.[8] Per ora usiamo l'editor contenuto nell'interfaccia grafica della versione Windows di R.[9]

Nel menu `File` di RGui scegliamo `Nuovo script`. Si aprirà una finestra dal titolo "`Senza titolo - Editor di R`". In essa scriviamo in ordine i nostri comandi (il nuovo nome della funzione, `cilindro.Vs()`, ove 's' ci ricorda 'script' ci servirà per controllare che effettivamente stiamo utilizzando quella creata dallo script e non quella che avevamo già):

```
cilindro.Vs <- function(r,h) {
  Ab <- pi * r^2
  V <- Ab * h
  return(V)
}
```

Salviamo quindi il file. Se scegliamo l'opzione "Salva con nome" del menu `File` ci comparirà la solita finestra per scegliere il nome del file e in quale cartella ('*directory*') metterlo. È il momento di prestare un po' di attenzione, per evitare di 'perdersi' il lavoro da qualche parte dell'`hard disk`. Di default la RGui propone di salvare il file nei Documenti. Ma questa è una pessima idea. Conviene crearsi subito una cartella facilmente riconoscibile, che chiamiamo R, all'interno di Documenti, nella quale creeremo eventualmente altre cartelle per i diversi 'progetti' (vedremo fra poco l'importanza di suddividere il lavoro nelle diverse cartelle). In questo caso creiamo la cartella "PrimeProve". Per quanto riguarda il nome del file, scegliamo qualcosa di mnemonico, con l'estensione ".R", ad esempio "cilindro.R".

A questo punto facciamo in modo tale che R riconosca tale cartella. Per controllare la *cartella corrente* (*working directory*), eseguiamo il comando `getwd()`, ove 'wd' sta appunto per working directory. Sul computer di chi scrive si ottiene

```
> getwd()
[1] "C:/Documents and Settings/Administrator/My Documents"
```

[8] Ad esempio emacs+ess, jEdit, notepad+, etc: vedi `http://www.roma1.infn.it/~dagos/RaScuola/`.

[9] Gli utenti Linux possono usare gedit o emacs opportunamente corredato di ESS (Emacs Speaks Statistics).

© G. D'Agostini 2013

Per modificare la working directory abbiamo due possibilità:[10]

- la prima è di dare l'intero *percorso* (*path*) mediante il comando `setwd()`:
  `> setwd("C:/...../R/PrimeProve")`
  ove i puntini di sospensione stanno per eventuali directory intermedie.

- la seconda è di utilizzare l'opzione "Cambia directory" del menu `File` dell'interfaccia grafica.

Fatto ciò (e dopo aver controllato con `getwd()` che R 'punti' alla cartella giusta) bisogna dire a R di eseguire lo script mediante il comando `source()`. Approfittiamo per imparare un altro trucchetto per digitare meno caratteri possibili: l'*interprete* delle righe di comando di R è in grado di completare un comando e anche nomi di file e di variabili se non ci sono ambiguità (e se l'*oggetto* esiste). È sufficiente cominciare a scrivere il nome e quindi premere il tasto 'tab' (quello contrassegnato con il simbolo '→|' a sinistra di 'Q') e ad un certo punto R proseguirà da solo, evitando fra l'altro eventuali errori di battitura. Se la cosa non funziona vuol dire o che il nome non esiste, oppure che ce ne sono più d'uno che iniziano nello stesso modo. Provare ad esempio scrivendo 'c' e premendo tab (non succede niente), poi aggiungendo la 'i' e riprovando, quindi 'l', finché R comincia ad aggiungere caratteri da solo, per fermarsi alla fine del nome o nel punto dove ha incontrato delle ambiguità. Nel nostro caso, bisogna arrivare a 'sou' affinché R riesca a completare `source()`. Aggiungiamo quindi la parentesi a sinistra e gli apicetti (singoli o doppi, ma preferibilmente quelli doppi) e, premendo tab, 'magicamente' verrà completato il nome cilindro.R (la 'magia' è dovuta al fatto che al momento la directory di lavoro contiene solo questo file e quindi R 'intuisce' quello che vogliamo fare). Si tratta solo di chiudere gli apicetti (dello stesso tipo di quelli usati per all'inizio) e la parentesi e il gioco è fatto. Il comando completo è
`> source("cilindro.R")`
Da questo momento la funzione `cilindro.Vs()` è a nostra disposizione. Proviamo:

```
> cilindro.Vs(h=8, r=5)
 [1] 628.3185
```

Possiamo allora modificare lo script, risalvarlo, rieseguirlo, e così via. Ad esempio possiamo aggiungere anche la definizione di una nuova funzione per calcolare l'area di base, scritta questa volta su una sola linea, tanto per insistere sul fatto che la scrittura su più linee è solo per una questione di leggibilità del programma e non ha niente a che vedere con il fatto che le istruzioni appartengano a uno script. Il nuovo file sarà quindi

```
cilindro.Vs <- function(r,h) {
  Ab <- pi * r^2     # calcola area di base
  V <- Ab * h        # calcola volume
  return(V)          # 'ritorna' il valore
}
cilindro.Abs <- function(r) pi * r^2  # area di base
```

e il comando
`> source("cilindro.R")`
definirà entrambe le funzioni.

