FONDAMENTI DI INFORMATICA

Corso di laurea in matematica Anno accademico 2004/05 Numero 1

R

R è un linguaggio di programmazione ad altissimo livello orientato soprattutto all'uso in statistica. In verità lo sbilanciamento verso la statistica non deriva dalla natura del linguaggio, ma dalla disponibilità di grandi raccolte di funzioni statistiche e dagli interessi dei ricercatori che lo hanno inventato e lo mantengono. R è gratuito e molto simile a un linguaggio commerciale, S, creato negli anni '80 e anch'esso molto usato. S viene commercializzato come sistema S-Plus. Le differenze non sono grandissime se non sul piano della programmazione, dove R aderisce a una impostazione probabilmente più maneggevole.

R ed S-Plus sono particolarmente popolari nella statistica medica, ma vengono anche usati nella statistica economica o sociale, in geografia, nella matematica finanziaria. L'alto livello del linguaggio permette di creare facilmente librerie di funzioni per nuove applicazioni. Il punto debole è la velocità di esecuzione in calcoli numerici in grandi dimensioni, mentre sono ricchissime le capacità grafiche.

Benché così indirizzato verso la statistica, R non deve essere considerato un pacchetto di statistica. È un vero linguaggio di programmazione, anzi un linguaggio di programmazione molto avanzato, e ciò permette di adattarlo ad ogni compito informatico. Nella stessa statistica questa flessibilità è molto importante proprio oggi, dove continuamente si scoprono nuovi bisogni applicativi, nuove necessità di tradurre metodi matematici, ad esempio nella statistica di complessi dati clinici o geografici, in strumenti informatici.

Le funzioni d'aiuto

R dispone di numerose funzioni d'aiuto. Da un lato ci sono varie guide disponibili sul sito, dall'altra parte si possono invocare gli aiuti mentre si sta lavorando con il programma. Con

```
help.start()
```

appare una pagina web (mantenuta sul vostro PC) attraverso la quale si accede a manuali e informazioni generali. Cliccando sulla voce *Packages* si trovano elenchi dei molti pacchetti di base o aggiuntivi disponibili.

Dopo `help.start()` le informazioni di aiuto appaiono sullo schermer del browser; per disattivare questa modalità si può usare `options(htmlhelp=FALSE)` dal terminale. Infatti spesso si lavora più velocemente rimanendo sul terminale. Ci sono diversi modi per ottenere le informazioni d'aiuto. Il modo più semplice, ma molto efficiente è di anteporre un punto interrogativo al comando su cui si desidera sapere di più; con

```
?help.start
```

vengono fornite i dettagli sull'utilizzo della `help.start`. R distingue tra il nome `f` di una funzione e la sua invocazione con `f()`; naturalmente la parentesi può anche contenere argomenti. Proprio su questi argomenti si consulterà spesso l'aiuto in linea. Leggendo più attentamente il testo della guida si osserva che le funzioni di R hanno spesso argomenti opzionali determinati dal loro nome; ciò è tipico dei linguaggi in qualche modo derivati dal Lisp e verrà ancora trattato con più dettaglio.

Per uscire da un file informativo chiamato con `?` basta premere il tasto `q`. Da R si esce con `q()` o, equivalentemente, con `quit()`. Per saperne di più si può usare il comando `?q`; le informazioni che ci vengono fornite a questo punto sono più complicate del necessario, come invero accade spesso, d'altra parte il sistema di aiuto in linea di R è veramente molto completo anche se non perfettamente organizzato, perché richiede che si sappia già come si chiamano i comandi e perché i comandi non sono raggruppati bene secondo le funzionalità. Per sapere di più su `help` guardare `?help` e `?apropos`.

Esiste comunque un'altra funzione che permette di cercare informazioni su comandi di cui non si conosce il nome o su gruppi di comandi. Questa funzione è `help.search`; per capire come bisogna utilizzarla battiamo `?help.search`. Assumiamo adesso che cerchiamo le funzioni trigonometriche. Proviamo prima con

```
help.search("trigo")
```

trovando una breve pagina d'aiuto che ci rimanda al pacchetto `Trig`. Se adesso battiamo `?Trig`, troviamo l'elenco delle funzioni disponibili

```
cos(x)
sin(x)
tan(x)
acos(x)
asin(x)
atan(x)
atan2(y,x)
```

con l'indicazione degli argomenti, seguito da un'esposizione sull'uso.

In questo numero

La matematica del futuro

R è un linguaggio molto ricco di funzioni e variazioni e perciò non facile da apprendere. Questo corso vuole perciò essere un'introduzione sistematica alla programmazione in R. Faremo tra l'altro vedere come R può essere utilizzato per creare un semplice, ma piuttosto efficiente sistema per la gestione di una banca dati. Gli esempi che proporremo saranno spesso tratti dalla statistica, ma R si presta anche molto bene a compiti di grafica e matematica.

The main scientific challenges of the twenty-first century may no longer be divided into the classical disciplines of mathematics, informatics, physics, chemistry, biology, etc. For example, theoretical biology is currently in the phase of formulating laws of nature in terms of mathematical statements; quantum chemistry has already become an important research field in applied mathematics; physics needs more and more input from computer science and mathematics, including logic and informatics; and, outside of the natural sciences, financial mathematics has developed highly reliable tools for economic market and stock analysis. But how will researchers be motivated to do interdisciplinary research in a university environment, given the current system in which academic careers (typically) advance based upon a record of publication in a single field?

www.wpi.ac.at/

And the missing ingredient in facing those problems ... is mathematics.

D. Donoho: High-dimensional data analysis - the curses and blessings of dimensionality. Internet 2000, 32p.

Installazione di R

Dalla pagina del corso scegliere la pagina di *R* e su questa *CRAN*. Vengono offerti i pacchetti per Linux, per Windows e per Mac.

L'installazione per Linux è molto semplice. Si ritira il file *.tar.gz* che si installa nel solito modo con

```
./configure
make
make install
```

A questo punto, basta battere `R` dalla tastiera e il programma parte.

Sotto Windows bisogna andare in *base* e scaricare il file *rw2001.exe*.

Programmare in R

Benché si tratti di un linguaggio ad alto livello, gli ideatori di R preferiscono presentare R come linguaggio con cui lavorare in linea e non mediante l'esecuzione di programmi scritti su files. Oggigiorno ciò non è perfettamente comprensibile ed è possibile scrivere programmi e farli eseguire nel modo familiare ai programmatori con la tecnica che adesso descriviamo e che permette allo stesso tempo il lavoro interattivo.

Prima creiamo una nuova cartella *Programmi* in cui vogliamo svolgere un determinato lavoro. In essa creiamo le due sottocartelle *Esempi* e *Libreria*. Nella prima conserveremo i nostri esempi; essa non è necessaria per la programmazione. Il contenuto della seconda può essere copiata dal sito del corso.

Sotto Windows dobbiamo creare in *Programmi* un alias per R, in modo che i nomi dei files possano essere indicati in forma abbreviata. Bisogna poi cambiare le *proprietà* del programma impostando la voce *Da* alla nome completo della cartella *Programmi*.

Nella stessa cartella *Programmi* copiamo adesso il file *alfa* dal sito del corso. Il programma principale (che cambia ogni volta a seconda dell'esperimento che stiamo eseguendo) risiederà nel file *programma* che avrà quindi funzioni simili a quelle del file *alfa.c* dei nostri programmi in C. Quando un esperimento ci sembra interessante e ben riuscito, lo copiamo da *programma* in un file della cartella *Esempi*.

Poi lavoriamo in questo modo: Facciamo partire R e battiamo, solo in questa fase,

```
source("alfa")
```

risp. `source("alfa.txt")` sotto Windows. Possiamo fare a meno di questo comando, se utilizziamo il file *.Rprofile*. Successivamente, dopo ogni cambiamento nel file *programma*, è sufficiente battere `Alfa()` dal terminale oppure ripetere questo comando usando il tasto ↑. Si esce da R con `q()`, rispondendo *no* alla domanda se vogliamo salvare l'ambiente di lavoro. In verità abbiamo ridefinito la funzione `q` in modo che la domanda non venga posta.

Lo scopo del file *alfa* è quello di rendere disponibile la funzione di esecuzione generale `Alfa` che a sua volta carica le funzioni contenute nella cartella *Libreria*. Il comando `source("alfa")` è necessario solo alla partenza di R perché, come si vede dal listato, verrà automaticamente eseguito ad ogni chiamata di `Alfa`.

In questo modo ogni cambiamento in *programma* o in uno dei files della cartella *Libreria* diventa effettivo.

La prima riga di *alfa* serve a cancellare tutti gli oggetti definiti precedentemente.

```
rm(list=ls())
#######################################
Caricalibreria = function ()
{for (x in dir("Libreria",
  full.names=T,recursive=T))
  source(x)}

# Sotto Windows "alfa.txt"
# e "programma.txt".
Alfa = function ()
{if (exists('TABELLA')) Db.salva()
source("alfa")
Caricalibreria()
source("programma")}

# Ridefinizione della funzione
# di uscita.
q = function ()
{A.q()
quit(save='no')}

# Bisogna modificare i nomi dei files.
...
```

A parte l'aggiunta dei suffissi, il file *alfa* non deve essere modificato. La funzione ausiliaria `A.q` (che scriviamo in un file separato che contiene le funzioni ausiliarie generali) chiama il comando `Db.salva()` per salvare le modifiche nella nostra banca dati; questo comando verrà trattato più avanti nel corso.

L'istruzione `quit(save='no')` richiede l'uscita senza domanda di conferma.

Scriviamo adesso la nostra prima funzione.

```
f = function (x)
{exp(-x*x)}

cat(f(2),'\n')
```

La funzione `f` così definita corrisponde alla funzione matematica $\bigcirc_x e^{-x^2}$, mentre `cat` è insieme a `print` la funzione di base per la visualizzazione (un po' complicata anch'essa come tutte le funzioni di input/output di R) che qui visualizza $f(x)$ per $x = 2$.

`'\n'` è il carattere di invio che, nell'output, fa in modo che dopo la visualizzazione il programma torni su una nuova riga.

Per eseguire il programma battiamo R dalla tastiera e poi, una volta in R, diamo il comando

```
Alfa()
```

che, in accordo con la sua definizione, carica il file *programma* e la libreria ed esegue le istruzioni così raccolte. Viene visualizzato il risultato:

```
0.01831564
```

cioè e^{-4}, come possiamo verificare aggiungendo nel file *programma* la riga

```
print(exp(-4))
```

Così possiamo continuare a lavorare, rimanendo in R, ma scrivendo il programma e le sue modifiche in *programma*, usando il terminale solo per ripetere il comando `Alfa()`. A questo scopo in ambiente Linux è sufficiente premere il tasto ↑ che utilizza la storia dei comandi dati in precedenza che si trova nel file nascosto *.Rhistory* nella nostra stessa cartella.

Il file .Rprofile

Se la nostra cartella (la stessa in cui si trovano *alfa* e *programma*) contiene un file *.Rprofile*, i comandi contenuti in questo file vengono eseguiti all'inizio di ogni sessione in R. Inseriamo quindi in esso l'istruzione

`source("alfa")` risp. `source("alfa.txt")`

potendo così successivamente fare a meno di battere questo comando ogni volta che apriamo R. Siccome il file *.Rprofile* rimane nascosto nei normali cataloghi delle cartelle e talvolta anche nel browser, conviene creare prima un file *Rprofile* e di questo un alias con il nome *.Rprofile* riconosciuto da R.

Installazione di pacchetti

Il modo più efficiente per installare un pacchetto *alfa* è di chiamare R come root, battendo successivamente il comando

```
install.packages('alfa',.libPaths()[1])
```

Il pacchetto viene poi caricato con

```
library('alfa')
```

Si possono installare più pacchetti con un solo comando:

```
install.packages(c('alfa','beta',...),
  .libPaths()[1])
```

Nomi in R

Nomi (detti anche identificatori) in R consistono di lettere, cifre o punti. Un nome non deve iniziare con una cifra e un punto iniziale non deve essere seguito da una cifra.

```
alfa
a35
.fx.588p
..66
```

sono nomi ammissibili, mentre non lo sono `35a`, `8_a6`, `a5+3`, `.66`, `.6a`. R distingue tra minuscole e maiuscole.

Bisogna anche evitare i nomi riservati di R, tra cui alcuni caratteri singoli:

```
c q t C D F I T
```

che però possono essere usati come nomi di variabili locali all'interno di funzioni.

I commenti

Se una riga contiene, al di fuori di una stringa, il carattere `#`, tutto il resto della riga è considerato un commento, compreso il carattere `#` stesso.

Molti altri linguaggi interpretati (Perl, Python, la shell di Unix) usano questo simbolo per i commenti. In C una funzione analoga è svolta dalla sequenza `//`.

Alcune costanti

Mentre π in R è una costante predefinita (`pi`), ciò non vale per il numero e che, essendo uguale ad e^1, può essere ottenuto come `exp(1)`. Ciò richiede però che ogni volta che lo utilizziamo deve essere ricalcolato; poniamo quindi le righe

```
Cm.e = 2.7182818284590452353
Cm.pid180 = 0.0174532925199432957
Cm.180dpi = 57.2957795130823208767
```

in un file della libreria che conterrà le costanti e le variabili globali. Gli altri due valori che abbiamo aggiunto corrispondono a $\frac{\pi}{180}$ e $\frac{180}{\pi}$.

Assegnamento

L'assegnamento di un valore (spesso rappresentato da un'espressione) a una variabile `x` avviene mediante l'istruzione

```
x = valore
```

Esiste anche la forma tradizionale (leggermente più generale)

```
x <- valore
```

sicuramente meno leggibile. Per saperne di più usare `?"<-"`.

Più istruzioni sulla stessa riga devono essere separate da un punto e virgola, mentre il punto e virgola alla fine di una riga è (a differenza dal C) facoltativo:

```
x=10; y=x*x+2; u=x+y
print(u)
```

Variabili globali e locali

Talvolta, e con parsimonia, si usano anche variabili *globali*. È possibile modificare il valore di una variabile globale dall'interno di una funzione seguendo il procedimento degli esempi che seguono.

Nel primo esempio vogliamo aumentare di uno il valore di un contatore globale, nel secondo esempio invece il valore di una qualsiasi variabile (che naturalmente deve essere numerica). Si osservi attentamente l'uso degli apici!

```
Aumentacontatore = function ()
assign('CONTATORE',CONTATORE+1,pos=1)

CONTATORE=0

for (k in 1:5) Aumentacontatore()
print (CONTATORE)
# output: 5
```

```
Aumenta = function (x)
{u=get(x,pos=1)
assign(x,u+1,pos=1)}

x=20
Aumenta('x') # Apici!
print(x)
# output: 21
```

In R variabili *locali* sono le variabili definite all'interno di una funzione, ad esempio mediante un'assegnazione.

Funzioni

Come abbiamo già visto negli esempi, l'intestazione di una funzione in R ha la forma seguente:

```
Fun = function (***)
```

Ad essa segue il corpo della funzione, in genere incluso tra parentesi graffe che possono mancare quando il corpo consiste solo di un'unica semplice istruzione. In caso di dubbio mettiamo sempre le graffe.

La parte indicata con `***` corrisponde agli argomenti della funzione. Nell'utilizzo degli argomenti R prevede alcuni meccanismi veramente potenti ed eleganti (per la maggior parte risalienti al Lisp), soprattutto la possibilità di usare argomenti facoltativi con impostazioni iniziali e argomenti ignoti.

Una funzione restituisce l'ultimo valore calcolato o il valore `v` se si usa l'istruzione `return(v)`. Non dimenticare le parentesi! `return()` da solo corrisponde a `return(NULL)`. Questo punto nasconde una piccola debolezza della sintassi di R; infatti quando, come accade non raramente nelle funzioni interattive, si vorrebbe avere la possibilità di uscire in modo immediato da un punto interno della funzione, l'istruzione `return()` fa in modo che la funzione restituisca il risultato `NULL` invece di nessun risultato. Se si vuole solo impedire che questo risultato nullo venga stampato, si può usare `invisible()` al posto di `return()`.

Useremo da ora in avanti per le funzioni sempre nomi che iniziano con una maiuscola, ricordandoci che sono riservati i seguenti nomi:

- `C` ... contrasti
- `D` ... derivata
- `F` ... falso
- `I` ... inibizione di interpretazione
- `T` ... vero

Programmazione funzionale

R è un linguaggio funzionale; ciò significa che, nei limiti imposti dalle condizioni necessarie di finitezza, dati insiemi X ed Y, gli elementi dell'insieme Y^X delle applicazioni da X in Y possono essere definiti e usati come gli elementi di un qualsiasi altro insiemi. Ciò corrisponde al fatto che nell'istruzione

```
f = function (***) AAAA
```

l'oggetto `function (***) AAAA`, in cui `AAAA` denota il corpo della funzione, è un oggetto di R a pieno titolo. In particolare funzioni possono essere non solo argomenti, ma anche risultati di funzioni. Ciò costituisce, sia sul piano teorico che nella espressività pratica del linguaggio, una differenza sostanziale con il C in cui funzioni possono essere argomenti ma non risultati di altre funzioni o, più precisamente, funzioni possono essere definite soltanto nella stesura del programma da parte del programmatore e non generate nel corso del programma stesso.

Possiamo ad esempio definire l'operatore di composizione di due funzioni, che corrisponde a $\mathrm{Comp} = \bigcirc_{(f,g)} f \circ g = \bigcirc_{(f,g)} \bigcirc_{x} f(g(x))$, in due righe:

```
Comp = function (f,g)
function (x) f(g(x))
```

Nell'esempio che segue formiamo

$$\mathrm{Quadsin} = (\bigcirc_{x} x^2) \circ \sin = \bigcirc_{x} \sin^2 x$$

```
Quad = function (x) x^2

# Quadrato del seno.
Quadsin = Comp(Quad,sin)

print(Quadsin(pi/4))
# output: 0.5
```

Siccome $\sin \frac{\pi}{4} = \frac{\sqrt{2}}{2}$, il risultato 0.5 è corretto. Per la stessa ragione è semplicissimo rinominare funzioni, ad esempio:

```
Radice = sqrt
print(Radice(9))
# output: 3
```

Options e print(x,n)

Il numero delle cifre significative che vengono visualizzate mediante `cat` o `print` in un numero reale è preimpostato a 7 e può essere modificato mediante il comando

```
options(digits=12)
```

per avere ad esempio 12 cifre significative. Il valore massimo è probabilmente 22, ma sembra che la precisione arrivi a circa 16 cifre significative. `options` funziona quindi in modo simile a `par`. Per vedere anche gli altri parametri provare `options()`.

Se si vuole stampare un valore solo una volta con un certo numero di cifre, si può utilizzare il secondo parametro di `print`.

Esempio:

```
e=exp(1); print(e)
# output: 2.718282

print(e,14)
# output: 2.7182818284590

options(digits=10)
print(e)
# output: 2.718281828
```

Operatori di arrotondamento

`x` denoti un numero reale. R prevede le seguenti funzioni di arrotondamento:

`round(x,n)` ... arrotondamente a `n` cifre decimali con arrotondamento alla cifra pari in caso di equidistanza.

`signif(x,n)` ... arrotondamento a `n` cifre significanti.

`trunc(x)` ... arrotondamento intero in direzione dello zero.

`floor(x)` ... parte intera di `x`; fornisce un risultato corretto anche per argomenti negativi.

`ceiling(x)` ... intero più vicino alla destra di `x`.

Queste funzioni possono essere applicate anche a vettori numerici, le prime due anche a numeri complessi.

abs (valore assoluto) e sign

`abs(x)` è il valore assoluto del numero reale o complesso `x`.

Secondo la filosofia di R, se `v` è un vettore, con `abs(v)` si ottiene il vettore dei valori assoluti degli elementi del vettore; lo stesso vale per una matrice dalla quale otteniamo quindi la matrice dei valori assoluti dei suoi coefficienti.

`sign(x)` è il segno di un numero reale (uguale a zero se `x` è uguale a zero).

Gli operatori c e seq

Uno dei punti forti di R è che molte funzioni sono definite direttamente per successioni finite di valori. Ciò significa che se definiamo una funzione in R per la funzione matematica $\bigcirc_x x^2$, la possiamo immediatamente applicare a una successione $(x_1, \ldots, x_n)$ per ottenere la successione $(x_1^2, \ldots, x_n^2)$. In C allo stesso scopo bisogna lavorare con un ciclo, ad esempio un `for`, e riflettere sulla struttura di dati che si vogliono utilizzare; in Lisp e Perl si può usare la funzione `map` che, data una funzione `f`, trasforma la successione nella successione $(f(x_1), \ldots, f(x_n))$. In R è tutto molto più semplice e automatico:

```
Quad = function (x) x^2

u=c(1,2,3,4,5)
v=Quad(u)
print(v)
```

con output

```
1 4 9 16 25
```

Si osservi qui che la successione `u` è stata creata utilizzando l'operatore `c`, il cui nome viene da *concatenate* e che unisce una sequenza di valori in un unico oggetto.

`?c` per dettagli.

Anche le operazioni algebriche vengono eseguite su tutti gli elementi di una successione; possiamo ad esempio moltiplicare una successione con un numero oppure anche due successioni tra di loro. Esempi:

```
u=c(1,2,3,4,5); v=c(2,1,3,5,6);
print(u+v)
print(u*v)
print(u+10)
print(10*u)
```

con output

```
3 3 6 9 11
2 2 9 20 30
11 12 13 14 15
10 20 30 40 50
```

```
c(12,24,60,36)/c(4,3,2)
```

dà il risultato `3 8 30 9` perché il vettore più corto in una tale espressione viene ripetuto ciclicamente. Viene però emesso un messaggio di avvertimento e quindi si dovrebbe evitare di usare questo tipo di espressioni all'interno di un programma.

Un singolo elemento è identico a un vettore con un elemento solo. In questo senso R non possiede oggetti scalari che sono invece rappresentati da vettori di lunghezza uno.

Un operatore molto utile per generare successioni di valori equidistanti è la funzione `seq`. Il risultato di

```
seq(a,b)
```

è la successione

```
a,a+1,a+2,...
```

che è continuata fino a quando non si supera `b`. Il passo di progressione è quindi `1` se non viene impostato come terzo argomento:

```
seq(a,b,p)
```

restituisce la successione

```
a,a+p,a+2p,...
```

anch'essa continuata fino a quando l'ultimo valore non supera `b`. Esempi:

```
u=seq(0,5); print(u)
u=seq(0,2,0.3); print(u)
u=seq(3.7,9); print(u)
u=seq(3.7,5,0.2); print(u)

# 0 1 2 3 4 5
# 0 0.3 0.6 0.9 1.2 1.5 1.8
# 3.7 4.7 5.7 6.7 7.7 8.7
# 3.7 3.9 4.1 4.3 4.5 4.7 4.9
```

`a:b` è un'abbreviazione per `seq(a,b,by=1)`.