[10]Per gli utenti Linux la cosa più semplice è quella di chiamare sia R che l'editor dalla directory nella quale si intende lavorare. Per gli utenti Windows che non vogliono dover ripetere l'operazione ad ogni sessione, è possibile modificare la directory di lavoro in maniera permanente, aggiungendo al file Rprofile.site, che si trova nella subdirectory `\etc` della directory in cui R è stato installato, una riga, opportunamente personalizzata secondo le proprie preferenze, che punti alla directory che volete usare; per esempio `setwd ("Z:\\Dino\\R-work area\\")`. Attenzione al doppio 'backslash'!

© G. D'Agostini 2013

In questo script abbiamo usato anche dei commenti, preceduti dal simbolo 'cancelletto'. R sa che deve semplicemente ignorare qualsiasi cosa a destra di tale simbolo, fino alla fine della linea. Si raccomanda fortemente di usare dei commenti nei programmi, sia per far capire ad eventuali altri utilizzatori a cosa servono le varie istruzioni, o i vari pezzi di programma, sia perché, passato del tempo, lo stesso autore del codice può dimenticarsi cosa aveva in mente nel 'momento creativo'.

Una funzione può anche chiamare altre funzioni e quindi potremmo trasformare il nostro script in

```
cilindro.Vs <- function(r,h) {
  V <- cilindro.Abs(r) * h  # area di base per h
  return(V)                 # 'ritorna' il valore
}
cilindro.Abs <- function(r) pi * r^2   # area di base
```

Si noti come l'ordine nella definizione non è importante. Quello che conta è che al momento in cui si chiama una funzione essa sia già stata definita. Inoltre questo esempio elementare mostra un altro dei concetti moderni della programmazione: è preferibile scomporre un programma in tante sottoparti, non soltanto se esse possono essere *riutilizzate*[11] in altri programmi, ma anche per rendere più leggibile e "testabile" il codice stesso.

1.5 Salvare e recuperare la sessione di lavoro

A questo punto abbiamo definito un certo numero di variabili e di funzioni, oltre al file contenente lo script, il quale è al sicuro nell'*hard disk* del computer. Per listare tutti gli oggetti a disposizione nella sessione possiamo usare il comando `ls()`:

```
> ls()
 [1] "a"            "area.base"   "b"           "c"            "cilindro.Ab"
 [6] "cilindro.Abs" "cilindro.V"  "cilindroV1"  "cilindro.V1a" "cilindro.V1b"
[11] "cilindro.V1c" "cilindro.Vc" "cilindro.Vs" "d"            "h"
[16] "r"            "ris1"        "ris2"        "volume"
```

Fra l'altro abbiamo imparato cosa significava '[1]' prima dei risultati: indica che il 'valore' (numerico, alfanumerico o logico) alla sua destra è, per così dire, il primo di tanti. Ovviamente questa informazione è irrilevante quando abbiamo un solo elemento, ma utile quando ne abbiamo tanti. Ad esempio vediamo subito come "`r`" sia il sedicesimo, e "`ris1`", senza dover stare a contarli uno a uno. Per avere maggiori informazioni informazioni su di essi, possiamo usare il comando `ls.str()`, il cui risultato non necessita di spiegazioni:

```
> ls.str()
a :  num 2
area.base :  num 78.5
b :  num 3
c :  num 5
cilindro.Ab : function (r)
cilindro.Abs : function (r)
cilindro.V : function (r, h)
cilindro.V1 : function (r, h)
cilindro.V1a : function (r, h)
cilindro.V1b : function (r, h)
cilindro.V1c : function (r, h)
```

[11] Ho letto da qualche parte che sono 'tre' i criteri di base per scrivere del buon *software*: "*re-use, re-use, re-use*", mutuati da quelli che determinano il valore di un immobile (*location, location and location*).

© G. D'Agostini 2013

```
cilindro.Vc : function (r, h)
cilindro.Vs : function (r, h)
d :  num 10
h :  num 1
r :  num 1
ris1 :  num 48
ris2 :  num 14
volume :  num 628
```

In pratica `ls.str()` applica la funzione `str()`, che fornisce informazioni sull'oggetto, a tutti gli oggetti dell'area di lavoro.

Per uscire da R usiamo il comando `q()` (con le parentesi, in quanto si tratta di una funzione: in questo linguaggio tutti i *comandi* sono funzioni e quindi hanno bisogno di parentesi anche in assenza di argomenti):

```
> q()
```

Inviato il comando, riceviamo un messaggio che ci chiede se vogliamo salvare l'*area di lavoro* a cui rispondiamo *Yes*. Ciò significa che tutto il nostro lavoro sarà salvato nella nostra cartella in due file:[12]

- `.RHistory` contiene la lista dei comandi eseguiti durante la sessione
- `.RData` contiene invece tutti gli oggetti che avevamo creato (sotto Windows questo viene mostrato con l'icona di R).

Se nel seguito vogliamo tornare a lavorare su questo 'progetto' possiamo[13]

- o andare nella directory di lavoro ('PrimeProve', nel nostro esempio) e fare doppio-click sull'icona dello spazio di lavoro;
- oppure chiamare R dall'icona del Desktop e successivamente caricare l'area di lavoro con l'apposita opzione del menù `File`.

In effetti, una volta rientrati possiamo facilmente verificare con `ls()` e mediante l'uso della freccetta di scorrimento verticale (↑) e che ci troviamo *quasi*[14] esattamente nella situazione che avevamo lasciato e possiamo quindi procedere con il lavoro.

Siccome a questo punto la maggior parte degli oggetti della nostra lista sono inutili, in quanto prove banali, impariamo come si fa a cancellarne qualcuno usano il comando `rm()` (che sta per *remove*):

```
> rm(a)
> rm(b,c,ris1,ris2)
```

controllando dopo ogni comando la lista dei rimanenti. Oppure possiamo fare piazza pulita con

```
> rm(list=ls())
```

che significa "rimuovi tutti gli oggetti della lista ottenuta mediante il comando `ls()`". In effetti possiamo verificare che non è rimasto niente:

---

[12] Agli utenti Linux si ricorda che per visualizzare i file che cominciano con un punto bisogna usare il parametro '`-a`' di `ls`, ovvero '`ls -a`'. Ancora meglio '`ls -latr`' in quanto ci mostra i dettagli ('`-l`'), in ordine temporale ('`-t`') inverso ('`-r`'), e quindi rimangono visualizzati sullo schermo, in fondo a una lunga lista, quelli modificati più di recentemente, che sono verosimilmente quelli che ci interessano di più.

[13] Per chi usa Linux, è sufficiente rilanciare R dalla directory che avevamo usato precedentemente.

[14] Ci sono, e bisogna dire fortunatamente, dei settaggi che sono persi e vanno eventualmente ridefiniti, come ad esempio quello sulle cifre con cui i risultati devono apparire – diciamo 'fortunatamente' perché se facciamo dei pasticci con i settaggi e non ci ricordiamo quelli originali possiamo sempre uscire e rientrare.

© G. D'Agostini 2013

```
> ls()
 [1] character(0)
```

Se abbiamo il dubbio di aver cancellato qualcosa di importante che avevamo creato nella versione precedente

- innanzitutto ricordiamo che abbiamo la lista dei comandi e quindi potremmo con un po' di pazienza ricostruire gli oggetti;
- potremmo decidere che è più rapido buttare il lavoro dell'attuale sessione e semplicemente uscire dalla sessione, naturalmente rispondendo negativamente alla domanda `Salva area di lavoro?`
- se siamo interessati a conservare la sola storia dei comandi (visto che comunque gli oggetti sono stati cancellati) potremmo utilizzare l'opzione `Salva cronologia` nel menù `File`, in quanto il file della cronologia è di tipo testo ('ascii') e quindi apribile con il 'Blocco Note' o con qualsiasi editor 'serio'.

Accenniamo infine che è possibile salvare anche soltanto uno o più oggetti specifici e richiamarli in una sessione successiva mediante le istruzioni `save()` e `load()`, dei quali non ci occupiamo (servono solo in casi particolare, quando degli oggetti richiedano molto tempo per essere ricalcolati ed eventualmente occupano talmente tanta memoria che è bene non salvarli nella sessione di lavoro).

1.6 Regole del gioco e qualche consiglio

Come è noto, i computer possono fare tante cose, purché opportunamente istruiti mediante istruzioni scritte in linguaggi comprensibili sia agli umani che, previa 'traduzione', alle CPU (unità centrale di programmazione) dei computer. Come accade per tutti i linguaggi, è importante rispettare delle regole grammaticali e sintattiche, se non si vuole essere fraintesi o non compresi per niente (che in genere è meglio di essere fraintesi). Ma mentre nelle interazioni fra umani, frasi sgrammaticate e dalla sintassi improbabile sono il più delle volte ugualmente comprese correttamente, grazie al contesto e alla gestualità, se al computer non si dice esattamente cosa deve fare, 'lui' semplicemente si rifiuta di farlo, generando un messaggio di errore, oppure fa quello che ha capito 'lui'. Abbiamo già visto, ad esempio, l'importanza di rispettare le usuali regole di priorità delle operazioni aritmetiche e di usare delle parentesi per indicare esplicitamente la priorità da rispettare.