Si può anche indicare il numero degli elementi della successione mediante il parametro `length`:

```
u=seq(1,length=7)
print(u)
# output: 1 2 3 4 5 6 7

u=seq(1,by=0.1,length=4)
print(u)
# output: 1.0 1.1 1.2 1.3

u=seq(1,4,length=6)
print(u)
# output: 1.0 1.6 2.2 2.8 3.4 4.0

u=seq(1,4,length=5)
print(u)
# output: 1.00 1.75 2.50 3.25 4.00
```

Questa rappresentazione è spesso da preferire quando si vuole generare una suddivisione di un intervallo in sottointervalli; bisogna solo ricordarsi che per n sottointervalli bisogna impostare `length` a $n+1$. Un esempio è la funzione `Ms.sudd` a pagina 6.

L'operatore di ripetizione rep

Per la ripetizione di elementi in un vettore (o in una lista) si usa l'operatore `rep`:

```
u=rep(1:4,2)
print(u)
# 1 2 3 4 1 2 3 4

u=rep(1:4,length.out=7)
print(u)
# 1 2 3 4 1 2 3

u=rep(1:4,each=2)
print(u)
# 1 1 2 2 3 3 4 4

u=rep(1:4,c(1,3,2,0))
print(u)
# 1 2 2 2 3 3
v=rep(1:4,c(2,0,0,3))
print(v)
# 1 1 4 4 4
```

sequence

Una funzione che si applica più raramente è `sequence`. Se `a=c(n1,n2,...)` è un vettore di numeri naturali, con `sequence(a)` si ottiene il vettore `c(1:n1,1:n2,...)`. Esempio:

```
a=c(3,1,2)
u=sequence(a)
print(u)
# 1 2 3 1 1 2
```

Argomenti ignoti (...)

In una funzione si possono usare anche argomenti non noti in anticipo; essi vengono indicati con tre punti che nel corpo della funzione possono poi essere trasformati in un vettore con

```
a=c(...)
```

oppure in una lista con

```
a=list(...)
```

Spesso in tal caso si usa un'istruzione che ha lo scopo di determinare il numero degli argomenti, per esempio

```
n=length(a)
```

I tre punti possono essere usate anche direttamente in un'altra funzione, ad esempio

```
G = function (...)
{x=***; y=***; H(x,y,...)}
```

Stringhe

Stringhe in R possono essere racchiuse indifferentemente tra virgolette o apici. Virgolette e apici all'interno di una stringa sono rappresentati da `\"` e `\'`.

Operatori logici

Vero e *falso* in R vengono espressi da `TRUE` e `FALSE` abbreviabili a `T` e `F`. Quando riportiamo l'output, scriviamo sempre `T` e `F` benché sullo schermo R stampi invece `TRUE` e `FALSE`.

Le abbreviazioni `T` e `F` sono comode ma pericolose, perché a differenza da `TRUE` e `FALSE` l'utente può ridefinirle. Ad esempio dopo `T=FALSE` e `F=TRUE` i loro significati vengono addirittura scambiati!

In un contesto logico il numero 0 viene valutato come falso, ogni altro numero come vero; viceversa in un contesto numerico `TRUE` viene convertito ad 1 e `FALSE` a 0.

`&` (AND) E `|` (OR) sono effettuati componente per componente, `&&` e `||` sulla prima componente di ogni vettore; le forme più lunghe si usano nelle istruzioni di controllo. `xor` è l'OR esclusivo, `!` la *negazione*.

```
u=c(0,2,3,0,5)
v=c(1,3,0,4,2)

print(u&v)
# output: F T F F T
print(u&&v)
# output: F

w=c(T,T,F,4,2)
print(w)
# output: 1 1 0 4 2

print(u&w)
# output: F T F F T
```

Operatori di confronto

Per uguaglianza e nonuguaglianza si usano `==` e `!=`. Naturalmente sono presenti anche gli altri confronti numerici. Gli operatori di confronto vengono eseguiti componente per componente:

```
u=c(1,2,3,4,5,6)
v=c(1,0,6,1,5,7)

print(u==v)
# output: T F F F T F

print(u!=v)
# output: F T T T F T

print(u>v)
# output: F T F T F F
```

Come si vede, == applicato a due vettori restituisce un *vettore* di valori booleani. Talvolta questo non è ciò che si vuole (ad esempio tipicamente in un `if`). La vera uguaglianza (anche più stretta di quella fornita da ==) di due oggetti in R deve essere determinata con `identical`:

```
u=c(1,2,3)
v=c(1,2,4)

print(u==v)
# output: T T F

{if (u==v) print('uguali')
else print('non uguali')}
# output: uguali

{if (identical(u,v)) print('uguali')
else print('non uguali')}
# output: non uguali

v=c(1,2,3)

{if (identical(u,v)) print('uguali')
else print('uguali')}
# output: uguali
```

In R `NULL` coincide con `c()`:

```
print identical(NULL,c())
# output: T
```

typeof

Con `typeof(x)` si ottiene il tipo di un singolo oggetto `x`. Esempi:

```
u=typeof(3)
print (u) # ouput: double

u=typeof(3.5)
print (u) # ouput: double

u=typeof("alfa")
print (u) # ouput: character

u=typeof(TRUE)
print (u) # ouput: logical

u=typeof(c(5,7))
print (u) # ouput: double
```

Un'nformazione meno fine è fornita da `mode` e `class`. Comunque in un linguaggio che non prevede dichiarazioni queste informazioni un po' difficili da imparare sono necessarie molto raramente.

if ... else

All'interno di una funzione o di un blocco tra parentesi graffe l'`if` di R viene usato nelle forme

```
if (condizione) istruzione
```

oppure

```
if (condizione) istruzione1
else istruzione2
```

Quando il comando complessivamente occupa più righe e non si trova all'interno di un blocco o di una funzione, bisogna racchiuderlo tra parentesi graffe.

ifelse

`x` sia un vettore di valori booleani o di oggetti convertibili a valori booleani, `a` e `b` due vettori qualsiasi, non necessariamente della stessa lunghezza di `x`. Allora

```
ifelse(x,a,b)
```

è il vettore che si ottiene sostituendo in `x` ogni valore `T` con un valore di `a` e ogni valore `F` con un valore di `b`, percorrendo (ciclicamente, quando necessario) i vettori `a` e `b`.

`ifelse` è effettivamente una funzione che può generare delle configurazioni piuttosto complicate e matematicamente interessanti. Assumiamo che vogliamo generare una successione della forma

$$(a_1, b_2, a_3, b_4, a_5, b_6, a_7, b_8, a_9, b_{10}, a_{11}, b_{12})$$

Allora possiamo procedere come in questo esempio:

```
x=1:12
a=c(100,NA,200,NA,300,NA);
b=c(NA,1,NA,2)
u=ifelse(x%%2,a,b)
cat(u,'\n')
```

con output

```
100 1 200 2 300 1 100 2 200 1 300 2
```

Infatti, siccome i vettori `a` e `b` vengono ripetuti ciclicamente, abbiamo questa situazione:

x	a	b
1	100	NA
2	NA	1
3	200	NA
4	NA	2
5	300	NA
6	NA	1
7	100	NA
8	NA	2
9	200	NA
10	NA	1
11	300	NA
12	NA	2

Secondo il programma, per un valore dispari nella colonna `x` viene scelto il valore da `a`, per un valore pari un valore da `b`.

`a` e `b` possono anche consistere di valori singoli, come in

```
x=c(2,5,1,2,3,4)
u=ifelse(x%%2,"dispari","pari")
for (v in u) cat(v,'\n')
```

con output

```
pari
dispari
dispari
pari
dispari
pari
```

Un uso tipico di `ifelse` è illustrato dal seguente esempio:

```
a=c(1,2,3,4,5,6)
b=c(1,5,3,4,7,0)

u=ifelse(a==b,'U','N')
print(u)
# "U" "N" "U" "U" "N" "N"
```

`ifelse` (nella sua forma non vettoriale) è una funzione molto importante anche in informatica teorica e nella teoria dei circuiti digitali, dove appare nella descrizione di *funzioni booleane* (cioè funzioni $\{0,1\}^n \longrightarrow \{0,1\}$) mediante *diagrammi binari di decisione*. Invece di `ifelse(x,a,b)` si scrive allora spesso $[x, a, b]$. Questo operatore ternario esiste anche in C dove viene scritto nella forma `x ? a : b`.

Cicli

R possiede tre istruzioni per l'esecuzione di cicli: `for`, `while` e `repeat`; esse vengono utilizzate con questa sintassi:

```
for (x in v) istruzioni

while (condizione) istruzioni

repeat istruzioni
```

dove `istruzioni` è un'istruzione singola oppure una successione di istruzioni che allora deve essere racchiusa tra parentesi graffe.

Per uscire da un ciclo si usa `break`, mentre `next` interrompe il passaggio corrente di un ciclo e porta all'immediata esecuzione del passaggio successivo, cosicché

```
for (x in 1:20)
{if (x%%2>0) next
print(x)}
```

stampa sullo schermo i numeri pari compresi tra 1 e 20.

Evitare i cicli

R non è particolarmente veloce nell'esecuzione di cicli. Le operazioni vettoriali permettono però spesso di evitare i cicli, semplificando non solo il lavoro del programmatore ma rendendo anche molto più veloce l'esecuzione dei programmi. Quindi ad esempio per creare una tavola dei quadrati dei numeri da 1 a 100 non useremo

```
tav=c()
for (n in 1:100) tav[n]=n^2
```

ma semplicemente

```
n=1:100; tav=n*n
```

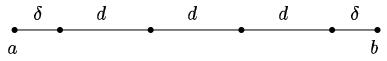
FONDAMENTI DI INFORMATICA

Corso di laurea in matematica | Anno accademico 2004/05 | Numero 2

Suddivisione di un intervallo

Illustriamo l'uso degli argomenti facoltativi nella seguente funzione che calcola i punti corrispondenti alla suddivisione di un intervallo $[a, b]$ in n sottointervalli. L'unico argomento obbligatorio è n, mentre a e b sono preimpostati ad $a = 0$ e $b = 1$.

Possiamo inoltre indicare un *margine* δ a cui corrisponde quindi una suddivisione di $[a+\delta, b-\delta]$, inizialmente impostato a $\delta = 0$, oppure il rapporto $\rho = \delta/d$ tra δ e la lunghezza d dei sottointervalli.



Per calcolare δ da ρ usiamo le relazioni

$$d = \frac{b-a-2\delta}{n} \quad \text{e } \rho = \frac{\delta}{d}$$

da cui

$$\rho = \frac{n\delta}{b-a-2\delta}$$

e

$$\rho b - \rho a - 2\rho\delta = n\delta,$$

poi

$$\delta(n+2\rho) = \rho(b-a)$$

per cui

$$\delta = \frac{\rho(b-a)}{n+2\rho}$$

Con la funzione `missing` di R si può controllare se un parametro è stato usato in una funzione. Inseriamo le funzioni riguardanti le successioni nella sezione MS della nostra libreria. La funzione cercata può essere quindi così programmata:

```
Ms.sudd = function (n,a=0,b=1,
  delta=0,rho=0)
{if (!missing(rho))
delta=rho*(b-a)/(n+2*rho)
seq(a+delta,b-delta,length=n+1)}
```

Si osservi che alla fine usiamo l'opzione `length=n+1` in `seq` perché per n sottointervalli abbiamo bisogno di $n+1$ punti.

Indici vettoriali

R possiede dei meccanismi molto generali e sofisticati per l'uso degli indici in vettori e matrici. Consideriamo il vettore

```
v=11:18
```

Indicando un singolo indice o un vettore di indici ne possiamo estrarre singoli elementi o parti:

```
w=v[2]
print(w)

w=v[c(2,3,8)]
print(w)

w=v[5:9]
print(w)
```

con output

```
12
12 13 18
15 16 17 18 NA
```

Mediante l'uso di indici negativi possiamo escludere alcuni elementi:

```
w=v[-2]
print(w)

w=v[-c(2,3,8)]
print(w)
```

ottenendo

```
11 13 14 15 16 17 18
11 14 15 16 17
```

Una caratteristica avanzata di R è che come indici si possono anche usare vettori di valori booleani, cioè vettori i cui componenti sono o `T` o `F`. Se in questo caso il vettore booleano ha una lunghezza minore di quella del vettore da cui vogliamo estrarre, i valori booleani vengono ciclicamente ripetuti. Esempi:

```
v=c(1:8)
filtro=c(F,F,T,T,F,T,F,F)
u=v[filtro]
print(u)
```

con output `3 4 6`. Infatti vengono riprodotti in `u` gli elementi di `v` che corrispondono a posizioni in cui il valore del vettore booleano è uguale a `T`.

Con `filtro=c(T,F)` otteniamo ogni secondo elemento di `v`, con `filtro=c(F,T)` ogni secondo elemento di `v`, saltando il primo:

```
v=c(1:8)
filtro=c(T,F)
u=v[filtro]
print(u)
filtro=c(F,T)
u=v[filtro]
print(u)
```

con output

```
1 3 5 7
2 4 6 8
```

Potremo così con grande facilità riscrivere in R un famoso e antico algoritmo (il *crivello di Eratostene*) che permette di trovare i numeri primi $\leq n$ per un dato numero naturale n.

In questo numero

P.quale

Creiamo adesso una funzione, simile allo `switch` e all'operatore ternario *punto interrogativo* del C, che permette di definire rapidamente funzioni mediante un elenco dei valori che la funzione assume. Esempi:

`P.quale(x,5,v1,8,v2)` è uguale a `v1` se `x` è uguale a 5 e uguale a `v2`, se `x` è uguale a 8, e corrisponderebbe quindi in C a

```
x==5 ? v1 : x==8 ? v2
```

mentre `P.quale(x,5,v1,8,v2,v3)` assume gli stessi valori dell'espressione precedente per `x` uguale a 5 o ad 8, e il valore `v3` in tutti gli altri casi, corrispondendo quindi in C a

```
x==5 ? v1 : x==8 ? v2 : v3
```

La funzione è così definita in R:

```
P.quale = function (x,...)
{a=list(...); i=1; n=length(a)
while (i<n)
{if (x==a[[i]]) return(a[[i+1]])
i=i+2}
if (i==n) return(a[[n]])}
```

`x` e i valori con cui viene confrontato devono essere numerici.

L'espressione v[]

Un po' sorprendente è che per un vettore `v` con `v[]` si ottiene il vettore dei valori di `v` stesso e non un vettore vuoto. Con ciò si può usare l'abbreviazione `v[]=7` come abbreviazione per `rep(7,length(v))`.

`v[][3]=2` è invece equivalente a `v[3]=2`; in tutto una notazione contraddittoria.

Una sorpresa nel for

Assumiamo che vogliamo eseguire una certa operazione per `k=1,...,m` se `m` è maggiore di 0, mentre l'operazione non viene eseguita se `m==0`. Allora dobbiamo usare la sequenza

```
if (m>0) for (k in 1:m)
```

che in C corrisponderebbe a

```
if (m>0) for (k=1;k<=m;k++)
```

Il programmatore in C sa però che non è necessario in questo caso anteporre al `for` la condizione `if (m>0)`, perché, se `m` non fosse maggiore di 0, il ciclo non verrebbe eseguito.

Perché in R dobbiamo allora inserire `if (m>0)`? La ragione è che il `for` percorre l'insieme `1:m` che, quando `m` è, come nel nostro programma poteva accadere, uguale a 0, corrisponde al vettore `c(1,0)`, e quindi l'istruzione che segue il `for` verrebbe eseguita due volte, prima per `k` uguale a 1 e poi per `k` uguale a 0.

R infatti definisce `a:b` per `b` minore di `a` come il vettore `c(a,a-1,a-2,...,b)`, con gli elementi elencati in ordine decrescente. Questa piccola comodità può confondere parecchio il programmatore.

NA

In statistica accade molto spesso che serie di misurazioni contengano dati non validi o non disponibili. A questo corrisponde il valore `NA` (un'abbreviazione per *not available*). La funzione `sqrt` che calcola la radice quadrata reale di un numero reale x non è definita per $x < 0$, anche se nel campo complesso x possiede le due radici $\sqrt{-x}i$ e $-\sqrt{-x}i$ (ad esempio le radici quadrate di -3 sono $\sqrt{3}i$ e $-\sqrt{3}i$). Se è dato un vettore `x` di numeri reali, non necessariamente tutti positivi, di cui, se sono positivi, vogliamo calcolare le radici quadrate, possiamo usare `ifelse` per convertire tutti i valori negativi in `NA`:

```
x=c(-1,2,0,3,-4,5)
x=ifelse(x>=0,x,NA)
u=sqrt(x)
for (v in u) cat(x,' ',v,'\n')
```

con output

```
-1 NA
2 1.414214
0 0
3 1.732051
-4 NA
5 2.236068
```

Piccoli operatori

La lunghezza di un vettore `v` si ottiene con `length(v)`.

`max(v)` e `min(v)` sono il massimo e il minimo di un vettore `v`; si ottiene il vettore il vettore `c(min(v),max(v))` con `range(v)`:

```
v=c(4:8,0,3,2,12)
print(c(min(v),max(v)))
# output: 0 12
print(range(v))
# output: 0 12
```

`prod(v)` è il prodotto degli elementi del vettore `v`; anche questa funzione può essere applicata a matrici, infatti `prod(A)` è uguale a `prod(c(A))`. Esempi:

```
print(prod(1:6))
# output: 720

print(prod(2,6,1,3,4,3))
# output: 432
```

`sum(v)` e `mean(v)` sono la somma e la media del vettore `v`. Esempi:

```
print(sum(1:6))
# output: 21

print(mean(1:6))
# output: 3.5
```

Queste funzioni vengono naturalmente usate molto spesso in statistica.

`cumsum(v)` e `diff(v)` sono i *vettori* delle somme cumulative e delle differenze di `v`.

```
a=c(3,2,5,8,7,6,10,9)
print(a)
# output: 3 2 5 8 7 6 10 9

s=cumsum(a)
print(s)
# output: 3 5 10 18 25 31 41 50

b=diff(s)
print(b)
# output: 2 5 8 7 6 10 9

print(diff(c(0,s)))
# output: 3 2 5 8 7 6 10 9
```

L'operatore che forma le differenze è inverso all'operatore di sommazione!

`sum`, `mean` e `prod` possono essere anche applicate a vettori di valori booleani che in questo caso vengono convertiti a 1 (per `T`) e 0 (per `F`). Questo implica che combinando `sum` con le operazioni logiche vettoriali si possono contare gli elementi di un vettore con determinate proprietà. Esempi:

```
u=c(T,T,F,T,F)
print(sum(u))
# output: 3

v=c(4:8,0,3,2)
# Contiamo gli elementi minori di 5:
print(sum(v<5))
# output: 4
print(mean(v<5))
# output: 0.5
```

Nell'ultimo esempio `v<5` è il vettore

```
c(T,F,F,F,F,T,T,T)
```

che nei contesti numerici viene trattato come il vettore `c(1,0,0,0,0,1,1,1)`. Altri esempi:

```
v=c(2,-3,5,8,-6,-1,0,9)
# Contiamo gli elementi negativi:
n=sum(v<0)
print(n) # 3

v=1:20
# Contiamo gli elementi
# con resto 3 modulo 4:
n=sum(v%%4==3)
print(n) # 5
```

Per invertire un vettore si usa `rev`:

```
print(rev(1:6))
# 6 5 4 3 2 1
```

Proiezione su $[0,1]$

In statistica conviene spesso trasformare i valori contenuti in un vettore `v` di dati numerici in valori compresi tra 0 e 1. Ciò può essere ottenuto con l'operazione

$$\bigcirc_x \frac{x-m}{M-m}$$

applicata agli elementi di `v`, dove m è il minimo in `v`, M il massimo. In R programmiamo

```
S.tra01 = function (v)
{m=min(v); (v-m)/(max(v)-m)}
```

Questa funzione fa parte della sezione S (statistica) della nostra libreria. Esempio:

```
x=1:5
print(S.tra01(x))
# 0.00 0.25 0.50 0.75 1.00
```

Operazioni insiemistiche

R contiene alcune semplici, ma potenti funzioni insiemistiche:

`is.element(x,A)`	...	$(x \in A)$
`union(A,B)`	...	$A \cup B$
`intersect(A,B)`	...	$A \cap B$
`setdiff(A,B)`	...	$A \setminus B$
`setequal(A,B)`	...	$(A = B)$

`is.element(x,A)` è equivalente a `x%in%A`.

Insiemi vengono rappresentati da vettori, astraendo dall'ordine degli elementi e da apparizioni molteplici dello stesso valore.

Per ottenere da un vettore un vettore con gli stessi elementi, che però appaiono una volta sola, si usa `unique`:

```
u=c(1,2,1,3,4,1,2,4,5)
v=unique(u)
print(v)
# output: 1 2 3 4 5
```

L'istruzione `duplicated(u)` restituisce invece un vettore booleano con `T` in ogni posizione in cui appare un elemento presente già in una posizione precedente di `u`:

```
u=c(1,2,1,3,2,2,4)
v=duplicated(u)
print(v)
# output: F F T F T T F
```

`u[!duplicated(u)]` è quindi probabilmente equivalente a `unique(u)`.

Potenze

Potenze possono essere scritte nella forma `x^a`; l'esponente `a` non deve essere necessariamente un numero naturale e può essere anche negativo:

```
print(2^3) # 8
print(2^(1/3)) # 1.259921
print(10^(-3)) # 0.001
```

Gli operatori %/% e %%

Per la divisione intera si usa l'operatore `%/%`, per il resto nella divisione intera `%%`. Provare `?'%%'`.

Purtroppo queste funzioni, come peraltro in C, non sono programmate correttamente dal punto di vista matematico per il caso che il secondo argomento (il divisore) sia negativo. Correggeremo (nel prossimo numero) questo difetto con alcune funzioni apposite. Esempi:

```
x=40%/%6
print(x) # 6
r=40%%6
print(r) # 4
# 40 = 6*6+4

x=(-40)%/%6
print(x) # -7
r=(-40)%%6
print(r) # 2
# -40 = -7*6+2

x=40%/%(-6)
print(x) # -7
r=40%%(-6)
print(r) # -2
# 40= -7*(-6)-2
# Ma in matematica si
# vorrebbe un resto >=0.
```

Il crivello di Eratostene

Definizione 8.1. Un numero naturale p si chiama *primo*, se $p \geq 2$ e se, per $d \in \mathbb{N}$ con $d|p$ si ha $d \in \{1, p\}$.

Osservazione 8.2. *$n \in \mathbb{N}+2$ sia un numero naturale ≥ 2 fissato. m sia un altro numero naturale con $2 \leq m \leq n$ che non sia primo. Allora esiste un primo p con $2 \leq p \leq \sqrt{n}$ tale che $p|m$.*

In altre parole, se sappiamo che $m \leq n$, per verificare che m sia primo, dobbiamo solo dimostrare che m non possiede divisori primi tra 2 e $\sqrt{n}$.

Nota 8.3. Sia $n \in \mathbb{N} + 2$ un numero naturale ≥ 2 fissato. Possiamo creare una lista di tutti i primi con il seguente procedimento che prende il nome di *crivello di Eratostene*; lo modifichiamo leggermente ai fini del programma in R con cui verrà realizzato.