Ci sono due tipo di traduzioni del linguaggio di programmazione in *linguaggio macchina* comprensibile alla CPU. Alcuni linguaggi di programmazione esigono la *compilazione* ('traduzione') di un intero file di istruzioni, il quale, corredato delle *librerie* necessarie (ovvero programmi di base per fare operazioni elementari, come quella di visualizzare i risultati), viene trasformato in un programma *eseguibile* (i famosi `.exe` degli ambienti Windows). R appartiene invece alla famiglia dei programmi *interpretati*: le istruzioni sono tradotte una alla volta, mano a mano che vengono impartite. Come stiamo vedendo, questa modalità offre l'indubbio vantaggio di un facile uso interattivo, a spese della velocità di esecuzione. Ma questo handicap è presente solo per programmi che ripetono un grandissimo numero di volte le stesse operazioni. (Impareremo infatti nel seguito come insegnare a R a ripetere delle istruzioni per un certo numero di volte o finché non è soddisfatta una certa condizione – ad esempio "traccia la traiettoria di una particella finché essa non tocca il suolo"). Ma il fatto che le operazioni di un linguaggio interpretato siano più lente di uno compilato non deve far pensare che la risposta sia 'lenta'. Nella maggior parte dei casi essa

© G. D'Agostini 2013

è istantanea e, anche se per cose complicatissime dovessimo attendere qualche secondo, questa attesa è più che trascurabile rispetto a quella di compilare i programmi e correggere gli errori (un grosso vantaggio dei linguaggi interpretati è che si possono eseguire in modo interattivo istruzioni singole, le quali vengono poi messe in uno script mano a mano che funzionano).

Vediamo ora una lista di informazioni e regolette utili, senza nessuna pretesa di completezza, obiettivo che va al di là dello scopo di questo testo. Precisiamo inoltre che la maggior parte dei punti che seguono (dai numeri complessi in poi) **possono essere saltati in una prima lettura** e sono stati raccolti in questo paragrafo solo per essere consultati più agevolmente al momento in cui le questioni trattate serviranno per le applicazioni.

1.6.1 Interi e reali

I valori numerici possono essere *interi* (1, 7, 134, ...) oppure espressi con i decimali (*reali*)[15] (1.0, 0.089, 100.62, ...).

1.6.2 Notazione scientifica

Valori numerici molto grandi o molto piccoli possono essere rappresentati soltanto in notazione scientifica, ad esempio:

```
> 2^36
[1] 68719476736
> 2^37
[1] 1.37439e+11
> sqrt(2e+200)
[1] 1.414214e+100
> (8e+300)^(-1/3)
[1] 5e-101
```

1.6.3 Uso delle parentesi tonde

Abbiamo visto come nelle operazioni matematiche valgano le regole usuali delle priorità, la quale può essere cambiata mediante opportuno uso di parentesi, come negli esempi seguenti:

```
> 3 + 4 * 2
[1] 11
> (3 + 4) * 2
[1] 14 .
```

Ci sono altri due usi delle parentesi tonde.

- Quello indubbiamente più importante è nelle chiamate a funzioni. Infatti servono non soltanto a racchiudere eventuali argomenti, ma vanno usate anche qualora una funzione non abbia (o sia chiamata senza) argomenti.
- Nell'uso interattivo di R (ovvero, se le istruzioni vengono eseguite direttamente da console e quindi non negli script!) le assegnazioni racchiuse fra parentesi causano la visualizzazione

[15] Un matematico puro può protestare, giustamente, che si tratta di abuso dell'aggettivo 'reale' usato in matematica, in quanto, strettamente parlando, si tratta sempre di numeri razionali, eventualmente approssimazioni di numeri reali, come abbiamo visto con *pi* (come esercizio si può provare ad aumentare il numero di cifre mediate la funzione che abbiamo visto all'inizio del capitolo per capire la lunghezza massima con cui R conosce tale numero reale: si scoprirà che è più che sufficiente per tutte le applicazioni pratiche).

© G. D'Agostini 2013

immediata sullo schermo del contenuto dell'oggetto. Quindi, non soltanto

```
> ( a <- pi/2 )
[1] 1.570796
```

ma anche

```
> ( cubo <- function(x) x^3 )
[1] function(x) x^3
```

1.6.4 Nomi degli oggetti

Per quanto riguarda i nomi delle variabili e di altri oggetti che incontreremo, R usa una convenzione simile a quella di molti altri moderni linguaggi di programmazione:

- i nomi devono essere composti di lettere, cifre o punti e devono cominciare con una lettera oppure con un punto seguito da una lettera
- la lunghezza dei nomi è sostanzialmente arbitraria;
- R è un linguaggio *case sensitive*, ovvero distingue maiuscole e minuscole e quindi, ad esempio, considera `Ab` diverso da `ab`;
- è buona norma scrivere dei nomi in modo facilmente riconoscibile e dal significato intuitivo, facendo uso di maiuscole o di punti per evidenziare i nomi composti, come `VolumeCilindro` o `volume.cilindro` e simili (le abbreviazioni vanno bene se, dato il contesto non provocano ambiguità: meglio spendere qualche secondo in più a scrivere un nome lungo piuttosto che rischiare di dimenticare cosa significava);
- ci sono poi dei nomi di una lettera riservati, che è *preferibile* non usare:[16] `c`, `q`, `s`, `t`, `C`, `D`, `F`, `I` e `T`.

Quindi, riassumendo, ecco alcuni esempi di nomi corretti

`.e.accapo`, `x100`, `x.100`...

e nomi non corretti

`100x`, `area quadrato`, `area-quadrato`, `.2dato`, ...

1.6.5 Numeri complessi

R è anche in grado di riconoscere e gestire correttamente i numeri complessi, purché glielo si dica esplicitamente scrivendo anche la parte immaginaria, benché nulla. Quindi:

```
> sqrt(-1)
[1] NaN
```

[16]R, a differenza di altri linguaggi, è abbastanza tollerante, al prezzo che ci si assuma la responsabilità di quello che si fa. Ad esempio, nel seguito associeremo spesso la variabile '`t`' a dei tempi, anche se in R è già definita la funzione '`t()`' per calcolare la trasposta di una matrice. Così pure, vedremo nel seguito la funzione `c()` per creare dei 'vettori' concatenando variabili o altri vettori. Ciò nonostante, possiamo usare contemporaneamente tale funzione insieme alla variabile `c`, come ad esempio in

```
> c <- 4
> c(c(c,c,c), c(c,c,c))
[1] 4 4 4 4 4 4
```

il cui senso sarà chiaro nel paragrafo **??**.

© G. D'Agostini 2013

```
Warning message:
 [1] In sqrt(-1) : NaNs produced
```
(sul significato di NaN torneremo fra pochissimo), mentre
```
> sqrt(-1 + 0i)
 [1] 0+1i
```
E, ovviamente, possiamo usare anche variabili a valore complesso:
```
> a <- 0 + 1i; b <- 2 + 3i; c <- 2 - 3i
> a + b + c
 [1] 4+1i
> a^2
 [1] -1+0i
> b^3
 [1] -46+9i
> b^c
 [1] -75.8927-236.0767i
> b * c
 [1] -3+2i
```
Infine, esistono delle funzioni per estrarre parte reale e parte immaginaria di in numero complesso (rispettivamente Re() e Im()), farne il complesso coniugato (Conj()), nonché calcolarne *modulo* (Mod(), ma funziona anche abs()) e *fase* (Arg()) della sua *rappresentazione polare*[17]. Ad esempio:
```
> Re(b)
 [1] 2
> Im(b)
 [1] 3
> Conj(b)
 [1] 2-3i
> Mod(b)
 [1] 3.605551
> abs(b)
 [1] 3.605551
> sqrt(Re(b)^2 + Im(b)^2)
 [1] 3.605551
> sqrt(b * Conj(b))
 [1] 3.605551+0i
> as.real( sqrt(b * Conj(b)) )
 [1] 3.605551
> Arg(b)
 [1] 0.9827937
> Arg(b) * 180/pi
 [1] 56.30993
```
(Si noti come abbiamo valutato il modulo di b in tre modi diversi e nel terzo abbiamo anche usato la funzione as.real() per *forzare* il risultato ad essere rappresentato come un numero reale.)

[17] Per chi non la conosce, può saltare al capitolo **??**, dove viene spiegata più estesamente, prima di tornare qui

© G. D'Agostini 2013

1.6.6 Sequenze di caratteri ('stringhe')

Ci sono poi altri oggetti che invece di contenere dei numeri contengono dei caratteri alfanumerici, come quando immettete *nome utente* e *password* in un computer o in un servizio web. Si parla, in gergo di programmazione, di *stringhe*[18] di caratteri. Ad esempio possiamo definire

```
> nome <- 'Dante'
> cognome <- 'Alighieri'
```

su cui possiamo effettuare un certo numero di operazioni, ad esempio (le varie funzioni sono autoesplicative e, naturalmente, tutti i risultati possono essere immagazzinati in variabili, anche se non lo faremo per rendere più agili i comandi):

```
> nchar(nome) + nchar(cognome)
[1] 14
> toupper(nome)
[1] "DANTE"
> tolower(cognome)
[1] "alighieri"
> ( poeta <- paste(nome, cognome) )
[1] "Dante Alighieri"
> substring(cognome, 1, 3)
[1] "Ali"
> substring(cognome, 4, nchar(cognome))
[1] "ghieri"
> strsplit(poeta, ' ')
[[1]]
[1] "Dante "   "Alighieri"
```

L'`1` fra doppie parentesi quadre della prima linea della risposta ci indica che il risultato di `strsplit()` è un oggetto un po' più complicato di quelli visti finora, ma per il momento la cosa può essere ignorata (vedi paragrafo **??**).

1.6.7 Operazioni logiche

Un'altra importante categoria di variabili sono quelle il cui valore può essere *vero* o *falso*. Come vedremo esse sono una delle componenti fondamentali della programmazione, in quanto permettono di dire al programma di fare cose diverse a seconda delle *condizioni* che si sono verificate. È quello che succede ad esempio nei videogiochi, che reagiscono a seconda di come agite sulla tastiera o su altre interfacce. Ad esempio il programa di un videogioco può chiedere all'interfaccia con la console del gioco "Il volante è girato verso destra?". Questa domanda ammette soltanto le risposte 'vero' o 'falso' (oltre che 'non lo so', che sembra uno *stato logico* di scarso interesse, ma vedremo fra un attimo come R è anche in grado di gestire tali situazioni) e la macchinina dell'ipotetico gioco si comporta di conseguenza.