(1) Definiamo due vettori variabili u e v e poniamo inizialmente
$$u = ()$$
$$v = (2, 3, \ldots, n)$$
(2) Calcoliamo la radice $r = \sqrt{n}$.
(3) Poniamo $p = v_1$.
(4) Se $p > r$, andiamo alla fine (F).
(5) Aggiungiamo p ad u, ridefinendo $u = (u, p)$ intendendo con ciò che p viene aggiunto come ultimo elemento ad u.
(6) Eliminiamo da v tutti i multipli di p (compreso p stesso).
(7) Torniamo al punto (3).
(F) A questo punto la concatenazione (u, v) di u e v rappresenta la successione ordinata dei primi $\leq n$.

Dimostrazione. Infatti il numero p al punto (3) è il più piccolo numero che non è stato eliminato da v nel passaggio precedente; esso non è quindi divisibile da nessun numero primo tra 2 e $p-1$ ed è perciò primo, cosicché lo aggiungiamo ad u. Poi eliminiamo da v tutti i multipli di p. Dall'osservazione 8.2 segue che ci possiamo fermare quando $p > \sqrt{n}$.

Possiamo trascrivere questo algoritmo direttamente in una funzione in R:

```
M.eratostene = function (n)
{v=2:n; u=c(); r=sqrt(n)
repeat
{p=v[1]; if(p>r) break;
u=c(u,p)
v=v[v%%p>0]}
c(u,v)}
```

Per provare la funzione, anticipando, per ottenere un output tabellare, la funzione `matrix` per la costruzione di matrici, usiamo

```
primi=M.eratostene(225)
A=matrix(primi,byrow=T,nrow=6)
print(A)
```

ottenendo l'output

```
  2   3   5   7  11  13  17  19
 23  29  31  37  41  43  47  53
 59  61  67  71  73  79  83  89
 97 101 103 107 109 113 127 131
137 139 149 151 157 163 167 173
179 181 191 193 197 199 211 223
```

v[v%%p>0]

Dobbiamo ancora analizzare questa misteriosa espressione che abbiamo usato nella funzione `Eratostene`. Intuitivamente è chiaro che dovrebbe rappresentare quegli elementi del vettore `v` il cui resto modulo p è maggiore di zero. Ma come si inquadra nella sintassi che abbiamo visto finora?

La spiegazione è che in `v[v%%p>0]` viene prima calcolato il vettore interno che funge da filtro, cioè il vettore booleano `v%%p>0` che, secondo il modo vettoriale delle operazioni di R, contiene il valore `T` in ogni posizione dove in `v` si trova un numero non divisibile per `p`, e nelle altre posizioni contiene il valore `F`.

All'inizio dell'algoritmo, quando `v=2:n` e `p=2`, `v%%>0` è uguale a `c(F,T,F,T,F,T,...)`, nel passaggio successivo, in cui

```
v=c(3,5,7,9,11,13,15,17,...)
```

`v%%3>0` è uguale a `c(F,T,T,F,T,T,F,T,...)`, e così via.

In modo simile (anche questo esempio è importante nella teoria dei numeri), se `v=0:n`, allora `v[v%%4==1]` consisterà di tutti i numeri naturali tra `0` ed `n` con resto 1 modulo 4.

Il teorema di Green e Tao

Una *successione aritmetica* è una successione (finita o infinita) di numeri naturali della forma
$$a, a+m, a+2m, \ldots$$
(con $m \in \mathbb{N}+1$). Se la successione è finita, chiamiamo il numero dei suoi elementi la sua lunghezza.

Soltanto nel 2004 Ben Green e Terence Tao hanno dimostrato uno dei più importanti risultati della matematica, utilizzando metodi difficili della teoria ergodica.

Essi hanno dimostrato *che esistono successioni aritmetiche (finite) di lunghezza arbitraria che consistono solo di primi*. Quanto sia difficile la dimostrazione lo si può intuire dal fatto che (secondo gli articoli di Green/Tao e di Pöppe riportati in bibliografia) la più lunga successione *nota* di questa forma ha soltanto 23 elementi; in R, se i numeri non fossero troppo grandi, essa potrebbe essere scritta nella forma

```
seq(56211383760397,length=23,
  by=44546738095860)
```

Possiamo invece verificare che la successione
$$\{199 + 210k \mid k = 0, \ldots, 9\}$$
consiste solo di primi nel modo seguente:

```
v=seq(199,by=210,length=10)
print(v)
# 199 409 619 829 1039 1249
# 1459 1669 1879 2089

P=M.eratostene(10000)
w=v[!is.element(v,P)]
print(w)
# numeric(0)
```

Assegnazione vettoriale

Le espressioni della forma `v[a]` possono essere anche usate per *assegnare* valori alla parte del vettore `v` descritta dall'espressione. Esempi:

```
v=1:8

v[c(3,5,2)]=0
print(v)
# output: 1 0 0 4 0 6 7 8

v[1:4]=c(2,3)
print(v)
# output: 2 3 2 3 0 6 7 8
# Quindi ripetizione di 2,3!

v[-c(7,8)]=c(4,5,1)
print(v)
# output: 4 5 1 4 5 1 7 8

v[-c(6,7,8)]=c(2,0,9)
print(v)
# output: 2 0 9 2 0 1 7 8
# Ma avvertenza!

v[c(T,F)]=c(5,3)
print(v)
# output: 5 0 3 2 5 1 3 8
```

Nel penultimo esempio R ci avverte che la lunghezza della parte dove vogliamo modificare i valori (in questo caso 5) non è un multiplo del vettore che contiene i nuovi valori (in questo caso 3).

Sono operazioni piuttosto potenti.

which

Se `b` un vettore di valori booleani, con `which(b)` si ottengono gli indici delle posizioni in cui `b` ha il valore `T`. Esempi:

```
b=c(T,T,F,T,F,F,T)
print(which(b))
# output: 1 2 4 7

v=c(1,3,7,6,3,2,0,5,8)
print(which(v>3))
# output: 3 4 8 9

v=c(0,1,1,0,0,1,2,3,1,0)
uno=which(v==1)
print(uno)
# output: 2 3 6 9
```

Per trovare gli indici in cui i coefficienti di un vettore numerico assumono il massimo o il minimo si può procedere così:

```
v=c(9,2,3,5,8,1,2,1,9,1)
print(which(v==max(v)))
# output: 1 9
print(which(v==min(v)))
# output: 6 8 10
```

match

`match(x,v)` è la posizione della prima apparizione di `x` nel vettore `v`. Se `x` non appare in `v`, il risultato è `NA` oppure il valore impostato mediante l'opzione `nomatch`.

Se anche `x` è un vettore, `match` restituisce il vettore dei risultati corrispondenti agli elementi di `x`. Esempi:

```
v=c(11,12,13,14,11,14,15,10)

p=match(14,v)
print(p) # output: 4
p=match(20,v)
print(p) # output: NA

p=match(20,v,nomatch=0)
print(p) # output: 0

p=match(c(13,11),v)
print(p) # output: 3 1
```

any e all

Se `v` è un vettore di valori booleani (cioè uguali a `TRUE` o `FALSE`), `any(v)` è vero se e solo se almeno uno dei componenti di `v` è vero, mentre `all(v)` è vero se e solo se tutti i componenti di `v` sono veri. Esempi:

```
v=c(T,T,F,T)
print(all(v))
# output: FALSE
print(any(v))
# output: TRUE

v=c()
print(all(v))
# output: TRUE
print(any(v))
# output: FALSE

u=c(-1,3,5,0,4)
print(u>0)
# output: FALSE TRUE TRUE FALSE TRUE
print(all(u>0))
# output: FALSE
print(any(u>0))
# output: TRUE

u=c(0.7,0.3,2,1)
v=u>u^2; print(v)
# output: TRUE TRUE FALSE FALSE
print(all(v))
# output: FALSE
print(any(v))
# output: TRUE
```

Il volume della sfera unitaria

Il volume della palla unitaria in $\mathbb{R}^m$ è, come si dimostra nei corsi di Analisi, uguale a

$$\frac{\pi^{\frac{m}{2}}}{\Gamma(\frac{m}{2}+1)}$$

È chiaro che per ottenere il volume di una palla di raggio r bisogna moltiplicare questa espressione con r^m. La funzione Γ è realizzata in R dalla funzione `gamma`. Possiamo quindi scrivere il seguente programma:

```
Vol = function (m,r=1)
{mm=m/2
vol=pi^mm/gamma(mm+1)
if (r!=1) vol=vol*r^m
vol}
```

Con

```
for (m in 1:6)
cat(m,'  ',Vol(m),'\n')
```

otteniamo allora

```
1  2
2  3.141593
3  4.18879
4  4.934802
5  5.263789
6  5.167713
```

La funzione `M.volume` della nostra libreria calcola invece il volume

$$\frac{\pi^{\frac{m}{2}}}{2^m\,\Gamma(\frac{m}{2}+1)}$$

della palla iscritta al cubo unitario tramite una formula di ricorsione (cfr. corso di Statistica multivariata).

head e tail

I primi tre elementi del vettore `v` si ottengono con `head(v,3)` o più semplicemente con `v[1:3]`, gli ultimi tre elementi con `v[(length(v)-2):length(v)]` oppure più semplicemente con `tail(v,3)`.

In particolare `tail(v,1)` è l'ultimo elemento di `v`.

Queste funzioni possono essere anche applicate a matrici e restituiscono in tal caso le prime risp. le ultime righe della matrice.

letters e LETTERS

`letters` è il vettore che consiste delle lettere minuscole `a-z`, `LETTERS` il vettore delle lettere maisucole `A-Z`.

.Last.value

L'ultimo valore calcolato sul terminale è `.Last.value`; usiamo una nostra funzione

```
A.ult = function () .Last.value
```

Conversione di tipo

Talvolta anche in R sono necessarie conversioni di tipo. Queste possono essere effettuate mediante la funzione `as` che si usa come nei seguenti esempi:

```
x=as(pi,'integer')
print(x)
# 3

x=as(pi,'numeric')
print(x)
# 3.141593

x=as(pi,'character')
print(x)
# "3.14159265358979"

x=as('135','integer')
print(x)
# 135

z=as(3,'complex')
print(z)
# 3+0i
```

Per il controllo del tipo si utilizza in modo analogo la funzione `is`:

```
print(is(pi,'integer'))
# FALSE
```

Un premio a John Chambers

Nel 1998 John Chambers, il principale ideatore di S, ha vinto il *Software System Award* della ACM (Association for Computing Machines) per la creazione di questo linguaggio per la statistica che, secondo l'ACM, *„forever altered how people analyze, visualize, and manipulate data ... S is an elegant, widely accepted, and enduring software system, with conceptual integrity, thanks to the insight, taste, and effort of John Chambers.“*

S è stato sviluppato ai famosi Bell Laboratories ed è oggi commercializzato dalla ditta Insightful (*www.insightful.com*). *„S has become a kind of lingua franca of statistical computing ...“* (John Fox)

Bibliografia

15609 **J. Chambers:** Programming with data. A guide to the S language. Springer 1998.

16699 **J. Fox:** An R and S-Plus companion to applied regression. Sage 2002.

17069 **B. Green/T. Tao:** The primes contain arbitrarily long arithmetic progressions. Internet 2004, 50p.

17067 **C. Pöppe:** Arithmetische Primzahlfolgen beliebiger Länge.
Spektrum der Wissenschaft 2005/4, 114-117.

17060 **P. Spector:** An introduction to S and S-Plus. Wadsworth 1994.

FONDAMENTI DI INFORMATICA

Corso di laurea in matematica | Anno accademico 2004/05 | Numero 3

Divisione con resto

Definizione 10.1. Per un numero reale x denotiamo con $[x]$ la sua parte intera, cioè l'intero più vicino a sinistra di x. Si ha sempre

$$x = [x] + \alpha$$

con $0 \leq \alpha < 1$. α è univocamente determinato e si chiama la parte frazionaria di x. Naturalmente

$$\alpha = 0 \iff x \in \mathbb{Z}$$

In R la funzione matematica $\bigcirc_x[x]$ è realizzata dalla funzione `floor` che, come abbiamo già osservato a pagina 3, fornisce un risultato corretto anche per $x < 0$.

Lemma 10.2. *Siano $a, b \in \mathbb{R}$ con $b > 0$. Allora*

$$a = \left[\frac{a}{b}\right] b + r$$

con $0 \leq r < b$.

Dimostrazione. $\frac{a}{b} = \left[\frac{a}{b}\right] + \alpha$

con $0 \leq \alpha < 1$. Perciò $a = \left[\frac{a}{b}\right] b + \alpha b$ con $0 \leq \alpha b < b$, perché $b > 0$. Possiamo porre $r := \alpha b$.

Osservazione 10.3. *q, b ed r siano numeri reali. Allora*

$$qb + r = (q+1)b + r - b$$

Lemma 10.4. *Siano $a, b \in \mathbb{R}$ con $b < 0$. Allora*

$$a = \left[\frac{a}{b}\right] b + s$$

con $b < s \leq 0$. Se $s < 0$, allora, ponendo $r := s - b$, abbiamo

$$a = \left(\left[\frac{a}{b}\right] + 1\right) b + r$$

con $0 < r < b$.

Dimostrazione. Abbiamo come prima

$$\frac{a}{b} = \left[\frac{a}{b}\right] + \alpha$$

con $0 \leq \alpha < 1$. Perciò $a = \left[\frac{a}{b}\right] b + \alpha b$ con $b < \alpha b \leq 0$, perché $b < 0$.

Possiamo porre $s := \alpha b$. L'ultima parte segue dall'osservazione 10.3.

Corollario 10.5. *Siano $a, b \in \mathbb{R}$ con $b \neq 0$. Allora*

$$a = qb + r$$

con $q \in \mathbb{Z}$ e $0 \leq r < |b|$. Il quoziente q ed il resto r possono essere ottenuti nel modo seguente:

- *Calcoliamo $q = \left[\frac{a}{b}\right]$ ed $r = a - qb$.*
- *Se $b > 0$ o $r = 0$, non dobbiamo fare altro.*
- *Se $b < 0$ ed $r < 0$, allora sostituiamo q con $q + 1$ ed r con $r - b$.*

Traduciamo l'algoritmo del corollario 10.5 in tre funzioni in R che verranno incluse nella sezione MA (aritmetica) della nostra libreria.

```
# Quoziente intero di 2 numeri reali.
Ma.div = function (a,b)
{q=floor(a/b); r=a-b*q
if ((b<0)&&(r<0)) q=q+1; q}

# Divisione e resto simultaneamente.
Ma.diveresto = function (a,b)
{q=floor(a/b); r=a-b*q;
if ((b<0)&&(r<0)) {q=q+1; r=r-b}
c(q,r)}

# Resto nella divisione di
# numeri reali.
Ma.resto = function (a,b)
{q=floor(a/b); r=a-b*q
if ((b<0)&&(r<0)) r=r-b; r}
```

In tutti i casi `a` e `b` possono anche essere negativi, ma `b` deve essere diverso da zero.

Riproviamo con l'esempio nella prima colonna a pagina 8:

```
u=Ma.diveresto(40,-6)
print(u)
# -6 4
```

Infatti $40 = (-6) \cdot (-6) + 4$.

Output con cat e print

In `print`, come sappiamo, il secondo argomento può essere usato per indicare il numero di cifre significative. Talvolta si usa anche ad esempio `print(round(x,4))`.

Piuttosto spesso è utile l'opzione `sep=''` in `cat`; naturalmente anche altri separatori possono essere impiegati, come in

```
cat(1:8,sep='=='); cat('\n')
# 1==2==3==4==5==6==7==8
```

Un tipico esempio dell'uso di `cat`:

```
v=1:10
cat('Il vettore ha',length(v),
  'elementi.\n')
# Il vettore ha 10 elementi.
```

In casi più complicati si userà comunque `sprintf`, eventualmente in combinazione con `cat` o `paste`. `cat` riconosce, a differenza da `sprintf`, il carattere speciale `\n`.

Al terminale spesso è sufficiente battere il nome dell'oggetto che si vuole visualizzare e non è necessario l'uso di `print` o `cat`.

In questo numero

Input dalla tastiera

Per l'input di un testo terminato da *invio* dalla tastiera si usa la funzione `readline` come in

```
nome=readline('Come ti chiami? ')
cat('Ciao, ',nome,'!\n', sep='')
```

Quando si impostano valori numerici, questi devono successivamente essere convertiti:

```
x=readline('Imposta x: ')
y=readline('Imposta y: ')
x=as(x,'numeric')
y=as(y,'numeric')
cat('x + y = ',x+y,'\n',sep='')
```

La funzione `Db.salva` del programma che svilupperemo per la gestione di una banca dati contiene istruzioni che richiedono una conferma da parte dell'utente con un messaggio variabile:

```
Db.salva = function (messaggio=
  'Vuoi salvare la tabella? (s/n): ')
{***
repeat
{s=readline(messaggio)
if (s=='s') {s=T; break}
if (s=='n') {s=F; break}}
***}
```

Se si vuole utilizzare il messaggio preimpostato, è sufficiente il comando `Db.salva()`; per cambiare il messaggio la funzione viene usata come in

```
Db.salva(messaggio=
  'Salvare i nuovi dati? (s/n): ')
```

nchar

Il numero dei caratteri di una stringa `x` lo si ottiene con `nchar(x)`. Esempio:

```
m=nchar('123456789')
print(m)
# output: 9
```

Questa funzione può essere utilizzata anche in modo vettoriale, quindi nella funzione `Sta3str` a pagina 11 abbiamo potuto utilizzare l'istruzione

```
m=max(nchar(c(A,B,C)))
```

L'opzione fill in cat

Un'opzione talvolta utile in `cat` è `fill` con cui si può impostare il numero di caratteri per riga (se gli oggetti da visualizzare sono elementi separati). Per stampare i primi ≤ 200, invece di creare una matrice come a pagina 8 avremmo anche potuto usare le istruzioni

```
primi=M.eratostene(200)
cat(primi,fill=35)
```

ottenendo però un output meno soddisfacente:

```
2 3 5 7 11 13 17 19 23 29 31 37 41
43 47 53 59 61 67 71 73 79 83 89
97 101 103 107 109 113 127 131 137
139 149 151 157 163 167 173 179
181 191 193 197 199
```

Scrivere su un file con cat

Con `cat` è anche possibile scrivere su un file, ad esempio

```
cat('alfa',file='nota')
```

oppure, se il testo deve essere aggiunto alla fine del file,

```
cat('alfa',file='nota',append=T)
```

La funzione O.s

Creiamo adesso una nostra funzione `O.s` per la stampa in forma matriciale di vettori anche non numerici. Possono essere impostati il numero di colonne o il numero di righe e il simbolo da usare per le posizioni mancanti nella matrice (le ultime posizioni in una matrice 6×4 se il vettore da stampare ha meno di 24 elementi). Anche qui anticipiamo la funzione `matrix` che verrà discussa nel prossimo numero. Nell'ultima istruzione usiamo due ulteriori opzioni di `print`, da un lato `na.print` per impostare il simbolo con cui vengono stampati i valori `NA`, dall'altro `quote=F` per togliere le virgolette nell'output.

```
O.s = function (a,col=1,righe=NA,vuoti='')
{lun=length(a)
if (!missing(righe))
col=Ma.div(lun,righe)+
  (Ma.resto(lun,righe)>0)
else righe=Ma.div(lun,col)+
  (Ma.resto(lun,col)>0)
a=c(a,rep(NA,righe*col-lun))
A=matrix(a,ncol=col,byrow=T)
print(A,na.print=vuoti,quote=F)}
```

Esempi:

```
primi=M.eratostene(300)
O.s(primi,righe=8,vuoti='*')
```

con output

```
  2   3   5   7  11  13  17  19
 23  29  31  37  41  43  47  53
 59  61  67  71  73  79  83  89
 97 101 103 107 109 113 127 131
137 139 149 151 157 163 167 173
179 181 191 193 197 199 211 223
227 229 233 239 241 251 257 263
269 271 277 281 283 293   *   *
```

```
O.s(1:40,col=4)
```

con output

```
 1  2  3  4
 5  6  7  8
 9 10 11 12
13 14 15 16
17 18 19 20
21 22 23 24
25 26 27 28
29 30 31 32
33 34 35 36
37 38 39 40
```

```
O.s(1:40,col=6)
```

con output

```
 1  2  3  4  5  6
 7  8  9 10 11 12
13 14 15 16 17 18
19 20 21 22 23 24
25 26 27 28 29 30
31 32 33 34 35 36
37 38 39 40
```

```
O.s(1:40,col=6,vuoti='--')
```

con output

```
 1  2  3  4  5  6
 7  8  9 10 11 12
13 14 15 16 17 18
19 20 21 22 23 24
25 26 27 28 29 30
31 32 33 34 35 36
37 38 39 40 -- --
```

sprintf

Per una formattazione ancora più precisa dell'output, soprattutto nel caso di tipi o formati misti, si può usare la funzione `sprintf`, che restituisce una stringa formattata che può successivamente essere stampata con `print`. Questa funzione è molto utile e imita la `sprintf` del C, con qualche piccolo difetto.

Consideriamo prima un esempio:

```
m=82; x=pi
u=sprintf("[%3d %-8.4f]",m,x)
print(u,quote=F)
# [ 82 3.1416  ]
```

Il primo parametro di `sprintf` è sempre una stringa. Questa può contenere delle *sigle di formato* che iniziano con `%` e indicano la posizione e il formato per la visualizzazione degli argomenti aggiuntivi. Nell'esempio `%3d` *tiene il posto* per il valore della variabile `m` che verrà visualizzata come intero su uno spazio di tre caratteri, mentre `%-8.4f` indica una variabile di tipo `double` che è visualizzata su uno spazio di 8 caratteri, arrotondata a 4 cifre dopo il punto decimale, e allineata a sinistra a causa del `-` (altrimenti l'allineamente avviene a destra). Quando i valori superano lo spazio indicato dalla sigla di formato, viene visualizzato lo stesso il numero completo, rendendo però imperfetta l'intabulazione che avevamo in mente. Esempio:

```
a=12345.67; b=20; n=24
u=sprintf('|%6.2f|%d|',a,n)
v=sprintf('|%6.2f|%d|',b,n)
print(u,quote=F)
print(v,quote=F)
#  |12345.67|24|
#  | 20.00|24|
```

Nonostante avessimo utilizzato la stessa sigla di formato `%6.2f` per `a` e `b`, prevedendo lo spazio di 6 caratteri per ciascuna di esse, l'allineamento non è più corretto, perché la `a` impiega più dei 6 caratteri previsti.

I formati più usati sono:

`%d`	intero
`%f`	`double`
`%s`	stringa
`%x`	rappr. esadecimale

Per specificare un segno di `%` nella sigla di formato di `printf` si usa `%%`. Mancano alcune possibilità previste nel C, in particolare `'\n'` non viene riconosciuto come carattere di invio.