In R i due possibili stati logici sono indicati con `TRUE` e `FALSE`, anche se anche le abbreviazioni `T` e `F` sono accettate.[19] Bisogna però usare le abbreviazioni con una certa precauzione (insomma,

[18] Inglesismo di *string*, che non ha niente a che vedere con i lacci delle scarpe.

[19] In realtà nelle espressioni logiche e nelle istruzioni condizionate al posto di `FALSE` si può usare un valore numerico nullo (intero o reale), mentre un 'non zero' viene interpretato come `TRUE`. Per provare se e come è interpretato logicamente un qualsiasi valore, si provi con istruzioni tipo

```
> if (0) print('vero')
> if (1) print('vero')
> if (-pi/2) print('vero')
```

Al contrario variabili logiche possono essere usate nelle operazioni aritmetiche ed in questo caso sono convertite in 1

© G. D'Agostini 2013

è sconsigliato!) in quanto, contrariamente a TRUE o FALSE, T e F possono essere *sovrascritte*, ovvero ridefinite, dall'utente. Per capire si provi a dare i comandi 'T=5' e 'TRUE=5" e vedere la differenza ('TRUE' è una parola *riservata* di R che non può essere usata dall'utente): nel secondo caso otteniamo un messaggio di errore e TRUE continua a valere ... TRUE.

Gli *operatori logici* di R permettono di fare dei confronti fra oggetti di vario tipo. Vediamo il caso del confronto fra numeri (e va da se che possiamo usare al loro posto delle variabili numeriche):

- *uguale* ('==', da non confondere con '=' essendo quest'ultimo un operatore di assegnazione e non di confronto)
```
> 2 + 2 == 4
[1] TRUE
> 7 * 8 == 63
[1] FALSE
```
(E così abbiamo imparato che le operazioni di addizione e moltiplicazione hanno una priorità superiore a quella del *confronto*.) Ovviamente il confronto sull'uguaglianza può essere effettuato anche fra stringhe:
```
> nome == 'DANTE'
[1] FALSE
> cognome == 'Alighieri'
[1] TRUE
> nome == cognome
[1] FALSE
```

- *diverso* ('!=')
```
> 2 + 2 != 4
[1] FALSE
> 7 * 8 != 63
[1] TRUE
> nome != cognome
[1] TRUE
> toupper(nome) != 'DANTE'
[1] FALSE
```

- *maggiore* ('>')
```
> sqrt(3456) > 500
[1] FALSE
```

- *minore* ('<')
```
> 0.56^2 < 0.5
[1] TRUE
```

- *maggiore o uguale* ('>=')
```
> 5 * 5 >= 25
[1] TRUE
```

(TRUE) o 0 (FALSE). Quindi R accetta anche espressioni di questo genere
```
> pi + (pi ^ 2 > 8)
[1] 4.141593
```
ma raccomandiamo di astenersi da simili virtuosismi a meno di non avere buone ragioni per farne uso.

© G. D'Agostini 2013

A & B		
B	B	
	TRUE	FALSE
TRUE	TRUE	FALSE
FALSE	FALSE	FALSE

A | B		
B	B	
	TRUE	FALSE
TRUE	TRUE	TRUE
FALSE	TRUE	FALSE

xor(A,B)		
B	B	
	TRUE	FALSE
TRUE	FALSE	TRUE
FALSE	TRUE	FALSE

Tabella 1.2: Tavole della verità degli operatori logici AND, OR e XOR

- *minore o uguale* (‘<=’)
```
> 10/5 <= 2
[1] TRUE
```

Variabili logiche sono quelle il cui contenuto è `FALSE` o `TRUE`, ad esempio
```
> gira.a.destra <- TRUE
```
Come esercizio si provi a convincersi della correttezza del seguente risultato (a cosa vi fa pensare il nome `discr.pos`?):
```
> a <- 2; b <- -1; c <- 5; ( discr.pos <- b^2 - 4*a*c > 0 )
[1] FALSE
```
(Attenzione alle priorità e a non confondere la freccia di assegnazione con quella di confronto. Si noti anche l'uso della coppia di parentesi tonde per far visualizzare al volo il risultato dell'operazione logica.)

Ci sono inoltre degli operatori che possono essere applicato solo a espressioni o variabili logiche e il cui risultato è ancora un valore logico:

- l'operatore di *negazione* (‘!’) inverte il valore logico, ad esempio
```
> (5 !> 7)
[1] TRUE
> !discr.pos
[1] TRUE
```

- l'operatore *AND* (‘&’ o ‘&&’[20]) confronta due stati logici e dà come risultato TRUE se sono *entrambi* veri, come in
```
> (2*2 == 4) & (-4 < 0)
[1] TRUE
> (48/6 == 8) & (4+2 == 7)
[1] FALSE
```

- l'operatore *OR* (‘|’ o ‘||’) confronta due stati logici e dà come risultato TRUE se almeno uno è vero, come in
```
> (2*2 == 4) | (-4 < 0)
[1] TRUE
> (48/6 == 8) | (4+2 == 7)
[1] TRUE
```

[20]Nota (tornarci nel cap sulla programmazione): ‘&’ and ‘&&’ indicate logical AND and ‘|’ and ‘||’ indicate logical OR. The shorter form performs elementwise comparisons in much the same way as arithmetic operators. The longer form evaluates left to right examining only the first element of each vector. Evaluation proceeds only until the result is determined. The longer form is appropriate for programming control-flow and typically preferred in ‘if’ clauses.

© G. D'Agostini 2013

fare anche della tabelle a tre input e, forse, nel cap sulla programmazione mostrare come generare la tabella mediante **outer** (Estendere intro a logica: → Think Again): p, q: $p \cup q$; $\overline{p \cup q}$; $\overline{p}$; $\overline{p} \cup q$

Le cosidette *tavole della verità* di questi operatori logici (una specie di tavola pitagorica in cui, dati gli stati logici di A e di B viene riportato il valore dell'operatore logico) sono mostrate in tabella 1.2. Per completezza riportiamo nella tabella anche la tavola della verità dell'operatore *OR esclusivo*,[21] o *XOR*, per il quale R non ha un simbolo dedicato, ma ricorre all'omonima funzione `xor()`.

1.6.8 Sintassi delle istruzioni

Questa è in effetti la prima cosa che abbiamo imparato.
Riassumendo:

- R dà un 'prompt' con il quale ci fa capire che è pronto a ricevere un nuovo comando. Il default è '>', ma volendo lo possiamo anche cambiare, ad esempio
```
> options(prompt = "Cosa devo fare? ")
Cosa devo fare? 2 + 2
 [1] 4
Cosa devo fare? 2 * pi
 [1] 6.283185
Cosa devo fare? options(prompt = "> ")
>
```
(Nell'ultima istruzione abbiamo reimpostato il prompt di default.)

- Nelle istruzioni gli spazi sono irrilevanti e, ad esempio, anche un comando del genere viene interpretato correttamente:
```
> 1+2      ^      5    /    8
 [1] 5
```
(il fatto che `1+2` sia attaccato non cambia la priorità delle operazioni!). Si raccomanda di fare un buon uso degli spazi per rendere più intellegibili le istruzioni.

- Tutti i caratteri che seguono il simbolo di cancelletto vengono ignorati in quanto tutto qual che segue è considerato un commento per essere umani e non per il computer. Ovviamente se tale simbolo è racchiuso da apicetti esso viene considerato un carattere come un altro, ad esempio
```
> cancelletto = '#'
> cancelletto
 [1] "# "
```

- Le istruzioni possono essere continuate su più righe e l'interprete di R continua a dare il prompt di continuazione ('+') finché ci sono parentesi aperte in sospeso.

- Al contrario, si possono scrivere più istruzioni sulla stessa riga separandole con punto e virgola.

[21] Il risultato logico è vero se una e una soltanto delle affermazioni logiche è vera. Si noti a tale proposito l'ambiguità dell'italiano 'o' in espressioni tipo "o uno o l'altro deve essere vero" a differenza del latino *vel*, che stava per quello che chiamiamo oggi l'OR, contrapposto a *aut* per lo XOR (si pensi ad espressioni tipo "AUT... AUT...").

© G. D'Agostini 2013

- Se si dà come istruzione il nome di un oggetto, viene visualizzato il suo contenuto. Questa regola non vale quando si esegue uno script mediante il comando `source()`. In questo caso bisogna indicare esplicitamente che si vuole 'stampare' (in senso lato – qui sta per inviare sulla console) l'oggetto, ad esempio mediante la funzione `print()` (nel seguito vedremo la funzione `cat()`). Ovviamente questa funzione può essere usata anche in una sessione interattiva, anche se in questo caso è una perdita di tempo, come ad esempio

```
> print(cancelletto)
[1] "# "
```

- Le funzioni vanno sempre chiamate con le parentesi, anche quando non hanno argomenti, come abbiamo visto per i comandi `ls()`, `getwd()` e `q()`. Nel caso si chiami la funzione senza parentesi,

  – se si tratta di una funzione scritta in linguaggio R, come quelle che abbiamo creato in questo capitolo, viene mostrata la funzione stessa, ad esempio

```
> cilindro.Vs
cilindro.Vs <- function(r,h) {
  V <- cilindro.Abs(r) * h
  return(V)
}
cilindro.Abs <- function(r) pi * r^2
```

  – se invece si tratta di una funzione interna del sistema, la quale, per ragioni di velocità di esecuzione, è stata scritta nel *linguaggio C*,[22] vengono mostrate solo alcune informazioni, come nel seguente esempio

```
> sqrt
[1] function (x) .Primitive("sqrt")
```