All'interno della specificazione di formato si possono usare `-` per l'allineamento a sinistra e `0` per indicare che al posto di uno spazio venga usato `0` come carattere di riempimento negli allineamenti a destra. Quest'ultima opzione viene usata spesso per rappresentare numeri in formato esadecimale byte per byte. Vogliamo ad esempio scrivere la tripla di numeri $(58, 11, 6)$ in forma esadecimale e byte per byte (cioè riservando due caratteri per ciascuno dei tre numeri). Le rappresentazioni esadecimali sono `3a`, `b` e `6`, quindi con

```
a=58; b=11; c=6
u=sprintf('%2x%2x%2x',a,b,c)
print(u,quote=F)
```

otteniamo `3a b 6` come output; per ottenere il formato desiderato `3a0b06` (richiesto ad esempio dalle specifiche dei colori in HTML) dobbiamo usare invece

```
a=58; b=11; c=6
u=sprintf('%02x%02x%02x',a,b,c)
print(u,quote=F)
```

Vogliamo adesso stampare tre stringhe variabili su righe separate; per allinearle, calcoliamo il massimo `m` delle loro lunghezze, usando la funzione `nchar`:

```
Sta3str = function (A,B,C)
{m=max(nchar(c(A,B,C)))
formato=sprintf('|%%%ds|',m)
for (x in c(A,B,C))
{a=sprintf(formato,x)
print(a,quote=F)}}
```

Con

```
Sta3str('Ferrara','Roma','Rovigo')
```

si ottiene l'output desiderato:

```
|Ferrara|
|   Roma|
| Rovigo|
```

Anche `sprintf` può essere usata in modo vettoriale, raramente necessario.

FONDAMENTI DI INFORMATICA

Corso di laurea in matematica — Anno accademico 2004/05 — Numero 4

Matrici

Matrici possono essere definite con il comando `matrix`. I componenti di una matrice possono essere numeri reali o complessi, stringhe oppure avere il valore `NA`.

`matrix` ha come argomento obbligatorio un vettore di dati; inoltre si possono indicare con `nrow` il numero delle righe o con `ncol` il numero delle righe. La matrice viene riempita *colonna per colonna*, se non si indica `byrow=T`. Se `a` è una matrice, con `c(a)` si ottiene il vettore dei suoi elementi, elencati colonna per colonna. Esempi:

```
dati=c(1,2,3,4,5,6,7,8,9,10,11,12)
a=matrix(dati,nrow=3)
print(a); print(c(a))
```

con output semplificato

```
1  4  7  10
2  5  8  11
3  6  9  12

1 2 3 4 5 6 7 8 9 10 11 12
```

e invece, con gli stessi dati,

```
a=matrix(dati,nrow=3,byrow=T)
print(a); print(c(a))
```

con output

```
1   2   3   4
5   6   7   8
9  10  11  12

1 5 9 2 6 10 3 7 11 4 8 12
```

Definiamo una nostra funzione per la creazione di una matrice, in cui è preimpostato il riempimento riga per riga:

```
Mm = function (dati,col,righe,pc=F)
matrix(dati,ncol=col,
  nrow=righe,byrow=!pc)
```

Scegliendo l'opzione `pc=T` anche con la nostra funzione la matrice viene riempita colonna per colonna.

Una matrice è direttamente convertibile in un vettore mediante l'operatore `c`, come abbiamo visto prima, ma anche in ogni contesto in cui R si aspetta un vettore. Quindi è corretta la sequenza

```
A=Mm(1:12,col=3)
print(A)

print(c(A))

B=Mm(A,col=4,pc=T)
print(B)
```

con output semplificato

```
 1   2   3
 4   5   6
 7   8   9
10  11  12

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

1  4  7  10
2  5  8  11
3  6  9  12
```

Il prodotto di Kronecker

Il prodotto tensoriale o di Kronecker di due matrici

$$A = \begin{pmatrix} a_{11} & \dots & a_{1m} \\ \vdots & & \vdots \\ a_{n1} & \dots & a_{nm} \end{pmatrix} \quad e \quad \mathrm{B} = \begin{pmatrix} b_{11} & \dots & b_{1s} \\ \vdots & & \vdots \\ b_{r1} & \dots & b_{rs} \end{pmatrix} \quad \text{è la matrice}$$

$$A \otimes B := \begin{pmatrix} a_{11}B & \dots & a_{1m}B \\ \vdots & & \vdots \\ a_{n1}B & \dots & a_{nm}B \end{pmatrix}$$

$$= \begin{pmatrix} a_{11}b_{11} & \dots & a_{11}b_{1s} & \dots & a_{1m}b_{11} & \dots & a_{1m}b_{1s} \\ \vdots & & \vdots & & \vdots & & \vdots \\ a_{11}b_{r1} & \dots & a_{11}b_{rs} & \dots & a_{1m}b_{r1} & \dots & a_{1m}b_{rs} \\ \dots & & & \dots & & & \dots \\ a_{n1}b_{11} & \dots & a_{n1}b_{1s} & \dots & a_{nm}b_{11} & \dots & a_{nm}b_{1s} \\ \vdots & & \vdots & & \vdots & & \vdots \\ a_{n1}b_{r1} & \dots & a_{n1}b_{rs} & \dots & a_{nm}b_{r1} & \dots & a_{nm}b_{rs} \end{pmatrix}$$

AB è quindi una matrice $nr \times ms$. In R essa corrisponde a `kronecker(A,B)`. Come in `outer` (pagina 14), anche in `kronecker` invece del prodotto si possono usare altre funzioni, con `kronecker(A,B,f)` si ottiene la matrice

$$\begin{pmatrix} f(a_{11},b_{11}) & \dots & f(a_{11},b_{1s}) & \dots & f(a_{1m},b_{11}) & \dots & f(a_{1m},b_{1s}) \\ \vdots & & \vdots & & \vdots & & \vdots \\ f(a_{11},b_{r1}) & \dots & f(a_{11},b_{rs}) & \dots & f(a_{1m},b_{r1}) & \dots & f(a_{1m},b_{rs}) \\ \dots & & & \dots & & & \dots \\ f(a_{n1},b_{11}) & \dots & f(a_{n1},b_{1s}) & \dots & f(a_{nm},b_{11}) & \dots & f(a_{nm},b_{1s}) \\ \vdots & & \vdots & & \vdots & & \vdots \\ f(a_{n1},b_{r1}) & \dots & f(a_{n1},b_{rs}) & \dots & f(a_{nm},b_{r1}) & \dots & f(a_{nm},b_{rs}) \end{pmatrix}$$

In questo numero

Operazioni matriciali

Abbiamo due operazioni di moltiplicazione per matrici: da un lato il prodotto elemento per elemento (prodotto di Hadamard), dall'altro la moltiplicazione matriciale.
A e B siano due matrici con le appropriate dimensioni (cioè stesso numero di righe e stesso numero di colonne per il prodotto di Hadamard, numero di colonne di A uguale al numero di righe di B per il prodotto matriciale). I coefficienti P_{ij} del prodotto di Hadamard $P := A \bullet B$ sono definiti da

$$P_{ij} = A_{ij} B_{ij}$$

i coefficienti C_{ij} del prodotto matriciale $C := AB$ da

$$C_{ij} = \sum_k A_{ik} B_{kj}.$$

Possiamo naturalmente anche formare la somma di due matrici della stessa forma e moltiplicare una matrice con un numero reale o complesso:

$$(A + B)_{ij} = A_{ij} + B_{ij}$$
$$(tA)_{ij} = tA_{ij}$$

In R le operazioni matriciali corrispondono ai seguenti comandi:

`A+B`	...	addizione
`t*A`	...	moltiplicazione di una matrice con un numero
`A*B`	...	prodotto di Hadamard
`A%*%B`	...	prodotto matriciale
`A^2`	...	non A^2, ma $A \bullet A$!
`t(A)`	...	trasposta A^t di A
`crossprod(A,B)`	...	$A^t B$

L'espressione `A%*%b` denota anche il prodotto di una matrice `A` con un vettore `b`, in accordo con la notazione matematica; confonde invece probabilmente che si possa usare l'operatore `%*%` anche per il prodotto scalare di vettori.

Indici matriciali

Il coefficiente nella `i`-esima riga e `j`-esima colonna di una matrice `A` viene denotato con `A[i,j]`.

Anche qui possiamo però usare espressioni più complesse, cosicché ad esempio con `A[c(1,3),c(2,4,6)]` otteniamo la matrice 2x3 che consiste dei coefficienti nella prima e terza riga e nella seconda, quarta e sesta colonna di `A`.

Se `u` e `v` sono vettori di indici, allora `A[u,]` è la matrice che consiste delle righe di `A` con indici da `u`, mentre `A[,v]` consiste delle colonne con indici da `v`. `A[-u,]` è invece la matrice da cui abbiamo tolto le righe con indici da `u`. Anche vettori booleani di indici possono essere usati per matrici. Esempi:

```
dati=c(11:16,21:26,31:36,41:46)

A=Mm(dati,righe=4)
print(A)
```

con output

```
11   12   13   14   15   16
21   22   23   24   25   26
31   32   33   34   35   36
41   42   43   44   45   46
```

---

e, continuando con la stessa matrice `A`,

```
B=A[c(1,2),]
print(B)
```

con output

```
11   12   13   14   15   16
21   22   23   24   25   26
```

```
B=A[,c(3,5)]
print(B)
```

con output

```
13   15
23   25
33   35
43   45
```

```
B=A[c(T,F),]
print(B)
```

con output

```
11   12   13   14   15   16
31   32   33   34   35   36
```

```
B=A[c(T,F),c(T,F)]
print(B)
```

con output

```
11   13   15
31   33   35
```

```
B=A[,-c(5,6)]
print(B)
```

con output

```
11   12   13   14
21   22   23   24
31   32   33   34
41   42   43   44
```

Se usiamo un filtro, otteniamo il vettore dei coefficienti della matrice che soddisfano il criterio espresso dal filtro, elencati colonna per colonna:

```
filtro=A%%4==1
print(filtro)

B=A[filtro]
print(B)
```

con output

```
F  F  T  F  F
T  F  F  F  T  F
F  F  T  F  F  F
T  F  F  F  T  F

21 41 13 33 25 45
```

Anche indici matriciali possono essere usati per assegnare valori a parti di una matrice:

```
A[1,]=c(91:96)
print(A)
A[,2]=c(81:84)
print(A)
```

con output

```
91   92   93   94   95   96
21   22   23   24   25   26
31   32   33   34   35   36
41   42   43   44   45   46

91   81   93   94   95   96
21   82   23   24   25   26
31   83   33   34   35   36
41   84   43   44   45   46
```

dimnames

La funzione `dimnames` è utile per la visualizzazione di tabelle; altrimenti nella riga dei titoli e nella colonna dei nomi delle righe apparirebbero solo gli indici. Infatti con

```
a=Mm(c(2248,20000,1587,2580,
  411,8900),col=2,pc=T)
print (a)
```

otteniamo l'output

```
      [,1] [,2]
[1,]  2248 2580
[2,] 20000  411
[3,]  1587 8900
```

che diventa più leggibile se aggiungiamo i significati dei numeri:

```
a=Mm(c(2248,20000,1587,2580,
  411,8900),col=2,pc=T)
dimnames(a)=list(c("Arabia saud.",
  "Mongolia","Svezia"),
  c("sup/kmq","ab/1000"))
print (a)
```

con output

```
             sup/kmq ab/1000
Arabia saud.    2248    2580
Mongolia       20000     411
Svezia          1587    8900
```

Si noti che nella lista vengono prima indicati i nomi delle righe, poi quelli delle colonne.

length, dim, ncol ed nrow

`length(v)` è la lunghezza di un vettore (pagina 7) o di una matrice. Nel caso di una matrice questa viene considerata come vettore, la lunghezza è quindi uguale ad nm, se la matrice possiede n righe ed m colonne.

`dim(A)` è allora il vettore (n, m).

`ncol(A)` restituisce il numero delle colonne, `nrow(A)` il numero delle righe di `A`.

Un po' a sorpresa, se `v` è un vettore, `dim(v)`, `ncol(v)` e `nrow(v)` sono tutti e tre uguali a `NULL`; ciò mostra che in R un vettore non viene automaticamente considerato come una matrice (a una riga o una colonna).

```
v=1:9
A=Mm(v,col=1)
print(identical(v,A))
# FALSE

B=Mm(v,righe=1)
print(identical(v,B))
# FALSE
```

head e tail per matrici

Le prime tre righe di una matrice `A` si ottengono con `head(A,3)` oppure con `A[1:3,]`, le ultime tre righe con `tail(A,3)` opppure con `A[(nrow(A)-2):nrow(A),]`. Cfr. pagina 9.

drop

Quando scegliamo solo una riga o una colonna di una matrice, queste vengono automaticamente convertite in un vettore. Talvolta però si vorrebbe poter considerare quella riga o colonna come matrice; per fare ciò bisogna aggiungere l'opzione `drop=F` all'indice.

Esempio:

```
dati=c(11:14,21:24,31:34)

A=Mm(dati,righe=3)
print(A)
cat('------------------------\n')

v=A[1,]
print(v)
cat('------------------------\n')

v=A[1,,drop=F]
print(v)
```

con output, stavolta completo, così come appare sullo schermo,

```
     [,1] [,2] [,3] [,4]
[1,]   11   12   13   14
[2,]   21   22   23   24
[3,]   31   32   33   34
------------------------
[1] 11 12 13 14
------------------------
     [,1] [,2] [,3] [,4]
[1,]   11   12   13   14
```

Talvolta un'operazione matriciale restituisce un singolo numero considerato però come matrice 1×1; in tal caso mediante la funzione `drop` possiamo ottenere il numero corrispondente. Ciò avviene per `%*%` e `crossprod` quando entrambi gli argomenti sono vettori; cfr. la funzione `Mv.scalare` a pagina 15.

rbind e cbind

Possiamo unire più righe con `rbind` e più colonne con `cbind`; le righe o colonne possono essere date singolarmente o anche come matrici. Esempi:

```
dati=c(11:13,21:23,31:33)
altridati=c(14:15,24:25,34:35)

A=Mm(dati,righe=3)
print(A)
B=Mm(altridati,righe=3)
print(B)
C=cbind(A,B)
print(C)
```

con output

```
11   12   13
21   22   23
31   32   33

14   15
24   25
34   35

11   12   13   14   15
21   22   23   24   25
31   32   33   34   35
```

---

e, continuando con la stessa matrice `C`,

```
D=rbind(C,81:85,91:95)
print(D)
```

con output

```
11   12   13   14   15
21   22   23   24   25
31   32   33   34   35
81   82   83   84   85
91   92   93   94   95
```

---

outer

Questa funzione crea da due successioni finite $(x_1, \ldots, x_n)$ e $(y_1, \ldots, y_m)$ e una funzione f la matrice

$$\begin{pmatrix} f(x_1,y_1) & \ldots & f(x_1,y_m) \\ \vdots & & \vdots \\ f(x_n,y_1) & \ldots & f(x_n,y_m) \end{pmatrix}$$

In R a ciò corrisponde l'espressione `outer(x,y,f)`.

Quando l'argomento `f` manca, viene usata la moltiplicazione. Quindi `outer(x,y)` è la matrice

$$\begin{pmatrix} x_1y_1 & \ldots & x_1y_m \\ \vdots & & \vdots \\ x_ny_1 & \ldots & x_ny_m \end{pmatrix}$$

`outer` può essere usata in molti modi, non solo per impostare la matrice dei valori per curve di livello, ma ad esempio anche per creare la tabella di moltiplicazione per un'operazione binaria.

Consideriamo l'insieme

$$\mathbb{Z}/6 := \{0,1,2,3,4,5\}$$

con l'addizione modulo 6; ciò significa che introduciamo per gli elementi di questo insieme un'operazione $\oplus$ definita da

$$a \oplus b := (a+b) \mod 6$$

dove $n \mod 6$ è il resto di n modulo 6.

Per vedere tutti i risultati che otteniamo in questo modo definiamo la tabella di moltiplicazione (in questo caso la tabella delle somme modulo 6) con le seguenti istruzioni in R:

```
smod6 = function (x,y) (x+y)%%6
x=seq(0,5)
tabella=outer(x,x,smod6)
dimnames(tabella)=list(0:5,0:5)
print(tabella)
```

con output

```
  0 1 2 3 4 5
0 0 1 2 3 4 5
1 1 2 3 4 5 0
2 2 3 4 5 0 1
3 3 4 5 0 1 2
4 4 5 0 1 2 3
5 5 0 1 2 3 4
```

Nello stesso modo possiamo definire un prodotto modulo 6, ponendo

$$a \odot b := (ab) \mod 6$$

In R allora usiamo la funzione

```
pmod6 = function (x,y) (x*y)%%6
```

al posto di `smod`. Otteniamo per questa operazione la tabella di moltiplicazione

```
  0 1 2 3 4 5
0 0 0 0 0 0 0
1 0 1 2 3 4 5
2 0 2 4 0 2 4
3 0 3 0 3 0 3
4 0 4 2 0 4 2
5 0 5 4 3 2 1
```

Nel corso di Algebra si dimostra che $(\mathbb{Z}/6, \oplus)$ è un gruppo abeliano, $(\mathbb{Z}/6, \odot)$ è un semigruppo commutativo con unità e che le due operazioni sono legate dalla legge distributiva

$$(a \oplus b) \odot c = (a \odot c) \oplus (b \odot c)$$

La tripla $(\mathbb{Z}/6, \oplus, \odot)$ è quindi un anello commutativo con unità. Se guardiamo bene la seconda tabella, vediamo però che

$$2 \odot 3 = 0$$

benché 2 e 3 siano diversi da 0. Questo anello possiede perciò *divisori di zero*.

Il gruppo simmetrico S_3

Non è difficile imitare la funzione `outer` in situazioni più generali di quanto questa funzione preveda. Vogliamo ad esempio calcolare la tavola di moltiplicazione del gruppo S_3 delle permutazioni di $\{1,2,3\}$. Ogni elemento $g \in S_3$ può essere rappresentato nella forma

$$g = (i,j,k)$$

con $i := g(1)$, $j := g(2)$, $k := g(3)$. Non scambiare questa notazione con quella per il ciclo $(i\ j\ k)$ che scriviamo senza virgole tra gli elementi. Se f è un'altro elemento di S_3, la composizione $fg = \bigcirc_x f(g(x))$ in R può essere scritta semplicemente come `f[g]` per il modo in cui sono utilizzati gli indici vettoriali. Infatti, assimilando la notazione matematica a quella che usiamo in R, abbiamo

$$\begin{aligned} fg &= ((fg)[1], (fg)[2], (fg)[3]) \\ &= (f[g[1]], f[g[2]], f[g[3]]) = f[g] \end{aligned}$$

Quindi ad esempio

$$\begin{aligned} (1,3,2)(3,1,2) &= (1,3,2)[3,1,2] \\ &= (2,1,3) \end{aligned}$$

E infatti

$$\begin{aligned} &1 \overset{(3,1,2)}{\longrightarrow} 3 \overset{(1,3,2)}{\longrightarrow} 2 \\ &2 \overset{(3,1,2)}{\longrightarrow} 1 \overset{(1,3,2)}{\longrightarrow} 1 \\ &3 \overset{(3,1,2)}{\longrightarrow} 2 \overset{(1,3,2)}{\longrightarrow} 3 \end{aligned}$$

Per ottenere la tripla di indici che descrive un elemento di S_3 anticipiamo di nuovo la funzione `paste`:

```
prodotto = function (f,g)
paste(f[g],collapse='')
```

Adesso possiamo creare la tabella di moltiplicazione di S_3 nel modo seguente:

```
prodotto = function (f,g)
paste(f[g],collapse='')

# Singoli elementi di S3.
id   = c(1,2,3)
t12  = c(2,1,3)
t23  = c(1,3,2)
t13  = c(3,2,1)
c123 = c(2,3,1)
c132 = c(3,1,2)

S3=Mm(c(id,t12,t23,t13,c123,c132),
  col=3)

A=Mm(0,col=6,righe=6)
# Matrice 6x6 con tutti 0.

for (i in 1:6) for (j in 1:6)
A[i,j]=prodotto(S3[i,],S3[j,])

nomi=A[1,] # !!
rownames(A)=colnames(A)=nomi

print(A,quote=F)
```

con output

```
    123 213 132 321 231 312
123 123 213 132 321 231 312
213 213 123 231 312 132 321
132 132 312 123 231 321 213
321 321 231 312 123 213 132
231 231 321 213 132 312 123
312 312 132 321 213 123 231
```

Il gruppo diedrale D_8

Il gruppo delle simmetrie di un poligono piano regolare con $n \geq 3$ vertici possiede $2n$ elementi ed è detto *gruppo diedrale* di ordine $2n$ e denotato con D_{2n} (da alcuni autori anche con D_n). Convincersi che $D_6 \cong S_3$.

D_8 è il gruppo delle simmetrie del quadrato; ogni simmetria trasforma i vertici in vertici e induce quindi una permutazione dei vertici che a sua volta determina la simmetria. Fare un disegno e convincersi che D_8 consiste esattamente dei seguenti elementi:

```
1234 2341 3412 4321 # rotazioni
2143 4321 # riflessioni agli assi
1432 3214 # riflessioni alle diagonali
```

Trovare la tavola di moltiplicazione.

La classe array

R prevede anche oggetti lineari a più di 2 dimensioni (analoghi ai tensori dell'algebra multilineare della matematica) che possono essere creati con la funzione `array`:

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

La struttura di un tale oggetto può essere anche definita in un secondo tempo:

```
a=1:24
dim(a)=c(3,4,2)
```

Naturalmente adesso possiamo ottenere l'elemento con gli indici `i,j,k` con `a[i,j,k]`.

Le istruzioni

```
a=1:24
dim(a)=c(4,6)
```

sono equivalenti ad `a=matrix(1:24,nrow=4)`.

row e col

Sorprendentemente utili sono le funzioni `row` e `col` (da non confondere con `nrow` e `ncol`). Se `A` è una matrice 4×3, queste funzioni hanno i valori

$$\texttt{row(A)} = \begin{pmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \\ 4 & 4 & 4 \end{pmatrix}$$

$$\texttt{col(A)} = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix}$$

Adesso

```
A[row(A)>col(A)]
```

è il vettore

$$a_{21}, a_{31}, a_{41}, a_{32}, a_{42}, a_{43}$$

che consiste degli elementi in cui l'indice di riga è maggiore dell'indice di colonna. Con

```
A[col(A)==row(A)]
```

otteniamo gli elementi della diagonale di `A` (più semplicemente però con `diag(A)`), con

```
A[row(A)==col(A)+1]
```

gli elementi che si trovano una posizione più in basso della diagonale:

```
A=Mm(1:16,col=4)
print(A)

v=A[row(A)==col(A)+1]
print(v)
```

con output

```
 1    2    3    4
 5    6    7    8
 9   10   11   12
13   14   15   16

5 10 15
```

upper.tri e lower.tri

Sia `A` una matrice 4×4. Allora con `upper.tri(A)` si ottiene la matrice booleana

```
F T T T
F F T T
F F F T
F F F F
```

con `upper.tri(A,diag=T)` anche gli elementi nella diagonale principale vengono posti uguali a `T`:

```
T T T T
F T T T
F F T T
F F F T
```

In modo analogo opera `lower.tri`. Esempio:

```
A=Mm(rep(0,16),col=4)

B=upper.tri(A)

B[B==T]=1:6
print(B)
```

con output

```
0  1  2  4
0  0  3  5
0  0  0  6
0  0  0  0
```

Si noti che anche qui la matrice viene riempita colonna per colonna.

diag

La funzione `diag` restituisce risultati diversi a seconda degli argomenti che vengono usati.