- Come abbiamo visto, contrariamente ad altri linguaggi, in R non c'è la necessità di *dichiarare* le variabili e il loro tipo (intero, reale, alfanumerica, etc.) prima di usarle. Inoltre le variabili possono essere semplicemente 'ridefinite' con altri oggetti, come ad esempio

```
> cancelletto <- sqrt(4)
> cancelletto
[1] 2
> cancelletto <- function(x) x*x*x
> cancelletto(3)
[1] 27
```

- Non solo, ma *anche le funzioni di sistema possono essere ridefinite*, a nostro rischio e pericolo (soltanto alcuni nomi riservati non possono essere usati, come abbiamo visto per `TRUE` e `FALSE`). Ad esempio, potremmo fare il seguente scherzo sulla sessione di lavoro di un amico

```
> sqrt <- function(x) "Le radici quadrate non me le ricordo "
```

  e quando lui proverà a calcolare una radice quadrata avrà una sorpresa:

```
> sqrt(9)
[1] "Le radici quadrate non me le ricordo "
```

[22] Ebbene sì, R è scritto in C! Ma perché allora non usare direttamente il C? Lo si capirà quando lo affronterete – e chi lo conosce e ha appena cominciato a vedere R lo ha già capito.

© G. D'Agostini 2013

Ma non è grave, basta rimuovere questa funzione e R ripristinerà la sua originale:[23]

```
> rm(sqrt)
> sqrt(9)
[1] 3
```

1.6.9 Help!

R ha un sistema di *help* molto avanzato. Se vogliamo sapere cosa fa una funzione basta farne richiesta con `help()`, ad esempio

```
> help(sqrt)
```

o più brevemente con

```
> ?sqrt
```

A seconda dei sistemi, la risposta sarà sulla console stessa (e in questo caso si può scorrere il testo in alto o in basso con le freccette, mentre per tornare al prompt dei comandi occorre dare '`q()`') o (opzione di default su Windows) sul *browser* predefinito (ad esempio Firefox). Se R non trova nessuna funzione con il nome cercato, esso stesso suggerisce di provare con `help.search('nome')`, il cui alias è `??nome`. Si provino ad esempio i seguenti comandi:

```
> ?fit
> ??fit
> help.search('fit')
```

Infine si può aprire sul browser il menù principale del menù con il comando

```
> help.start()
```

Il menù navigabile permetterà di accedere a manuali, corso introduttivo, descrizione dei *pacchetti* installati e altro. [A seconda dei sistemi operativi e delle versioni di R, questa opzione potrebbe essere quella standard, senza la necessità di attivarla quindi con `help.start()`.]

Infine, per chi è in rete, oggigiorno il modo più rapido ed efficace di avere un aiuto è quello di fare una ricerca su Google con opportune parole chiavi, fra cui '`linguaggio R`' o meglio ancora '`R language`' in quanto è più facile trovare blog, forum, tutorial o altro in lingua inglese. Un comando di R permette di fare una ricerca sul sito `http://search.r-project.org` riportando il risultato sul browser. Ad esempio

```
> RSiteSearch('graphics')
```

[23]In verità anche quando `sqrt()` era 'sovrascritta' potevamo ugualmente eseguire radici quadrate, specificando esplicitamente che vogliamo usare `sqrt()` del pacchetto 'base':

```
> base::sqrt(9)
[1] 3
```

© G. D'Agostini 2013

1.7 Tabella riepilogativa

Concludiamo il capitolo con una tabella dei principali comandi e funzioni (e ‘altro’), ove il numero nella colonna centrale rappresenta la pagina della prima apparizione.

operatori e funzioni matematiche di base	7	vedi tabella 1.1
`options()`	6	mostra opzioni varie, ad es.
	6	`options()$digits`
	23	`options(prompt='...')`
`options(digits=10)`	6	cambia il numero di cifre decimali visualizzate
`function()`	9	funzione
`return()`	11	‘ritorno’ funzione
`getwd()`	12	visualizza la directory di lavoro
`setwd()`	13	cambia la directory di lavoro
`source()`	13	esegue uno script
`ls()`	14	mostra gli oggetti nello spazio di lavoro
`ls.str()`	14	come `ls()`, con più dettagli
`str()`	15	mostra informazioni su un oggetto
`q()`	15	uscire da una sessione R
`.RHistory`	15	file con la storia dei comandi
`.RData`	15	file con gli oggetti
`rm()`	15	rimuove oggetti dallo spazio di lavoro [`rm(list=ls())` rimuove tutti gli oggetti]
`save()`	16	salva lo spazio di lavoro su file
`load()`	16	carica lo spazio di lavoro da file
`Re()`, `Im()`, `Mod()`, `Arg()`, `Conj()`	19	operazioni su numeri complessi
`as.real()`	19	forza una variable ad una rappresentazione ’reale’
`nchar()`, `toupper()`, `tolower()`, `paste()`, `substring()`, `strsplit()`	20	operazioni su stringhe di caratteri
`base::sqrt()`	25	per forzare R ad usare funzione nativa
`help()`, ‘?’	25	help
`help.search()`	25	
`help.start()`	25	attiva l’help sul browser

© G. D’Agostini 2013