1. Se `A` è una matrice, `diag(A)` è il vettore degli elementi diagonali di `A`. Possiamo quindi con `sum(diag(A))` ottenere (nel caso che `A` sia quadratica) la traccia di `A`, come faremo più avanti su questa stessa pagina.

2. Se `v` è un vettore di lunghezza ≥ 2, `diag(v)` è una matrice diagonale con diagonale `v`.

3. Se `n` è un numero naturale ≥ 1, allora `diag(n)` è la matrice identica $n \times n$.

Per alcune variazioni meno importanti consultare `?diag`.

`diag` può essere anche usata alla sinistra di un'assegnazione:

```
A=Mm(rep(0,16),col=4)

diag(A)=c(3,4,5,8)
print(A)
```

con output

```
3  0  0  0
0  4  0  0
0  0  5  0
0  0  0  8
```

oppure

```
A=Mm(rep(0,16),col=4)
diag(A)=5
print(A)
```

con output

```
5  0  0  0
0  5  0  0
0  0  5  0
0  0  0  5
```

Se A è una matrice $n \times n$, la matrice $A - x\delta$ può essere ottenuta con `A-x*diag(n)`:

```
A=Mm(1:16,col=4)
B=A-3*diag(4)
print(A)
print(B)
```

con output

```
 1   2   3   4
 5   6   7   8
 9  10  11  12
13  14  15  16

-2   2   3   4
 5   3   7   8
 9  10   8  12
13  14  15  13
```

Prodotto scalare e lunghezza

Nel prodotto scalare usiamo la funzione `crossprod` di R che calcola il prodotto $A^t B$ per matrici; evitiamo la possibile, ma un po' infelice abbreviazione `a%*%b` a cui abbiamo accennato a pagina 12.

`drop` trasforma una matrice 1×1 in uno scalare, come sappiamo dalla pagina 13.

```
Mv.scalare = function (a,b)
{p=crossprod(a,b); drop(p)}

Mv.lun = function (x)
sqrt(Mv.scalare(x,x))
```

det (determinante) e traccia

`det(A)` è il determinante della matrice `A`.

Per la traccia sembra che non ci sia una apposita funzione che però possiamo facilmente definire:

```
Mm.traccia = function (A)
{if (length(A)>1) sum(diag(A))
else c(A)}
```

Esempio: Con

```
A=Mm(c(5,7,1:5,8:2,1:4,4:0,8,9),col=5)
print(A)

print(det(A))
print(Mm.traccia(A))
```

otteniamo

```
5  7  1  2  3
4  5  8  7  6
5  4  3  2  1
2  3  4  4  3
2  1  0  8  9

696
26
```

Sistemi lineari con R

R permette, per matrici che devono essere quadratiche e invertibili, la risoluzione di un sistema lineare

$$Ax = b$$

mediante l'istruzione

```
x=solve(A,b)
```

Proviamo prima con il sistema

$$3x - 2y = 8$$
$$x + 6y = 5$$

```
dati=c(3,-2,1,6)
A=Mm(dati,righe=2)
v=c(8,5)
x=solve(A,v)
print(x)
```

L'output è `2.90 0.35`, in accordo con quanto si trova con la regola di Cramer, infatti

$$x = \frac{\begin{vmatrix} 8 & -2 \\ 5 & 6 \end{vmatrix}}{20} = \frac{48+10}{20} = \frac{58}{20} = 2.9$$

$$y = \frac{\begin{vmatrix} 3 & 8 \\ 1 & 5 \end{vmatrix}}{20} = \frac{15-8}{20} = \frac{7}{20} = 0.35$$

Proviamo con il sistema

$$3x + 2y - z = 10$$
$$4x - 9y + 2z = 6$$
$$x + y - 14z = 2$$

```
dati=c(3,2,-1,4,-9,2,1,1,-14)
A=Mm(dati,righe=3)
v=c(10,6,2)
x=solve(A,v)
print(round(x,4))
```

La risposta è

```
2.9305 0.6611 0.1137
```

Se vogliamo le soluzioni come numeri razionali, nell'ultima riga del programma possiamo usare

```
print(fractions(x))
```

La funzione `fractions` richiede però la libreria *MASS*, che dobbiamo caricare con

```
library(MASS)
```

Otteniamo allora l'output

```
1392/475 314/475 54/475
```

L'inversa di una matrice non singolare `A` è data da

```
solve(A)
```

Per verificare se la matrice è invertibile si può calcolare il determinante con `det(A)`.

La libreria *MASS* contiene la funzione `ginv` che calcola l'inversa generalizzata nel senso di Penrose di matrici non necessariamente quadratiche.

Autovalori

Situazione 16.1. K sia un campo ed $A \in K_n^n$ una matrice quadratica.

Teorema 16.2. *Consideriamo l'applicazione*

$$\varphi = \bigcirc_x Ax : K^n \longrightarrow K^n$$

Allora i seguenti enunciati sono equivalenti:

(1) φ *non è biiettiva.*
(2) φ *non è suriettiva.*
(3) φ *non è iniettiva.*
(4) $\det A = 0$.

È chiaro che la condizione (3) si verifica se e solo se esiste un vettore $x \neq 0$ *in* K^n *tale che* $Ax = 0$.

Dimostrazione. Corsi di Geometria.

Definizione 16.3. Un vettore $x \in K^n$ si chiama un *autovettore* di A, se $x \neq 0$ e se esiste un elemento $\lambda \in K$ tale che $Ax = \lambda x$.
Ciò accade se e solo se $(A - \lambda\delta)x = 0$.
Denotiamo qui, come altre volte, con δ la matrice identica.

Definizione 16.4. Un elemento $\lambda \in K$ si chiama un *autovalore* di A, se esiste un vettore $x \in K^n$ con $x \neq 0$ e $Ax = \lambda x$.
In tal caso diciamo anche che x è un autovettore di A per l'autovalore λ e che λ è un autovalore di A per l'autovettore x.

Osservazione 16.5. *Sia* $\lambda \in K$. *Allora* λ *è un autovalore di* A *se e solo se*

$$\det(A - \lambda\delta) = 0.$$

In R gli autovalori di una matrice reale o complessa si trovano con la funzione `eigen`. Essa calcola, se non si pone l'opzione

```
only.values=T
```

anche un sistema di autovettori; ciò può rallentare il calcolo per matrici molto grandi. Il risultato è una *lista* in R, un concetto che tratteremo fra poco, e le componenti si ottengono con la sintassi `$values` e `$vectors`. Creiamo due funzioni per la nostra libreria:

```
Mm.autovalori = function (A)
{eigen(A,only.values=T)$values}

Mm.autovettori = function (A)
{eigen(A)$vectors}
```

Calcoliamo gli autovalori delle matrici

$$\begin{pmatrix} 6 & -2 \\ 2 & 6 \end{pmatrix} \quad \text{e} \quad \begin{pmatrix} 6 & 2 \\ 2 & 6 \end{pmatrix}$$

con

```
A=Mm(c(6,-2,2,6),col=2)
lambda=Mm.autovalori(A)
print(round(lambda,2))

A=Mm(c(6,2,2,6),col=2)
lambda=Mm.autovalori(A)
print(round(lambda,2))
```

trovando `6+2i 6-2i` per la prima matrice e `8 4` per la seconda; quest'ultima è reale e simmetrica e possiede quindi solo autovalori reali.

I cerchi di Gershgorin

Per una matrice reale o complessa 2×2

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

definiamo i cerchi

$$C_1 := \{z \in \mathbb{C} \mid |z - a| \leq |b|\}$$
$$C_2 := \{z \in \mathbb{C} \mid |z - d| \leq |c|\}$$

similmente per una matrice 3×3

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$$

poniamo

$$C_1 := \{z \in \mathbb{C} \mid |z - a_{11}| \leq |a_{12}| + |a_{13}|\}$$
$$C_2 := \{z \in \mathbb{C} \mid |z - a_{22}| \leq |a_{21}| + |a_{23}|\}$$
$$C_3 := \{z \in \mathbb{C} \mid |z - a_{33}| \leq |a_{31}| + |a_{32}|\}$$

Questi cerchi, che si definiscono in modo analogo per le dimensioni superiori, si chiamano i *cerchi di Gershgorin* di A.

I centri dei cerchi di Gershgorin sono gli elementi nella diagonale principale della matrice; se la matrice è reale o se almeno gli elementi nella diagonale sono tutti reali, i centri dei cerchi di Gershgorin si troveranno quindi sull'asse reale.

Teorema 16.6 di Gershgorin. *Gli autovalori di* A *sono tutti contenuti nell'unione dei suoi cerchi di Gershgorin.*

Dimostrazione. Corsi di Analisi numerica. Non difficile.

Definiamo una funzione che restituisce il centro e il raggio dell'`i`-esimo cerchio di Gershgorin della matrice `A`:

```
Mm.gershgorin = function (A,i)
{centro=A[i,i]
raggio=sum(abs(A[i,]))-abs(centro)
c(centro,raggio)}
```

Calcolare i cerchi di Gershgorin di

$$A = \begin{pmatrix} 1 & 4 & 1 & 3 \\ 2 & -1 & 4 & 0 \\ 3 & 2 & 5 & -2 \\ 4 & -3 & 1 & 8 \end{pmatrix}$$

con

```
dati=c(1,4,1,3,2,-1,4,0,
  3,2,5,-2,4,-3,1,8)
A=Mm(dati,righe=4)

for (i in 1:4)
  print(Mm.gershgorin(A,i))
# 1 8 | -1 6 | 5 7 | 8 8
```

FONDAMENTI DI INFORMATICA

Corso di laurea in matematica | Anno accademico 2004/05 | Numero 5

do.call

R possiede alcuni meccanismi molto potenti per la trasformazione degli argomenti di una funzione. Assumiamo che una funzione `f` sia definita per gli argomenti `x`, `y` e `z`,

```
f = function (x,y,z)
***
```

che però nel nostro programma `x,y,z` non appaiano direttamente come valori singoli, ma in forma di una lista `a`. Allora possiamo eseguire la funzione con il comando

```
do.call(f,a)
```

Si noti che il primo argomento deve essere il nome della funzione (prima della versione 2.1.0 doveva essere una stringa che conteneva quel nome), il secondo una *lista*. Se `x,y,z` sono disponibili sotto forma di un vettore `v`, dobbiamo usare

```
do.call(f,as(v,'list'))
```

Esempio:

```
f = function (x,y,z)
x+y+z

a=list(1,4,12)
s=do.call(f,a)
print(s) # 17

v=c(3,7,9)
s=do.call(f,as(v,'list'))
print(s) # 19
```

Per poter usare anche vettori di argomenti senza conversione a una lista (ciò ha senso naturalmente solo se si tratta di argomenti dello stesso tipo), definiamo una nostra funzione

```
P.esegui = function (f,a)
do.call(f,as(a,'list'))
```

che può essere usata nel modo seguente:

```
f = function (x,y,z) x+y+z

x=P.esegui(f,list(1,4,12))
print(x) # 17

x=P.esegui(f,c(3,7,9))
print(x) # 19
```

oppure, con la stessa `f`,

```
mmm = function (a)
c(min(a),mean(a),max(a))

x=P.esegui(f,mmm(1:10))
print(x) #
print(mmm(1:10)) # 1 5.5 10
```

In alcuni casi `do.call` non funziona correttamente, ad esempio

```
x = do.call(mean,as(1:10,'list'))
print(x) # 1 - non corretto.
```

expression ed eval

R permette di creare *espressioni formali* con la funzione `expression`. Mentre quindi dopo

```
formula=expression(x+3)
```

`formula` è un'espressione formale il cui valore non viene però calcolato (infatti questa espressione è lecita anche quando `x` non è ancora stato definito), con

```
valore=eval(formula)
```

ne possiamo calcolare il valore. Esempio:

```
formula=expression(x+3)
x=10
u=eval(formula)

print(formula)
print(u)
```

con output

```
expression(x + 3)
[1] 13
```

Stringhe possono essere interpretate come comandi anche secondo il seguente schema:

```
u='x=5; y=6; z=x*y+1'
com=parse(text=u)
eval(com)
print(z) # 31
```

ma il metodo più generale e forse più facilmente comprensibile è l'utilizzo di una connessione di testo (cioè di una stringa che viene usata come un file di testo) combinato con `source`:

```
u='x=4; y=6; z=x*y+1'
conn=textConnection(u)
source(conn); close(conn)
print(z) # 25
```

`parse` e soprattutto `textConnection` sono però piuttosto lenti nell'esecuzione, come illustrato nella terza colonna, e quindi non vengono usate nelle operazioni comuni che devono essere ripetute molte volte.

Le funzioni `substitute` e `deparse` sono piuttosto versatili e generali; la prima restituisce un'espressione dopo aver effettuato delle sostituzioni per le quali, nel caso generale, bisogna consultare `?substitute`, la seconda trasforma un'espressione in una stringa. L'utilizzo più semplice è l'uso in coppia:

```
f = function (x)
{nomex=deparse(substitute(x))
cat(nomex,'^2 = ',x^2,'\n',sep='')}

f(7)
# 7^2 = 49

s=10
f(s)
# s^2 = 100
```

Come si vede, in questo modo siamo in grado di ricavare nell'output anche il nome usato per l'argomento della funzione.

In questo numero

system.time

Possiamo usare `system.time` per verificare la lentezza di `textConnection`; dei 5 tempi che vengono indicati (in secondi) bisogna guardare soprattutto il terzo (tempo trascorso) e il primo (tempo di CPU usato dall'utente, cfr. `?system.time`):

```
f= function (n)
{u='x=5; y=6; z=x*y+1'
for (i in 1:n)
{com=parse(text=u)
eval(com)}}

print(system.time(f(1000)))
# 0.13 0 0.13 0 0

g = function (n)
{u='x=4; y=6; z=x*y+1'
for (i in 1:n)
{conn=textConnection(u)
source(conn); close(conn)}}

print(system.time(g(1000)))
# 1.83 0.01 1.83 0 0
```

formals e body

Con `formals` si ottiene una lista degli argomenti di una funzione che comprende i valori preimpostati; `body` restituisce il corpo di una funzione. Esempio:

```
f = function (x,y=1)
{z=x+y^2; print(z)}

arg=formals(f)
print(arg)
# Output semplificato:
# $x
# $y
# 1

corpo=body(f)
print(corpo)
# {z = x + y^2
# print(z)}

x=7; y=2
eval(corpo)
# 11
```

on.exit e system

All'interno di una funzione `on.exit(istr)` fa in modo che `istr` venga eseguita quando la funzione termina. Ciò può essere usato in funzioni molto ramificate per non dover ripetere un'istruzione (in C si userebbe `goto fine;`) oppure per ripristinare parametri grafici o di sistema modificati dalla funzione.

Sotto Linux un comando `istr` della shell può essere eseguito dall'interno di un programma mediante `system(istr)`. L'output del comando appare sul terminale oppure, con l'opzione `intern=T`, diventa il risultato della funzione. Quindi

```
u=system('ls -l',intern=T)
```

fa in modo che `u` sia un vettore di stringhe, ognuna delle quali contiene le informazioni riguardanti un file contenuto nella cartella di lavoro. Forse `system` funziona anche sotto Windows. Sotto Linux sono anche previste le funzioni `pipe` e `fifo` per la creazione di *pipelines*.

Abbiamo già imparato alle pagine 3 e 4 come definire una funzione in R. Quando si batte il nome della funzione senza le parentesi, si ottiene il codice sorgente; ciò può essere utile per una veloce visualizzazione ad esempio per ricordare dei parametri, senza dover aprire il file nel nella libreria che contiene la funzione.

Funzioni e nomi di funzioni possono essere usati anche in vettori, ad esempio

```
for (f in c(sum,prod,mean))
print(f(1:10))
```

con output

```
55
3628800
5.5
```

Lo schema di Horner

Sia dato un polinomio

$$f = a_0 x^n + a_1 x^{n-1} + \cdots + a_n \in A[x]$$

dove A è un qualsiasi anello commutativo.

Per $\alpha \in A$ vogliamo calcolare $f(\alpha)$. Sia ad esempio $f = 3x^4+5x^3+6x^2+8x+17$. Poniamo

$$\begin{aligned}
b_0 &= 3 \\
b_1 &= b_0\alpha + 5 &&= 3\alpha + 5 \\
b_2 &= b_1\alpha + 6 &&= 3\alpha^2 + 5\alpha + 6 \\
b_3 &= b_2\alpha + 8 &&= 3\alpha^3 + 5\alpha^2 + 6\alpha + 8 \\
b_4 &= b_3\alpha + 17 &&= 3\alpha^4 + 5\alpha^3 + 6\alpha^2 + 8\alpha + 17
\end{aligned}$$

e vediamo che $b_4 = f(\alpha)$. Lo stesso si può fare nel caso generale:

$$\begin{aligned}
b_0 &= a_0 \\
b_1 &= b_0\alpha + a_1 \\
&\cdots \\
b_k &= b_{k-1}\alpha + a_k \\
&\cdots \\
b_n &= b_{n-1}\alpha + a_n
\end{aligned}$$

con $b_n = f(\alpha)$, come dimostriamo adesso. Consideriamo il polinomio

$$g := b_0 x^{n-1} + b_1 x^{n-2} + \cdots + b_{n-1}.$$

Allora, usando che $\alpha b_k = b_{k+1} - a_{k+1}$ per $k = 0, \ldots, n-1$, abbiamo

$$\begin{aligned}
\alpha g &= \alpha b_0 x^{n-1} + \alpha b_1 x^{n-2} + \cdots + \alpha b_{n-1} \\
&= (b_1 - a_1)x^{n-1} + (b_2 - a_2)x^{n-2} + \ldots \\
&\quad + (b_{n-1} - a_{n-1})x + b_n - a_n \\
&= (b_1 x^{n-1} + b_2 x^{n-2} + \cdots + b_{n-1}x + b_n) \\
&\quad - (a_1 x^{n-1} + a_2 x^{n-2} + \ldots a_{n-1}x + a_n) \\
&= x(g - b_0 x^{n-1}) + b_n - (f - a_0 x^n) \\
&= xg - b_0 x^n + b_n - f + a_0 x^n \\
&= xg + b_n - f
\end{aligned}$$

quindi

$$\boxed{f = (x - \alpha)g + b_n}$$

e ciò implica $f(\alpha) = b_n$.

$b_0, \ldots, b_{n-1}$ sono perciò i coefficienti del quoziente nella divisione con resto di f per $x - \alpha$, mentre b_n è il resto, uguale a $f(\alpha)$.

Questo algoritmo è detto *schema di Horner* o *schema di Ruffini* ed è molto più veloce del calcolo separato delle potenze di α (tranne nel caso che il polinomio consista di una sola o di pochissime potenze, in cui si userà invece l'algoritmo del contadino russo (pagina 16 del corso di Programmazione) oppure, in R, semplicemente l'operazione `x^n`.

Traduciamo l'algoritmo di Horner in una funzione in R; in essa il parametro alfa è preimpostato a 2.

```
M.horner = function (v,alfa=2)
{b=v[1]
for (i in 2:(length(v)))
b=b*alfa+v[i]; b}
```

Lo schema di Horner permette una elegante versione ricorsiva, descritta dalla relazione

$$\begin{aligned}
&\mathrm{val}(\alpha, a_0, a_1, \ldots, a_n) \\
&\quad = \alpha \cdot \mathrm{val}(\alpha, a_0, a_1, \ldots, a_{n-1}) + a_n
\end{aligned}$$

(dimostrata a pagina 16 del corso di Programmazione), meno efficiente della forma iterativa però per la ripetuta creazione dei vettori `v[1:n]`:

```
hornric = function (v, alfa=2)
{n=length(v)-1; if (n==0) v
else alfa*hornric(v[1:n],alfa)+v[n+1]}
```

Una frequente applicazione dello schema di Horner è il calcolo del valore corrispondente a una rappresentazione binaria o esadecimale.

Infatti otteniamo $(1,0,0,1,1,0,1,1,1)_2$ come

```
M.horner(c(1,0,0,1,1,0,1,1,1))
```

e $(A,F,7,3,0,5,E)_{16}$ come

```
M.horner(c(10,15,7,3,0,5,14),alfa=16):

x=M.horner(c(1,0,0,1,1,0,1,1,1))
print(x) # 311

y=M.horner(c(10,15,7,3,0,5,14),alfa=16)
print(y) # 183971934
```

Recall

Nelle funzioni ricorsive la funzione chiamante se stessa può essere indicata con `Recall`; possiamo così scrivere `hornric` nel modo seguente:

```
hornric = function (v, alfa=2)
{n=length(v)-1; if (n==0) v
else alfa*Recall(v[1:n],alfa)+v[n+1]}
```

Rappresentazione binaria

Ogni numero naturale n possiede una rappresentazione binaria, cioè una rappresentazione della forma

$$n = a_k 2^k + a_{k-1} 2^{k-1} + \cdots + a_1 2 + a_0$$

con coefficienti (o cifre binarie) $a_i \in \{0,1\}$. Per $n = 0$ usiamo $k = 0$ ed $a_0 = 0$; per $n > 0$ chiediamo che $a_k \neq 0$. Con queste condizioni k e gli a_i sono univocamente determinati. Sia $r_2(n) = (a_k, \ldots, a_0)$ il vettore i cui elementi sono queste cifre. Dalla rappresentazione binaria si deduce la seguente relazione ricorsiva:

$$r_2(n) = \begin{cases} (n) & \text{se } n \leq 1 \\ (r_2(\frac{n}{2}), 0) & \text{se } n \text{ è pari} \\ (r_2(\frac{n-1}{2}), 1) & \text{se } n \text{ è dispari} \end{cases}$$

La funzione `Ma.rapp2` che definiamo accetta un secondo parametro facoltativo `cifre`; quando questo è maggiore del numero di cifre necessarie per la rappresentazione binaria di n, i posti iniziali vuoti vengono riempiti con zeri usando la funzione `rep` di R.

```
Ma.rapp2 = function (n, cifre=NA)
{if (n<=1) v=n
else if (n%%2==0) v=c(Recall(n/2),0)
else v=c(Recall((n-1)/2),1)
if (missing(cifre)) v else
{n=length(v);
if (n>=cifre) v
else c(rep(0,cifre-n),v)}}
```

Per provare la funzione scriviamo in *programma* queste istruzioni:

```
for (n in c(0:10,seq(19,240,29)))
cat(sprintf('%3d',n),' ',
  Ma.rapp2(n,cifre=8),'\n')
```

Otteniamo l'output

```
  0   0 0 0 0 0 0 0 0
  1   0 0 0 0 0 0 0 1
  2   0 0 0 0 0 0 1 0
  3   0 0 0 0 0 0 1 1
  4   0 0 0 0 0 1 0 0
  5   0 0 0 0 0 1 0 1
  6   0 0 0 0 0 1 1 0
  7   0 0 0 0 0 1 1 1
  8   0 0 0 0 1 0 0 0
  9   0 0 0 0 1 0 0 1
 10   0 0 0 0 1 0 1 0
 19   0 0 0 1 0 0 1 1
 48   0 0 1 1 0 0 0 0
 77   0 1 0 0 1 1 0 1
106   0 1 1 0 1 0 1 0
135   1 0 0 0 0 1 1 1
164   1 0 1 0 0 1 0 0
193   1 1 0 0 0 0 0 1
222   1 1 0 1 1 1 1 0
```

Se nella prima riga scriviamo

```
for (n in 0:255)
```

otteniamo i vertici dell'ipercubo 2^8.

L'ipercubo

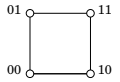
Sia $X = \{1, \ldots, n\}$ con $n \geq 0$.

Identificando $\mathcal{P}(X)$ con 2^n, geometricamente otteniamo un *ipercubo* che può essere visualizzato nel modo seguente.

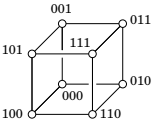
$n = 0$:

$n = 1$:

$n = 2$:



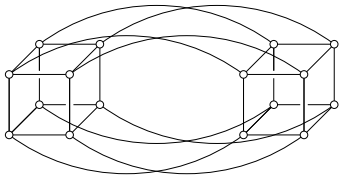
$n = 3$:



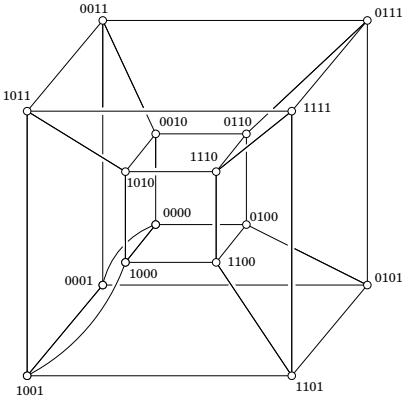
$n = 4$: L'ipercubo a 4 dimensioni si ottiene dal cubo 3-dimensionale attraverso la relazione

$$2^4 = 2^3 \times \{0, 1\}$$

Dobbiamo quindi creare due copie del cubo 3-dimensionale. Nella rappresentazione grafica inoltre sono adiacenti e quindi connessi con una linea quei vertici che si distinguono in una sola coordinata. Oltre ai legami all'interno dei due cubi dobbiamo perciò unire i punti $(x, y, z, 0)$ e $(x, y, z, 1)$ per ogni x, y, z.



La figura diventa molto più semplice, se si pone uno dei due cubi (quello con la quarta coordinata $= 0$) all'interno dell'altro (quello con la quarta coordinata $= 1$):



$n \geq 5$: Teoricamente anche qui si può usare la relazione

$$2^n = 2^{n-1} \times \{0, 1\}$$

ma la visualizzazione diventa difficoltosa.

Ogni vertice dell'ipercubo corrisponde a un elemento di 2^n che nell'interpretazione insiemistica rappresenta a sua volta un sottoinsieme di X (il punto 0101 ad esempio l'insieme $\{2, 4\}$ se $X = \{1, 2, 3, 4\}$).

La distanza di Hamming

La distanza di Hamming è definita come il numero delle coordinate in cui due elementi di 2^n differiscono e che in R può essere calcolata con la seguente semplicissima funzione:

```
Mfb.hamm = function (u,v)
sum(abs(u-v))
```

---

lapply ed sapply

Siano $x = (x_1, \ldots, x_n)$ un vettore o una lista ed f una funzione che non operi già in modo vettoriale. Allora con `lapply(x,f)` otteniamo la sequenza $(f(x_1), \ldots, f(x_n))$ in forma di una lista, con `sapply(x,f)` la stessa sequenza in forma di un vettore quando ciò è possibile.

Se la `f` ha argomenti addizionali, questi possono essere aggiunti ai parametri di `lapply` ed `sapply`, quindi il risultato di `sapply(x,f,a,b)` diventa il vettore $(f(x_1, a, b), \ldots, f(x_n, a, b))$. Talvolta è necessario applicare l'operatore `c` al risultato. Esempio:

```
f = function (x) x^2
u=sapply(2:5,f)
print(u)
# output: 4 9 16 25

f=function (x,rho=1) x^2+rho
u=sapply(2:5,f,rho=2)
print(u)
# output: 6 11 18 27
```

Le funzioni `lapply` e `sapply` sono caratteristiche di molti linguaggi ad alto livello e già presenti nel più antico di tutti, il *Lisp*. Nel *Perl* esse corrispondono alla funzione `map`.

La funzione `Mfb.cubo` restituisce l'ipercubo in forma di una lista:

```
Mfb.cubo = function (n)
lapply(0:(2^n-1),Ma.rapp2,cifre=n)
```

Non dimenticare le parentesi attorno a `2^n-1`.

Sistemi di Lindenmayer

Sia A un insieme. Il *monoide libero* generato da A è l'insieme

$$A^* := \bigcup_{n=0}^{\infty} A^n$$

Denotiamo con ε l'unico elemento di A^0. Poniamo $A^+ := A^* \setminus \{\varepsilon\}$.

Gli elementi di A^* si chiamano *parole* sull'alfabeto A e siccome $A^n \cap A^k = \emptyset$ per $n \neq k$, per ogni $v \in A^*$ esiste esattamente un $n \in \mathbb{N}$ tale che $v \in A^n$; questo n si chiama la *lunghezza* di v e viene denotato con $|v|$. In particolare $|\varepsilon| = 0$; ε si chiama la parola vuota.

Se usiamo la concatenazione di parole come composizione, A^* diventa un monoide, altamente noncommutativo.

T sia un monoide e $\varphi_0 : A \longrightarrow T$ un'applicazione qualsiasi. Allora esiste esiste un unico omomorfismo di monoidi $\varphi : A^* \longrightarrow T$ che su A coincide con φ_0.

Questo teorema fondamentale è stato dimostrato nel corso di Algoritmi.

Gli endomorfismi di A^* (cioè gli omomorfismi di monoidi $A^* \longrightarrow A^*$) sono noti anche come *sistemi di Lindenmayer*.

Aristid Lindenmayer (1925-1989) era un botanico olandese che utilizzò questi sistemi per descrivere (soprattutto in modo grafico) ed analizzare l'accrescimento di piante. Un endomorfismo di A^* può essere considerato come un meccanismo di *riscrittura*; l'idea è di imitare un principio generale della natura:

organismo semplice iniziale + leggi

↓

organismo complesso

Un semplicissimo, quanto antico, esempio di riscrittura è il *fiocco di neve* di Koch (1905), discusso nel corso di Algoritmi: Si parte da un elemento *iniziatore*, il triangolo equilatero, e da un elemento *generatore* (che contiene le leggi), costituita da una linea spezzata orientata che consta di quattro parti della stessa lunghezza; quindi si sostituisce ogni lato del triangolo iniziatore con una riga del generatore, ridotta in modo tale (se si vuole che lo spazio occupato dalla figura rimanga lo stesso) da avere gli estremi coincidenti con quelli del segmento da sostituire. Iterando questo procedimento si perviene ad un'immagine che assomiglia a un fiocco di neve. Per il disegno sul calcolatore naturalmente bisognerà fermarsi dopo un certo numero di iterazioni, matematicamente si può anche considerare il limite delle figure ottenute, ad esempio rispetto a una metrica (*metrica di Hausdorff*) sull'insieme dei sottoinsiemi compatti non vuoti di $\mathbb{R}^2$.

È importante che nei sistemi di Lindenmayer la riscrittura avviene *in parallelo*, cioè le regole vengono applicate simultaneamente ad ogni carattere di una data parola, a differenza da quanto accada nei linguaggi di Chomsky (usati spesso per descrivere i linguaggi di programmazione).

Elenchiamo alcune delle principali applicazioni dei sistemi di Lindenmayer.

Da un lato questi sistemi possono essere impiegati per simulare l'accrescimento di un organismo o di un intero sistema ecologico e per analizzarne i meccanismi di crescita. Si possono così individuare i parametri che determinano l'evoluzione di un organismo o di un ecosistema.

Due campi dove più intensamente si impiegano piante virtuali sono il cinema e i giochi al calcolatore, dove vengono usate in scene esterne, in effetti speciali, nella simulazione di paesaggi che possono essere esplorati interattivamente.

Sistemi di Lindenmayer possono essere usati per la memorizzazione economica di immagini. Infatti, invece di dover conservare tutto il contenuto di una parte intera dello schermo (ad esempio 600x600 pixel = 360000 bit = 45000 byte per un'immagine in bianconero) è sufficiente conservare la stringa che rappresenta l'iniziatore (ad esempio 50 byte) e le stringhe che contengono le leggi di crescita (ad esempio 20x30 byte = 600 byte), quindi in tutto 650 invece di 45000 byte.

La successione di Morse

Questa successione è forse il più noto esempio di un sistema di Lindenmayer. Essa compare sotto molte vesti nella *dinamica simbolica* (lo studio delle periodicità e quasiperiodicità di parole infinite, cioè di elementi di $A^{\mathbb{N}}$ o $A^{\mathbb{Z}}$). Infatti la successione di Morse è la più semplice successione quasiperiodica, ma non periodica. Essa è definita nel modo seguente:

$A = \{0, 1\}$
generatore: 0
leggi: $0 \longmapsto 01, 1 \longmapsto 10$

Quindi la successione si sviluppa in questo modo:

0
01
0110
01101001
0110100110010110
...

Si vede che la successione può essere generata anche in altri modi, ad esempio aggiungendo alla successione ottenuta al passo precedente la successione che si ottiene da essa scambiando 1 con 0. Vediamo in particolare che la successione si allunga sempre senza mai cambiare nelle parti costruite negli stadi precedenti.

La dinamica simbolica viene classicamente e ancora oggi utilizzata nello studio di sistemi dinamici. Immaginiamo infatti un punto che si muove in uno spazio X in tempi discreti, raggiungendo le posizioni $x_0, x_1, x_2, \ldots$. Assumiamo che sia data una partizione $X = U \dot\cup V$ di X e che

$x_0 \in U \quad x_1 \in U \quad x_2 \in V \quad x_3 \in U$
$x_4 \in V \quad x_5 \in U \quad x_6 \in U \quad x_7 \in V$
$x_8 \in V \quad x_9 \in V \quad x_{10} \in V \quad x_{11} \in V$
...

Allora possiamo associare a questo movimento la successione

$UUVUVUUVUUVV\ldots$

o, più brevemente,

001010010011...

che fornisce già alcune indicazioni sul movimento. Potremmo adesso raffinare la partizione (lavorando con più sottoinsiemi e quindi con più lettere nel nostro alfabeto) per ottenere rappresentazioni sempre più fedeli del nostro sistema dinamico. Questa tecnica è molto utilizzata in vari campi della matematica pura e della fisica statistica.

Un ramo di applicazione più recente e interessante della dinamica simbolica è l'analisi dei testi, ad esempi in informatica e bioinformatica. In questo caso parole finite vengono studiate come parti di parole infinite a cui si possono applicare i metodi della dinamica simbolica.

La funzione Mlin

È facilissimo, ma istruttivo, creare una funzione per la realizzazione di sistemi di Lindenmayer. Nell'esempio ci limitiamo al caso $A = \{0, 1\}$.

Rappresentiamo le parole di A^* come stringhe che contengono solo le lettere `'0'` e `'1'`. La funzione in R è semplicemente

```
Mlin = function (a,f) c(sapply(a,f))
```

Per la successione di Morse `f` è ad esempio

```
Mlin.morse = function (x)
P.quale(x,0,c(0,1),1,c(1,0))
```

Per provare usiamo

```
u=0
for (k in 1:6)
{cat(u,'\n',sep='')
u=Mlin(u,Mlin.morse)}
```

con output

```
0
01
0110
01101001
0110100110010110
0110100110010110100101100110100 1
```

Alcune funzioni per matrici

Con le seguenti funzioni possiamo ottenere le liste delle colonne e delle righe di una matrice.

```
Mm.col = function (A)
lapply(1:ncol(A),
  function (j) A[,j])

Mm.righe = function (A)
lapply(1:nrow(A),
  function (i) A[i,])
```

Per il calcolo veloce di somme e medie delle colonne o righe di una matrice R fornisce le funzioni `rowSums`, `colSums`, `rowMeans` e `colMeans`. Ad esempio `rowMeans(A)` restituisce il vettore che contiene le medie delle righe di `A`:

```
A = Mm (1:24,col=4)
print(A)

mr = rowMeans(A)
print(mr)
```

con output

```
 1  2  3  4
 5  6  7  8
 9 10 11 12
13 14 15 16
17 18 19 20
21 22 23 24

2.5 6.5 10.5 14.5 18.5 22.5
```

La media della seconda riga si ottiene con `rowMeans(A)[2]` oppure anche con `mean(A[,2])`.

mapply

f sia una funzione di 2 variabili ed $x = (x_1, \ldots, x_n)$, $y = (y_1, \ldots, y_n)$.
Allora con `mapply(f,x,y)` otteniamo il vettore $f(x_1, y_1), \ldots, f(x_n, y_n))$.

Nello stesso modo, se g è una funzione di 3 variabili e se $z = (z_1, \ldots, z_n)$, con `mapply(g,x,y,z)` otteniamo il vettore $(g(x_1, y_1, z_1), \ldots, g(x_n, y_n, z_n))$. Gli operandi possono anche essere liste; in tal caso il risultato è una lista. La funzione che viene applicata (il primo argomento di `mapply`) può avere un numero arbitrario di argomenti.

Argomenti opzionali di f vengono aggiunti agli operandi:

```
f = function (x,y,alfa=1)
alfa*(x+y)

a=1:5
b=11:15

u=mapply(f,a,b,alfa=2)
print(u)
# 24 28 32 36 40
```

Un confronto

„R has been changing and improving so fast that it is difficult for any of the commercial alternatives to keep up. There are several reasons for this. First, R is easily extended. Second, the R Foundation for Statistical Computing has provided a supportive organizational framework that makes it easy for people to share. Third, there are hundreds and perhaps thousands of competent professionals the world over who have been frustrated in the past by the steep price of commercial software for many things, and R provides a shockingly easy and open alternative that helps people share their latest developments with the entire world in a way that replaces that frustration with the pride of contributing to something incredibly useful.“
(Spencer Graves)

John Chambers sarà il primo membro emerito tra gli scienziati dei Bell Laboratories a Murray Hill.

Bibliografia

16035 **O. Deussen:** Computergenerierte Pflanzen. Springer 2003.

850 **H. Furstenberg:** Recurrence in ergodic theory and combinatorial number theory. Princeton UP 1981.

14258 **W. Gottschalk/G. Hedlund:** Topological dynamics. AMS 1968.

17057 **U. Ligges:** Programmieren mit R. Springer 2005.

2226 **P. Prusinkiewicz/A. Lindenmayer:** The algorithmic beauty of plants. Springer 1990.

1188 **G. Rozenberg/A. Salomaa:** The mathematical theory of L systems. Academic Press 1980.

17060 **P. Spector:** An introduction to S and S-Plus. Wadsworth 1994.

17117 **W. Venables:** Mind your language. R News 2/2 (2002), 24-26.

15605 **W. Venables/B. Ripley:** S programming. Springer 2000.

FONDAMENTI DI INFORMATICA

Corso di laurea in matematica | Anno accademico 2004/05 | Numero 6

apply

Questa funzione è fondamentale per la trasformazione di righe o colonne di una matrice. Siano `f` una funzione definita per vettori (a valori non necessariamente numerici) che per ogni argomento restituisca un vettore della stessa lunghezza ≥ 1 ed `A` una matrice (a valori non necessariamente numerici). Allora

```
t(apply(A,1,f,***))
```

è la matrice che si ottiene da `A` eseguendo `f` su ogni riga di `A`, ed

```
apply(A,2,f,***)
```

è la matrice che si ottiene eseguendo `f` su ogni colonna di `A`. In entrambi i casi `***` indica eventuali ulteriori argomenti di `f`.

Esempi:

```
A = Mm(c(1:4,2:9,3:7,16:9,1:3),
  col=4,pc=T)
print(A)

# 1 5  5 12
# 2 6  6 11
# 3 7  7 10
# 4 8 16  9
# 2 9 15  1
# 3 3 14  2
# 4 4 13  3

B = apply(A,2,sort)
print(B)

# 1 3  5  1
# 2 4  6  2
# 2 5  7  3
# 3 6 13  9
# 3 7 14 10
# 4 8 15 11
# 4 9 16 12
```

Nel secondo esempio abbiamo usato la funzione `sort` che restituisce un vettore ordinato per grandezza, come vedremo.

Per applicare la funzione `S.tra01` introdotta a pagina 7 a tutte le colonne di una matrice, ottenendo così la matrice

$$X^{01} := (X_1^{01}, \ldots, X_m^{01})$$

combiniamo questa funzione con `apply`:

```
Sm.tra01 = function (X)
apply(X,2,S.tra01)
```

Se, in analogia con le funzioni `colSums` ecc. viste a pagina 20, vogliamo calcolare massimi e minimi di righe e colonne di una matrice `A`, possiamo usare semplicemente `apply(A,1,max)` e `apply(A,1,min)` per le righe, `apply(A,2,max)` e `apply(A,2,min)` per le colonne.

Nello stesso modo con `apply(A,2,var)` si ottengono le varianze delle colonne di `A`.

$a = (a_1, \ldots, a_n)$ sia un vettore numerico. Come abbiamo visto a pagina 7, con `diff(a)` otteniamo il vettore

$$(a_2 - a_1, a_3 - a_2, \ldots, a_n - a_{n-1})$$

Siccome

$$a_i > a_{i-1} \iff a_i - a_{i-1} > 0$$

`sum(diff(a)>0)` indica quante volte un elemento del vettore è maggiore di quello precedente:

```
a=c(1,2,5,2,1,3,5,2)
volte=sum(diff(a)>0)
print(volte)
```

Se A è una matrice, con

```
apply(A,2,function (x) sum(diff(x)>0))
```

possiamo calcolare quante volte in ogni colonna un termine sia maggiore di quello precedente.

Ordinamento (sort)

Consideriamo le seguenti funzioni:

```
rev sort rank order
```

`rev` (che abbiamo già incontrato a pagina 7) non è una vera funzione di ordinamento, ma restituisce semplicemente gli elementi di un vettore in ordine invertito:

```
u=c(1:3,8,0,3:6,2); print(u)
# output: 1 2 3 8 0 3 4 5 6 2
i=rev(u); print(i)
# output: 2 6 5 4 3 0 8 3 2 1
```

La funzione di ordinamento `sort` ordina un vettore di numeri reali, in ordine crescente se non è indicata l'opzione `decreasing=T`. Per il trattamento dei valori `NA` consultare `?sort`. Usiamo lo stesso `u`:

```
crescente=sort(u)
print(crescente)
# output: 0 1 2 2 3 3 4 5 6 8
decrescente=sort(u,decreasing=T)
print(decrescente)
# output: 8 6 5 4 3 3 2 2 1 0
```

Per ottenere la somma dei 4 elementi più grandi di un vettore possiamo ordinarlo in modo decrescente, poi fare la somma dei primi 4 elementi del vettore ordinato:

```
v=c(13,4:8,0,11,12,3,2,14)
decrescente=sort(v,decreasing=T)
print(sum(decrescente[1:4]))
# output: 50
```

Per calcolare la media di un vettore da cui abbiamo tolto l'elemento più grande e l'elemento più piccolo, procediamo così:

```
v=1:7
o=sort(v)
o=sort(v)[-c(1,length(v))]
print(mean(o))
# output: 4
```

Infatti $2+3+4+5+6 = 20$, e la media di questi 5 numeri è 4.

In questo numero

rank

Con `rank` si ottiene la posizione di ogni elemento nella successione ordinata:

```
u=c(4,5,1,6,9,11,2)
print(sort(u))
# 1 2 4 5 6 9 11

print(rank(u))
# 3 4 1 5 6 7 2

u=c(1,2,3,8,0,3,4,5,6,2,3)
print(sort(u))
# 0 1 2 2 3 3 3 4 5 6 8

print(rank(u))
# 2 3.5 6 11 1 6 8 9 10 3.5 6
```

Come si vede, quando il vettore contiene valori uguali, `rank` assegna a questi elementi la media dei ranghi. Con l'impostazione `ties.method='first'` in `rank` i ranghi diventano invece unici, assegnando un rango minore a quelli tra due o più elementi uguali che nel vettore appaiono per primi:

```
u=c(1,2,3,8,0,3,4,5,6,2,3)
print(sort(u))
# 0 1 2 2 3 3 3 4 5 6 8

x=rank(u,ties.method='first')
print(x)
# 2 3 5 11 1 6 8 9 10 4 7
```

Le scelte `'min'` e `'max'` sono spesso utilizzate in classifiche sportive. Manca l'opzione `decreasing=T`, per questo applichiamo `rank` ai risultati negativi:

```
risultati=c(450,430,415,422,
  430,452,450,435,423,415)
ordinati=sort(risultati,
  decreasing=T)
ranghi=rank(-risultati,
  ties.method='min')

A=Mm(c(ordinati,sort(ranghi)),
  col=2,pc=T)

print(A)
```

con output

```
452    1
450    2
450    2
435    4
430    5
430    5
423    7
422    8
415    9
415    9
```

order

`order` restituisce successivamente la posizione dell'elemento più piccolo nella successione originale, poi quella del secondo e così via:

```
u=c(1,2,3,8,0,3,4,5,6,2,3)
print(u)
# 1 2 3 8 0 3 4 5 6 2 3

print(sort(u))
# 0 1 2 2 3 3 3 4 5 6 8

print(order(u))
# output: 5 1 2 10 3 6 7 8 9 4
```

`u[order(u)]` coincide con `sort(u)`! Questa funzione un po' astratta ha molte applicazioni.

Per ordinare le righe di una matrice `A` rispetto ai valori nella prima colonna si può procedere come nel seguente esempio:

```
A=Mm(c(3,2,1,4,8,7,6,11),col=2,pc=T)
print(A)

ordine=order(A[,1])
B=A[ordine,]
print(B)
```

con output

```
3   8
2   7
1   6
4  11

1   6
2   7
3   8
4  11
```

`order` può essere applicata anche a più argomenti; con `order(x,y,z)` si ottiene l'ordine degli elementi di `x` in cui in caso di parità viene tenuto conto dell'ordine in `y` e in caso di ulteriore parità dell'ordine in `z`. In questo modo le righe di una matrice possono essere ordinate secondo più criteri:

```
x=c(3,1,1,1,4,2,2)
y=c(5,6,6,2,3,8,8)
z=c(6,1,0,2,4,5,7)

u=order(x,y,z)
print(u)
# 4 3 2 6 7 1 5

A=Mm(c(x,y,z),col=3,pc=T)
B=A[u,]
print(B)
```

con output

```
1  2  2
1  6  0
1  6  1
2  8  5
2  8  7
3  5  6
4  3  4
```

Supponiamo di voler trovare i 3 elementi più piccoli di un vettore `v`. A questo scopo possiamo usare sia l'istruzione `sort(v)[1:3]` che `v[order(v)[1:3]]`. In modo simile si trovano i 3 elementi più grandi.

`order` può essere usata anche per trovare i valori di x per cui una funzione $\bigcirc_x f(x)$ assume un minimo o un massimo su un insieme finito di punti. Consideriamo la funzione

$$\bigcirc_x \sin(x^2+1) : [0, \sqrt{\pi-1}] \longrightarrow \mathbb{R}$$

Essa dovrebbe assumere un massimo (uguale a 1) in $x = \sqrt{\frac{\pi}{2} - 1} = 0.75551$. Proviamo a trovare il massimo con `order`:

```
f = function (x) sin(x^2+1)

x=seq(0,sqrt(pi-1),by=0.001)
y=f(x)

u=order(y)
k=tail(u,1)
xmax=x[k]
print(xmax)     # 0.756
print(f(xmax)) # 0.9999997
```

Il risultato è ottimo, tenendo conto del fatto che nella successione degli x abbiamo utilizzato incrementi di 0.001.

Numeri complessi

La notazione per i numeri complessi in R è molto simile a quella usata in matematica. Il numero complesso $6 + 2i$ viene denotato con `6+2i`, ma per $6 + i$ bisogna usare `6+1i`, perché `i` verrebbe interpretato come il nome di una variabile. Numeri reali, ad esempio 0 o 7, in alcuni contesti vengono interpretati automaticamente come numeri complessi; quando necessario si usano $0+0i$ e $7+0i$ oppure `as(0,'complex')` e `as(7,'complex')`. Si possono formare vettori e matrici di numeri complessi, ad esempio

```
z = c(3,4+2i,5,1i)
print(z)
# 3+0i 4+2i 5+0i 0+1i
```

Vettori di numeri complessi possono essere considerati come vettori di punti nel piano e ciò è utilissimo per la grafica nel piano. Se `z` è un vettore di numeri complessi, con `mean(z)` otteniamo il baricentro dei punti dati; se `z1` e `z2` sono numeri complessi distinti, con `z1+t*(z2-z1)` otteniamo tutti i punti della retta passante per `z1` e `z2`. R fornisce alcune funzioni di base per i numeri complessi:

`Re(z)`	... parte reale di z
`Im(z)`	... parte immaginaria di z
`Conj(z)`	... complesso coniugato $\overline{z}$
`abs(z)`	... modulo $\lvert z\rvert$
`Arg(z)`	... angolo α in $z = \lvert z\rvert e^{i\alpha}$

Invece di `abs(z)` si può anche usare `Mod(z)`.

Anche le funzioni `exp`, `log`, `cos` e `sin` possono essere usate per numeri complessi.

Il teorema di Rouché

Teorema 22.1 (teorema di Rouché). *g ed h siano due polinomi a coefficienti complessi e C una circonferenza (cioè il bordo di un cerchio) nel piano tale che*

$$|g(z)| > |h(z)| \text{ per ogni } z \in C.$$

La disuguaglianza è quindi richiesta solo sul bordo del cerchio, non al suo interno.

Allora i polinomi g e $g+h$ possiedono all'interno del cerchio lo stesso numero di radici, se queste vengono contate con le loro molteplicità.

Questo è un teorema molto importante dell'analisi complessa. Non confondere C con $\mathbb{C}$.

Il teorema di Rouché si usa nel modo seguente. Assumiamo che vogliamo studiare gli zeri di un polinomio f. Allora decomponiamo f nella forma $f = g + h$, dove g è un polinomio sui cui zeri, almeno all'interno di un cerchio, abbiamo sufficienti informazioni. Consideriamo in altre parole f come perturbazione di g mediante h. Se riusciamo a dimostrare che sul bordo del cerchio si ha sempre $|g(z)| > |h(z)|$, allora possiamo concludere che f e g all'interno del cerchio possiedono lo stesso numero di radici, tenuto conto della loro molteplicità. Quindi se g nel cerchio possiede una sola radice tripla, non possiamo dire che anche f possiede una radice tripla, possiamo però dire che f nel cerchio possiede esattamente tre radici se queste vengono contate con le loro molteplicità.

Teorema 22.2. *Siano*

$$f = a_0 + a_1 x + \cdots + a_n x^n$$

un polinomio con coefficienti complessi, $n \in \mathbb{N}+1$ e $a_n \neq 0$. Allora esistono numeri complessi $\alpha_1, \ldots, \alpha_n$, univocamente determinati, tali che

$$f = a_n(x-\alpha_1)\cdots(x-\alpha_n)$$

Questa uguaglianza è intesa come uguaglianza di polinomi, cioè sviluppando il prodotto a destra si ottiene un polinomio con gli stessi coefficienti di f e quindi proprio f.

È chiaro anche che $f(\alpha_k) = 0$ per ogni k e si può dimostrare che ogni radice di f, cioè ogni α per cui $f(\alpha) = 0$, è uno degli α_k.

Dimostrazione. Questo teorema si chiama il teorema fondamentale dell'algebra. Per la dimostrazione dell'esistenza di una radice usiamo il teorema di Rouché e consideriamo f come perturbazione di

$$g := a_n x^n$$

mediante

$$h := a_{n-1}x^{n-1} + \cdots + a_0$$

Per ogni $z \in \mathbb{C}$ si ha $|g(z)| = |a_n||z|^n$ e si vede facilmente che per $|z|$ sufficientemente grande questa n-esima potenza domina largamente le potenze inferiori e quindi $|h(z)|$, per quanto grandi siano i coefficienti di queste potenze inferiori rispetto al coefficiente a_n di x^n.

Perciò esiste certamente un $R > 0$ tale che per ogni z che appartiene alla circonferenza di raggio R (cioè tale che $|z| = R$) si ha $|g(z)| > |h(z)|$.

Il teorema di Rouché implica allora che f, il nostro polinomio dato, all'interno del cerchio $|z| < R$ possiede lo stesso numero di radici come g. Ma $g = a_n x^n$ in questo cerchio possiede la radice $z = 0$ di molteplicità n. E quindi anche f possiede all'interno dello stesso cerchio n radici e quindi almeno una radice perché per ipotesi $n \geq 1$.

Radici di un polinomio

In R le radici di un polinomio possono essere ottenute con la funzione `polyroot` che prende come argomento il vettore dei coefficienti di f elencati iniziando con il coefficiente costante. Per arrotondare i risultati usiamo la funzione `round` vista a pagina 3. Esempio:

```
radici=polyroot(c(6,-5,7,-5,1))
print(round(radici,2))
```

con output `0+1i 0-1i 2+0i 3+0i`. Infatti

$$6 - 5x + 7x^2 - 5x^3 + x^4 = (x-2)(x-3)(x-i)(x+i)$$

Le formule di Euler e de Moivre

Ogni punto $z = (x, y)$ di $\mathbb{R}^2$ può essere scritto nella forma

$$z = (r\cos\alpha, r\sin\alpha)$$

dove $r \geq 0$ è univocamente determinato ed $\alpha \in \mathbb{R}$. Notiamo che

$$r = |z| = \sqrt{x^2 + y^2}$$

Se $z \neq 0$, anche α è univocamente determinato se chiediamo $0 \leq \alpha < 2\pi$ oppure, come il matematico preferisce dire, univocamente determinato modulo 2π.

Nei corsi di Analisi si dimostra che

$$e^{i\alpha} := \cos\alpha + i\sin\alpha$$

(formula di Euler). Dal teorema di addizione per le funzioni trigonometriche si deduce immediatamente che

$$\boxed{e^{i(\alpha+\beta)} = e^{i\alpha} \cdot e^{i\beta}}$$

per $\alpha, \beta \in \mathbb{R}$. Si vede che nel campo complesso il teorema di addizione assume una forma molto più semplice.

Possiamo quindi scrivere ogni numero complesso z nella forma $z = re^{i\alpha}$ con r ed α come sopra.

Sia adesso anche $\beta \in \mathbb{R}$. Allora

$$e^{i\beta} z = re^{i(\alpha+\beta)}$$

come segue dal teorema di addizione appena visto. Il prodotto $e^{i\beta} z$ ha quindi la stessa lunghezza di z ed un angolo aumentato di β rispetto a z. Ciò significa che la moltiplicazione con $e^{i\beta}$ è la stessa cosa come una *rotazione* per l'angolo β in senso antiorario.

Prendiamo adesso un numero complesso c arbitrario. Lo possiamo rappresentare nella forma $c = se^{i\beta}$ con $s \geq 0$. Per il prodotto cz otteniamo evidentemente

$$cz = sre^{i(\alpha+\beta)}$$

e quindi vediamo che la moltiplicazione con un numero complesso consiste sempre di una rotazione combinata con un allungamento (o accorciamento se $|c| < 1$).

Sia $z = re^{i\alpha} \in \mathbb{C}$. Per $n \geq 1$ allora, secondo le formule viste precedentemente, abbiamo

$$\boxed{z^n = r^n e^{in\alpha}}$$

Questa è la *formula di de Moivre*.

Per $z = x + iy$ possiamo anche più in generale definire

$$\boxed{e^z := e^x e^{iy}}$$

Allora

$$\boxed{e^{z+w} = e^z \cdot e^w}$$

per ogni $z, w \in \mathbb{C}$. La funzione esponenziale è quindi un omomorfismo

$$(\mathbb{C}, +) \longrightarrow (\mathbb{C}, \cdot)$$

La formula

$$e^{iz} = \cos z + i\sin z$$

rimane valida anche per $z \in \mathbb{C}$ non necessariamente reale, ma e^{iz} si trova sulla circonferenza unitaria se e solo se $z \in \mathbb{R}$.

Radici n-esime dell'unità

Sia $n \in \mathbb{N} + 1$ fissato. Consideriamo il numero

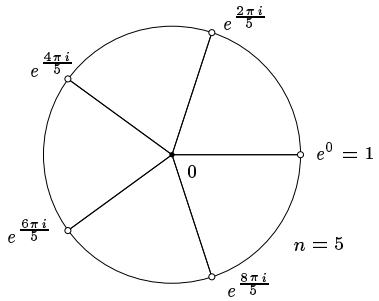
$$\varepsilon = e^{\frac{2\pi i}{n}}$$

(che naturalmente dipende da n). Dalla formula di de Moivre segue che ε elevato alla n-esima potenza è uguale a 1. È anche chiaro che

$$(\varepsilon^k)^n = (\varepsilon^n)^k = 1^k = 1$$

per ogni $k \in \mathbb{Z}$. Consideriamo adesso i numeri $1, \varepsilon, \varepsilon^2, \ldots, \varepsilon^{n-1}$.

Dalla formula di Euler vediamo che gli ε^k si trovano tutti sul cerchio unitario, con ε^k ruotato di $\dfrac{2\pi}{n}$ (cioè di $\dfrac{360}{n}$ gradi) rispetto ad ε^{k-1}. Essi formano in altre parole (almeno per $n \geq 3$) i vertici del poligono regolare con n vertici iscritto al cerchio unitario con primo vertice uguale ad 1.



Ciò implica che gli ε^k per $k = 0, 1, \ldots, n-1$ sono tutti distinti tra di loro e costituiscono un insieme di n radici del polinomio $x^n - 1$, mentre per altri k i valori si ripetono. Dal teorema fondamentale dell'algebra segue che ogni radice di $x^n - 1$ è uno degli ε^k e che

$$x^n - 1 = (x-1)(x-\varepsilon)(x-\varepsilon^2)\cdots(x-\varepsilon^{n-1})$$

I numeri $1, \varepsilon, \varepsilon^2, \ldots, \varepsilon^{n-1}$ si chiamano le n-esime *radici dell'unità*.

Se poniamo

$$\mathcal{U}_n := \{1, \varepsilon, \varepsilon^2, \ldots, \varepsilon^{n-1}\}$$

$\mathcal{U}_n$ diventa, con la moltiplicazione naturale in $S^1 = \{z \in \mathbb{C} \mid |z| = 1\}$,

$$\varepsilon^k \varepsilon^j = \varepsilon^{(k+j) \bmod n}$$

un gruppo ciclico (e quindi abeliano) isomorfo a $\mathbb{Z}/n$. Questa isomorfia è molto importante nell'analisi armonica (analisi di Fourier) dei gruppi finiti e in teoria dei numeri perché introduce la possibilità di utilizzare le funzioni trigonometriche in questi campi della matematica.

Radici di un numero complesso

Siano $r \geq 0$ un numero reale non negativo ed $n \in \mathbb{N} + 1$. Nei corsi di Analisi si impara che esiste un unico numero $\rho \geq 0$ tale che $\rho^n = r$. Denotiamo questo numero con $\sqrt[n]{r}$.

Sia adesso $z \neq 0$ un numero complesso diverso da 0. Allora $z = re^{it}$ con $r > 0$ e $t \in \mathbb{R}$. Cerchiamo le radici n-esime di z, cioè le radici del polinomio $x^n = z$. Una radice la troviamo subito; infatti la formula di de Moivre implica che

$$\alpha_1 := \sqrt[n]{r} e^{i\frac{t}{n}}$$

soddisfa l'equazione $\alpha_1^n = z$. Se λ adesso è un numero complesso tale che $\lambda^n = 1$, allora anche

$$(\alpha_1 \lambda)^n = \alpha_1^n \lambda^n = \alpha_1^n = r$$

Però noi conosciamo i λ per cui $\lambda^n = 1$; sono le n-esime radici dell'unità. Quindi ciascuno dei numeri

$$\begin{aligned}
\alpha_1 &:= \sqrt[n]{r} e^{i\frac{t}{n}} \\
\alpha_2 &:= \varepsilon\alpha_1 \\
\alpha_3 &:= \varepsilon^2\alpha_1 \\
&\ldots \\
\alpha_n &:= \varepsilon^{n-1}\alpha_1
\end{aligned}$$

è una radice di $x^n - z$. D'altra parte la moltiplicazione di un numero complesso w con ε^k corrisponde alla rotazione di w per un angolo di $k\,\dfrac{360}{n}$ gradi e, siccome $z \neq 0$ e quindi anche $\alpha_1 \neq 0$, tutti gli α_k sono distinti tra di loro. Dal teorema fondamentale dell'algebra segue che

$$x^n - z = (x - \alpha_1)\cdots(x - \alpha_n)$$

e che ogni numero complesso α che soddisfa l'equazione $\alpha^n = z$ è uno degli α_k.

Troviamo con R le quarte radici di $1+3i$, cioè le radici del polinomio $x^4 - (1 + 3i)$:

```
radici=polyroot(c(-1-3i,0,0,0,1))
print(round(radici,2))
```

L'output è

```
 1.27+0.41i -0.41+1.27i
-1.27-0.41i  0.41-1.27i
```

Con `(1+3i)^(1/4)` otteniamo solo la prima di queste radici:

```
w=(1+3i)^(1/4)
print(round(w,2))
#  1.27+0.41i
```

in accordo con l'output precedente.

FONDAMENTI DI INFORMATICA

Corso di laurea in matematica | Anno accademico 2004/05 | Numero 7

paste

Questa potente funzione di semplice sintassi, che abbiamo usato a pagina 14, permette un flessibile concatenamento di stringhe (o di oggetti convertibili a stringhe). La funzione prevede due opzioni, `sep` per il carattere di unione tra le stringhe generate (preimpostato a `" "`), e `collapse`, inizialmente uguale a `NULL`. Assegnando a `collapse` una stringa `z`, `paste` restituisce un'unica parola che si ottiene concatenando le stringhe del risultato standard con quella stringa `z`. Esempio:

```
vocali=c('a','e','i','o','u')
u=paste(vocali)
print(u)
# "a" "e" "i" "o" "u"
# Nessun effetto.

vocali=c('a','e','i','o','u')
u=paste(vocali,collapse='')
print(u)
# "aeiou"

u=paste(c("alfa","beta"),1)
print(u)
# "alfa 1" "beta 1"

u=paste(c("alfa","beta"),1,
    collapse=" + ")
print(u)
# "alfa 1 + beta 1"
```

Si vede che `paste` opera in modo vettoriale, riciclando i componenti di vettori più corti dell'argomento più lungo. Esempi:

```
u=paste(c("u","v"),1,c("a","b","c"),
    sep="")
print (u)
# "u1a" "v1b" "u1c"

u=paste(c("u","v"),1,c("a","b","c"),
    collapse=' + ')
print (u)
# "u 1 a + v 1 b + u 1 c"

u=paste(c("u","v"),1,c("a","b","c"),
    sep='', collapse=' + ')
print (u)
# "u1a + v1b + u1c"

u=paste(c("u","v"),1,c("a","b","c"),
    sep='==', collapse=' + ')
print (u)
# "u==1==a + v==1==b + u==1==c"

u=paste(1:4,5:6,1:3)
print(u)
# "1 5 1" "2 6 2" "3 5 3" "4 6 1"

u=paste(1:4,5:6,1:3,sep='.')
print(u)
# "1.5.1" "2.6.2" "3.5.3" "4.6.1"

u=paste(1:4,5:6,1:3,
  collapse=', ')
print(u)
# "1 5 1, 2 6 2, 3 5 3, 4 6 1"

u=paste(1:4,5:6,1:3,sep='',
  collapse=', ')
print(u)
# "151, 262, 353, 461"

u=paste(1:4,5:6,1:3,sep='.',
  collapse=', ')
print(u)
# "1.5.1, 2.6.2, 3.5.3, 4.6.1"
```

Con `collapse='\n'` possiamo unire un vettore in un testo su più righe.

```
righe=c('Prima riga.',
  'Seconda riga.',
  'Terza riga.')
u=paste(righe,collapse='\n')
cat(u,'\n')
# Prima riga.
# Seconda riga.
# Terza riga.
```

Stavolta abbiamo dovuto usare `cat` per l'output, perché `print` non riconosce il carattere speciale `\n`.

`paste` viene usata spesso in combinazione con `cat`, `sprintf` o `eval`.

substring

Per estrarre sottostringhe da una stringa si usa `substring`. La sintassi è molto semplice: `substring(a,i,j)` è la stringa che consiste dei caratteri dalla `i`-esima alla `j`-esima posizione in `a`, mentre `substring(a,i)` è la stringa che consiste dei caratteri dalla `i`-esima posizione fino alla fine di `a`. Se `i` supera la lunghezza di `a` o se `j` è minore di `i`, la funzione restituisce la stringa vuota; se `j` supera la lunghezza di `a`, viene restituita la stringa `substring(a,i,nchar(j))`; se `i` è minore di 1, il risultato è `substring(a,1,j)`.

```
a='Il bel castello'
u=substring(a,8,13)
print(u)
# "castel"
```

oppure

```
a='Il bel castello'
v=substring(a,8)
print(v)
# "castello"
```

Se i limiti `i` e `j` sono vettori, viene restituito un vettore di sottostringhe, come nel seguente esempio:

```
a='1234567890'
w=substring(a,4:6,8:9)
print(w)
# "45678" "56789" "678"
```

In questo caso sono quindi stati formati le coppie di indici $(4,8)$, $(5,9)$ e $(6,8)$, riciclando il secondo vettore più corto.

Il primo, il secondo e l'ultimo carattere di una stringa `a` possono essere trovati con `substring(a,1,1)`, `substring(a,2,2)` e `substring(a,nchar(a))`.

In questo numero

chartr

Questa è una semplice funzione di trascrizione di caratteri: `chartr(x,y,a)` è il vettore che si ottiene sostituendo negli elementi di `a` ogni carattere che appare nella stringa `x` con il carattere che si trova nella stessa posizione in `y` (che quindi deve avere la stessa lunghezza di `x`). Esempio:

```
u=c("ACGGTTA","CGGATA","GGATTACCAA")
v=chartr("GACT","0123",u)
print(v)
```

con output

```
"1200331" "200131" "0013312211"
```

In questo modo possiamo trasformare sequenze genetiche scritte nelle lettere tradizionali G, A, C, T in stringhe numeriche, ad esempio per una successiva analisi armonica su $\mathbb{Z}/4$.

abbreviate

Con `abbreviate(a,2)` si ottiene un vettore di parole che costituiscono abbreviazioni, possibilmente a 2 lettere, delle parole che sono elementi del vettore `a`. A parole diverse vengono associate abbreviazioni distinte, perciò talvolta un'abbreviazione contiene più lettere di quante indicate nel secondo argomento.

```
citta=c('Roma','Bologna','Bolzano',
  'Ferrara','Padova','Ravenna',
  'Parma')
x=as.character(abbreviate(citta,2))
print(x)
# "Rm" "Blg" "Blz" "Fr" "Pd" "Rv" "Pr"
```

Espressioni regolari

Un'espressione regolare è una formula che descrive un insieme di parole. Questo importante concetto dell'informatica teorica è entrato in molti linguaggi di programmazione, soprattutto nel mondo Unix. Usiamo qui la sintassi valida per il Perl, che può essere scelta mediante l'opzione `perl=T` nelle funzioni previste in R per le espressioni regolari. Sotto Unix si può consultare `man pcre` per la libreria PCRE.

Una parola come espressione regolare corrisponde all'insieme di tutte le parole che la contengono (ad esempio `alfa` è contenuta in `alfabeto` e `stalfano`, ma non in `stalfino`. `^alfa` indica invece che `alfa` si deve trovare all'inizio della riga, `alfa$` che si deve trovare alla fine. È come se `^` e `$` fossero due lettere invisibili che denotano inizio e fine della riga. Il carattere spazio viene trattato come gli altri, quindi con `a lfa` si trova `kappa lfa`, ma non `alfabeto`.

Il punto `.` denota un carattere qualsiasi, ma un asterisco `*` non può essere usato da solo, ma indica una ripetizione arbitraria (anche vuota) del carattere che lo precede. Quindi `a*` sta per le parole `a`, `aa`, `aaa`, ... , e anche per la parola vuota. Per quest'ultima ragione `alfa*ino` trova `alfino`. Per escludere la parola vuota si usa `+` al posto dell'asterisco. Ad esempio `+` indica almeno uno e possibilmente più spazi vuoti. Questa espressione regolare viene spesso usata per separare le parti di una riga. Per esempio `*, +` trova `alfa , beta`, ma non `alfa ,beta`. Il punto interrogativo `?` dopo un carattere indica che quel carattere può apparire una volta oppure mancare, quindi `alfa?ino` trova `alfino` e `alfaino`, ma non `alfaaino`.

Le parentesi quadre vengono utilizzate per indicare insiemi di caratteri oppure il complemento di un tale insieme. `[aeiou]` denota le vocali minuscole e `[^aeiou]` tutti i caratteri che non siano vocali minuscole. È il cappuccio `^` che indica il complemento. Quindi `r[aeio]ma` trova `rima` e `romano`, mentre `[Rr][aeio]ma` trova anche `Roma`. Si possono anche usare trattini per indicare insiemi di caratteri successivi naturali, ad esempio `[a-zP]` è l'insieme di tutte le lettere minuscole dell'alfabeto comune insieme alla `P` maiuscola, e `[A-Za-z0-9]` sono i cosiddetti caratteri alfanumerici, cioè le lettere dell'alfabeto insieme alle cifre. Per questo insieme si può usare l'abbreviazione `\\w`, per il suo complemento `\\W`.

La barra verticale `|` tra due espressioni regolari indica che almeno una delle due deve essere soddisfatta. Si possono usare le parentesi rotonde: `a|b|c` è la stessa cosa come `[abc]`, `r(oma|ume)no` trova `romano` e `rumeno`.

Per indicare i caratteri speciali `.`, `*`, `^` ecc. a differenza dal Perl (in cui è sufficiente un singolo `\`, come peraltro in `\w` e `\W`) bisogna anteporgli `\\`, ad esempio `\\.` per indicare veramente un punto e non un carattere qualsiasi.

Le espressioni regolari sono uno degli strumenti più frequentemente utilizzati nel trattamento di testi; spesso inserendo o togliendo pochi caratteri in un'istruzione è possibile effettuare una modifica che in C poteva richiedere la completa riscrittura di una parte consistente del programma.

I modificatori (?m) e (?s)

Il punto `.` nelle espressioni regolari sta per un carattere qualsiasi diverso dal carattere di nuova riga. Aggiungendo `(?s)` all'espressione regolare, si ottiene che `.` comprende anche il carattere di nuova riga; si farà così quando con `.*` si vuole denotare una successione arbitraria di caratteri che si può estendere anche su più righe.

Come visto a ..., `^` e `$` vengono utilizzati per indicare l'inizio e la fine della stringa. Questo significato cambia se aggiungiamo `(?m)`: in questo caso `^` indica anche l'inizio di una riga (cioè l'inizio della stringa oppure una posizione preceduta da un carattere di nuova riga), e similmente `$` indica anche la fine di una riga (cioè la fine della stringa oppure una posizione a cui segue un carattere di nuova riga).

I modificatori `(?s)` e `(?m)` valgono dalla posizione in cui si trovano nell'espressione regolare; è possibile, come vedremo, anche limitare la loro validità anche a un segmento usando la sintassi `(?s:segmento)` e `(?m:segmento)`.

Se, mentre si usa il modificatore `(?m)`, ci si vuole riferire all'inizio della stringa, si utilizza `\\A` (che senza `(?m)` ha lo stesso significato di `^`); mentre similmente `\\z` indica la fine della stringa anche in presenza di `(?m)` ed è invece equivalente a `$` in assenza di `(?m)`.

Siccome stringhe prelevate da un file spesso contengono un ultimo carattere di nuova riga, esiste un altro simbolo `\\Z` che corrisponde alla posizione precedente a questo ultimo carattere di nuova riga, quando presente, altrimenti alla vera fine della stringa.

I simboli `\\A`, `\\z` e `\\Z` perdono naturalmente il loro significato all'interno di parentesi quadre.

I metasimboli

Abbiamo già spiegato il significato di `\\A`, `\\z` e `\\Z`. I simboli `\\0`, `\\n` e `\\t` vengono usati più o meno come in C e indicano il carattere ASCII 0, il carattere di nuova riga e il tabulatore. Vedere `?regex`. Esistono numerosi altri metasimboli, di cui elenchiamo quelli più comuni, sufficienti in quasi tutte le applicazioni pratiche:

`\\w`	carattere alfanumerico, equivalente a [A-Za-z0-9_]
`\\W`	non carattere alfanumerico
`\\d`	[0-9] – il *d* deriva da *digit* (cifra)
`\\D`	[^0-9]
`\\s`	spazio bianco, probabilmente `[ \\n\\r\\f]`
`\\S`	non spazio bianco

toupper e tolower

`toupper(a)` e `tolower(a)` restituiscono i vettori di parole che si ottengono sostituendo negli elementi di `a` tutti i caratteri con maiuscole risp. minuscole.

```
a=c('alfa','Roma')
b=toupper(a)
print(b)
# "ALFA" "ROMA"
```

I metacaratteri

I seguenti caratteri hanno un significato speciale nelle espressioni regolari: `\`, `|`, `(`, `)`, `[`, `]`, `{`, `}`, `^`, `$`, `*`, `+`, `?`, `.` e, all'interno di `[ ]`, anche `–`. Per privare questi caratteri del loro significato speciale, è sufficiente preporgli un `\\` (talvolta basta un semplice `\`).

- `\\` viene usato per dare a un carattere il suo significato normale. Spesso però è sufficiente un backslash semplice.
- `|` indica scelte alternative, che vengono esaminate da sinistra a destra.
- `()` Le parentesi tonde vengono usate in più modi. Possono servire a racchiudere semplicemente un'espressione per limitare il raggio d'azione di un'alternativa, per distinguere ad esempio `a(u| v)` da `au| v`, oppure per catturare una parte da usare ancora (cfr. pagina 26). Altri usi delle parentesi tonde vengono descritti separatamente.
- `[]` Le parentesi quadre racchiudono insiemi di caratteri oppure il loro complemento (se subito dopo la parentesi iniziale `[` si trova un `^` che in questo contesto non ha più il significato di inizio di parola che ha al di fuori delle parentesi quadre).
- `{}` Le parentesi graffe permettono la quantificazione delle ripetizioni: `a{3}` significa `aaa`, `a{2,5}` comprende `aa`, `aaa`, `aaaa` ed `aaaaa`.
- `^` Questo carattere indica l'inizio di parola (oppure, quando è presente il modificatore `(?m)`, anche l'inizio di una riga), quando non si trova all'interno delle parentesi quadre, dove, se si trova all'inizio, significa la formazione del complemento.
- `$` indica la fine della parola o della riga a seconda che manchi o sia presente l'operatore `(?m)`.
- `*` L'asterisco è un quantificatore e indica che il simbolo precedente può essere ripetuto un numero arbitrario di volte (o anche mancare). Un `?` altera il comportamento di `*` come vedremo.
- `+` Ha lo stesso significato di `*`, tranne che il simbolo deve apparire almeno una volta. Un `?` altera il comportamento di `+`.
- `?` `a?` significa che `a` può apparire oppure no, con preferenza per il primo caso; `a??` invece con preferenza per il secondo caso. `*?` significa che viene scelta la corrispondenza più breve possibile (altrimenti il Perl sceglie la più lunga); un discorso analogo vale per `+?`.
- `.` sta per un singolo carattere che deve essere diverso dal carattere di nuova riga se non è presente il modificatore `(?s)`.
- `-` all'interno di parentesi quadre può essere usato per denotare un insieme di caratteri attigui. Per avere un semplice `-` all'interno delle parentesi graffe si deve usare `\-` o `\\-`.

Il modificatore (?i)

Aggiungendo `(?i)` all'espressione regolare non viene distinto tra minuscole e maiuscole. L'uso e il significato della posizione sono uguali come per `(?m)` e `(?s)`; in particolare si può limitare la richiesta a un segmento con `(?i:segmento)`. La `i` viene da *ignore case*; infatti nelle funzioni per le espressioni regolari si può anche usare l'opzione `ignore.case=T` invece di `(?i)`.

grep

`a` sia un vettore di parole ed `E` un'espressione regolare. Allora `grep(E,a,perl=T)` restituisce gli indici degli elementi di `a` che contengono una parte descritta da `E`, mentre con `grep(E,a,perl=T,value=T)` fornisce questi elementi stessi. Se non viene trovato nessun elemento, il risultato è `numeric(0)` nel primo caso, `character(0)` nel secondo.

```
a=c('Bologna','Bolzano','Ferrara',
  'Padova','Parma','Roma')

u=grep('^pa',a,perl=T)
print(u)
# numeric(0)

u=grep('(?i)^pa',a,perl=T)
print(u)
# 4 5

u=grep('(?i)^pa',a,perl=T,value=T)
print(u)
# "Padova" "Parma"

u=grep('^[BR]',a,perl=T)
print(u)
# 1 2 6

u=grep('^[B-F]',a,perl=T)
print(u)
# 1 2 3

u=grep('[mn]a$',a,perl=T,value=T)
print(u)
# "Bologna" "Parma" "Roma"

u=grep('^.[ae]',a,perl=T)
print(u)
# 3 4 5

u=grep('....',a,perl=T)
print(u)
# 1 2 3 4 5 6

u=grep('^....$',a,perl=T,value=T)
print(u)
# "Roma"
```

Con l'opzione `fixed=T` e togliendo (in contrasto con ciò che dice l'aiuto) `perl=T`, l'espressione regolare viene considerata come stringa così com'è:

```
a='v[3]+6'
u=grep('v[3]',a,perl=T)
print(u)
# numeric(0)

u=grep('v[3]',a,fixed=T)
print(u)
# 1
```

Definiamo una nostra funzione:

```
Tr.grep = function (E,x,parole=F)
grep(E,x,perl=T,value=parole)
```

regexpr

Siano di nuovo `a` un vettore di parole ed `E` un'espressione regolare.

`regexpr(E,a,perl=T)` è allora il vettore delle posizioni in cui `E` viene trovata negli elementi di `a` con la posizione uguale a `-1` se non c'è corrispondenza; il vettore dei risultati è fornito di un attributo `'match.length'` che contiene le lunghezze delle parti corrispondenti.

```
a=c('Bologna','Parma','Pisa','Roma')
u=regexpr('ma$',a,perl=T)
print(u)
# -1 4 -1 3
# attr(,"match.length")
# -1 2 -1 2
```

Se vogliamo solo il vettore delle posizioni, dobbiamo applicare `as.numeric` al risultato; questa funzione può essere anche utilizzata per trasformare il vettore delle lunghezze.

```
a=c('Bologna','Parma','Pisa','Roma')
u=regexpr('ma$',a,perl=T)
print(as.numeric(u))
# -1 4 -1 3
print(as.numeric(attr(u,
  'match.length')))
# -1 2 -1 2
```

Definiamo adesso due funzioni nostre, di cui la prima restituisce le parti trovate, con una stringa vuota in mancanza di corrispondenza.

```
Tr.partitrovate = function (E,a)
{ris=regexpr(E,a,perl=T)
pos=as.numeric(ris)
lun=as.numeric(attr(ris,'match.length'))
substring(a,pos,pos+lun-1)}

# Vettore delle posizioni in cui E
# si trova negli elementi di v.
# -1 quando un elemento non corrisponde.
Tr.pos = function (E,a)
{as.numeric(regexpr(E,a,perl=T))}
```

Esempi:

```
a=c('01-356-1815','01-4561','a183')
E='56\\d|\\w18'
u=Tr.pos(E,a)
print(u)
# -1 5 1

v=Tr.partitrovate(E,a)
print(v)
# "" "561" "a18"
```

strsplit

`strsplit(a,E,perl=T)` restituisce una *lista* che, per ogni elemento `x` del vettore di parole `a`, consiste delle parti che si ottengono suddividendo `x` in corrispondenza all'espressione regolare `E`.

```
a='Rosa,, Aldo, Piero , Maria'
u=strsplit(a,'[, ]+',perl=T)
print(u[[1]])
# "Rosa" "Aldo" "Piero" "Maria"

a='Rosa,, Aldo, Piero , Maria'
u=strsplit(a,' *, *',perl=T)
print(u[[1]])
# "Rosa" "" "Aldo" "Piero" "Maria"
```

Si noti che nel primo esempio più virgole e spazi mescolati valgono come un separatore solo, mentre nel secondo ogni virgola genera una parte. L'espressione `' *, *'` viene usata molto spesso.

Siccome il risultato `u` è una lista, per stampare l'unica componente di `u` abbiamo dovuto riferirci a `u[[1]]`.

```
a=c('Rosa, Aldo', 'Piero , Maria')
u=strsplit(a,' *, *',perl=T)
print(u)
# [[1]]
# [1] "Rosa" "Aldo"
# [[2]]
# [1] "Piero" "Maria"
```

Sostituzioni con gsub e sub

`a` sia un vettore di parole, `E` un'espressione regolare ed `s` una stringa.

Allora `gsub(E,s,a,perl=T)` è il vettore di parole che si ottiene sostituendo in ogni elemento di `a` ogni parte corrispondente ad `E` con `s`.

```
u=gsub(' ','','A: 12345 :F',perl=T)
print(u)
# "A:12345:F"

a=c('alfa','beta','gamma')
u=gsub('[ae]','i',a,perl=T)
print(u)
# "ilfi" "biti" "gimmi"

a='Tasso-Ariosto-Dante'
u=gsub('-',', ',a,perl=T)
print(u)
# "Tasso, Ariosto, Dante"

a='ababaco'
u=gsub('aba','ibi',a,perl=T)
print(u)
# "ibibaco"
```

Nell'ultimo esempio si vede che l'elaborazione continua dalla posizione già raggiunta.

`sub` si usa meno e si distingue da `gsub` per il fatto che solo la prima corrispondenza di `E` in ogni parola del vettore `a` viene sostituita.

I riferimenti \\1, \\2, ...

Le parentesi tonde possono essere usate semplicemente per raggruppare gli elementi di una parte di un'espressione regolare. Allo stesso tempo però il contenuto della parte della corrispondenza rilevata viene memorizzato in variabili numerate `\\1`, `\\2`, ... che possono essere utilizzate successivamente. Questo aspetto in R sembra però molto meno elaborato e versatile che in Perl e probabilmente è difficile imitare, tranne in casi molto semplici, il potente modificatore `/e` del Perl. Esempi:

```
a='brrr che freddo!'
u=gsub('(.)\\1+','\\1',a,perl=T)
print(u)
# "br che fredo!"

a=c('x1','x2','y35','88')
u=gsub('^([a-z]+)(\\d+)','\\1[\\2]',
  a,perl=T)
print(u)
# "x[1]" "x[2]" "y[35]" "88"
```

Parentesi tonde speciali

Esistono anche parentesi tonde speciali il cui contenuto non viene memorizzato nelle variabili `\\1`, `\\2`, ..., di cui conosciamo già `(?m)`, `(?s)` e `(?i)`. Nella tabella `x` indica un segmento.

`(?:x)`	Parentesi non memorizzata.
`a(?=x)`	`a` deve essere seguito da `x`.
`a(?!x)`	`a` non deve essere seguito da `x`.
`(?<=x)a`	`a` deve essere preceduto da `x`.
`(?<!x)`	`a` non deve essere preceduto da `x`.
`(?i:x)`	Attiva il modificatore `(?i)` per il contenuto della parentesi.
`(?-i:x)`	Disattiva il modificatore `(?i)` per il contenuto della parentesi.
`(?s:x)`	Attiva il modificatore `(?s)` per il contenuto della parentesi.
`(?-s:x)`	Disattiva il modificatore `(?s)` per il contenuto della parentesi.

Le parentesi tonde speciali non occupano posto!

Lettura a triple del DNA

Nel codice genetico l'aminoacido isoleucina è rappresentato dalle triple *ATA*, *ATC* e *ATT*. Una stringa deve però essere letta a triple, quindi il primo *ATA* (a partire dalla seconda lettera) in

TATATCTGCAATTTGATAGATCGA

non verrà tradotto in isoleucina, perché appartiene in parte alla tripla *TAT* e in parte ad *ATC*. Possiamo trovare le triple che corrispondono all'isoleucina con il seguente procedimento a più passi:

```
a='TATATCTGCAATTTGATAGATCGA'
b=gsub('(...)(?!$)','\\1-',a,perl=T)
print(b)
# "TAT-ATC-TGC-AAT-TTG-ATA-GAT-CGA"
sp=strsplit(b,'\\-',perl=T)[[1]]
print(sp)
# "TAT" "ATC" "TGC" "AAT" "TTG"
#     "ATA" "GAT" "CGA"
d=Tr.grep('AT[ACT]',sp,parole=T)
print(d)
# "ATC" "ATA"
```

plot

Diamo solo le varianti essenziali di questi comandi. In particolare useremo `plot` solo per predisporre la finestra grafica, non per il disegno stesso.

`plot` richiede (nell'uso che ne facciamo) come primi argomenti l'indicazione dei limiti per le coordinate x ed y, entrambi nella forma `c(a,b)`. Se vogliamo disegnare il coseno, possiamo ad esempio impostare l'intervallo per x con `intx=c(-2*pi,2*pi)` e l'intervallo per y con `inty=c(-1,1)`.

I comandi di base per impostare la finestra grafica sono contenuti nella seguente funzione della nostra libreria:

```
Gr = function (x,y,cornice=F,
  rxy=1,assi=F,marg=0)
{par(mai=rep(marg,4), omi=c(0,0,0,0),
  lwd=0.5);
plot(x,y,type='n',xlab='',ylab='',
  asp=rxy,axes=assi,frame.plot=cornice)}
```

Grafici di funzioni

Il grafico di una funzione `f` (di R o definita da noi) si ottiene con la riga

```
lines(x,f(x))
```

adattando i parametri di ambiente (dimensioni degli intervalli per l'ascissa e per l'ordinata, colori, linee ausiliarie) secondo le necessità. Il parametro `x` deve essere un vettore di valori reali che matematicamente siano ad esempio $(x_1, \dots, x_k)$; `lines` congiunge i punti $(x_1, f(x_1)), \dots, (x_k, f(x_k))$ con segmenti di rette. Se i punti di `x` sono sufficientemente vicini, avremo l'impressione di una curva continua. In genere, se la funzione non oscilla troppo, è sufficiente una risoluzione di 0.01, definendo `x=seq(a,b,by=0.01)`.

Per figure piane possiamo usare con grande vantaggio la rappresentazione complessa. Per un grafico di funzioni possiamo scrivere

```
lines(x+f(x)*1i)
```

La comodità della forma complessa sta soprattutto nel fatto che possiamo così facilmente eseguire operazioni geometriche, le quali nel piano sono tutte esprimibili mediante operazioni algebriche con numeri complessi. Per spostare il grafico di 1 verso l'alto è sufficiente

```
lines(1i+x+f(x)*1i)
```

per girarlo di $90°$ è sufficiente

```
lines(1i*(x+f(x)*1i))
```

e così via.

Octobrina elegans

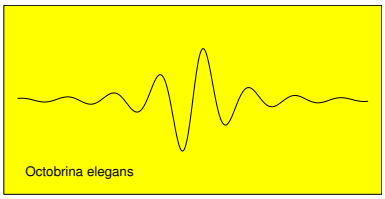
Rappresentiamo il grafico della funzione

$$\bigcirc_x \frac{2\sin 4x}{1+x^2}$$

a cui in R diamo il nome `Oct`:

```
Oct = function (x)
{2*sin(4*x)/(1+x*x)}
```

nell'intervallo $[-6, 6]$.



mediante le istruzioni

```
Gr.postscript('27-octobrina.ps',5,2.5)
latox=c(-6,6); latoy=c(-3,3);
par(bg='yellow',cex=0.4)
Gr(latox,latoy,cornice=T)
t=seq(-6,6,by=0.01)
lines(t+Oct(t)*1i)
text(-6-2.5i,'Octobrina elegans',pos=4)
dev.off();
```

Per vedere la figura sullo schermo bisogna eliminare la prima riga e inserire `locator(1)` prima di `dev.off()`.

Il programma contiene due nuove istruzioni, `text`, che viene usata per aggiungere testi a una figura, e `Gr.postscript`, che crea un file *PostScript* che può essere agganciato al file *Latex*. Il primo parametro di `text` indica il punto a cui si riferisce la posizione del testo; se il parametro `pos` è uguale a 4, il testo viene inserito alla destra del punto di riferimento. Quando questo parametro manca, il testo viene centrato nel punto. Vedere `?text` per le altre possibilità.

Per ottenere una scritta piccola abbiamo modificato il parametro grafico `cex` (nella terza riga); `cex` è un'abbreviazione per *character expansion*.

Curve piane parametrizzate

Una *curva parametrizzata* in $\mathbb{R}^2$ è un'applicazione $\varphi : I \longrightarrow \mathbb{R}^2$, dove I è un intervallo di $\mathbb{R}$. Per ogni $t \in I$ abbiamo un punto $\varphi(t)$ del piano, le cui coordinate $x(t)$ ed $y(t)$ dipendono da t e sono, appunto, legate alla φ dalla relazione $\varphi(t) = (x(t), y(t))$. In questo modo sono definite due funzioni $x : I \longrightarrow \mathbb{R}^2$ e $y : I \longrightarrow \mathbb{R}^2$ che a loro volta determinano la φ. Spesso si scrive allora

$$x = x(t)$$
$$y = y(t)$$

Notiamo subito che il grafico di una funzione reale $f : I \longrightarrow \mathbb{R}^2$ definita su un intervallo è un caso speciale di curva parametrizzata che può essere rappresentato nella forma

$$x = t$$
$$y = f(t)$$

R si presta particolarmente bene per la rappresentazione di curve piane parametrizzate, perché, una volta definito l'intervallo che si vuole usare e che deve essere rappresentato come successione finita `t` di punti, è sufficiente l'istruzione

```
lines(x(t),y(t))
```

nel programma per ottenere la curva.

Anche per le curve parametrizzate utilizzeremo la notazione complessa, quindi

$$\varphi(t) = x(t) + y(t)i$$

e nel disegno della curva in R useremo

```
lines(x(t)+y(t)*1i)
```

ma anche, se la φ è rappresentata direttamente in R come funzione `phi` a valori complessi, istruzioni della forma

```
lines(phi(t))
```

Bibliografia

(13558) **J. Friedl:** Mastering regular expressions. O'Reilly 1997.

13936 **L. Wall/T. Christiansen/J. Orwant:** Programming Perl. O'Reilly 2000.

FONDAMENTI DI INFORMATICA

Corso di laurea in matematica — Anno accademico 2004/05 — Numero 8

Curve di livello

Curve di livello di una funzione f in due variabili non sono altro che gli insiemi della forma $(f = k)$, dove k è una costante. In particolare per $k = 0$ si ottengono gli zeri di f; spesso però è interessante guardare come variano queste curve con il variare di k e soprattutto in quel caso, cioè quando si disegnano le curve $(f = k)$ contemporaneamente per più valori di k, in modo simile come in un atlante si disegnano curve di punti con la stessa altezza o con la stessa quantità di precipitazioni o con la stessa temperatura media ecc., si parla di curve di livello.

In R a questo scopo si può usare la funzione `contour` che, nella versione più semplice, si usa come nella funzione `Gr.livelli` che creiamo per la nostra libreria:

```
Gr.livelli = function (x,y,f,
  valori=NULL,livelli=0,
  scritte=1,colore=1,spessore=0.4)
{if (identical(valori,NULL))
  valori=outer(x,y,f);
contour(x,y,valori,levels=livelli,
  col=colore,lwd=spessore,
  labcex=0.6*scritte,
  drawlabels=(scritte>0),add=T)}
```

Il significato dei parametri `colore` e `spessore` (della matita) è chiaro. Il parametro `scritte` riguarda i parametri `drawlabels` e `labcex` della funzione originale, di cui il primo, se posto uguale a `TRUE`, implica che sulle linee di livello vengono indicati i valori che corrispondono a quei livelli, mentre `labcex` indica la grandezza delle scritte, di cui abbiamo bisogno anche noi perché il valore preimpostato per la visualizzazione sullo schermo risulta troppo grande per le immagini che otteniamo con `Gr.postscript`; per queste ultime sceglieremo quindi `scritte=0.5`. Si noti che nella nostra funzione `drawlabels` è definito attraverso l'espressione booleana `(scritte>0)`; per ottenere una figura senza scritte sulle curve di livello useremo `scritte=0`.

Nella funzione `contour` abbiamo posto l'argomento `add` uguale a `T`, perché in questo modo possiamo esguire più volte il comando `contour` (o `Gr.livelli`) sulla stessa grafica senza cancellare il contenuto precedente; `labcex` riguarda la grandezza delle scritte con cui vengono indicati i valori dei livelli.

`livelli` (nell'originale `levels`) è un singolo numero o un vettore di numeri che rappresenta l'insieme dei livelli che vogliamo disegnare; con `livelli=0` (scelta prestabilita in `Gr.livelli`) viene mostrato l'insieme degli zeri di f, con `livelli=c(0,1,2)` gli insiemi $(f = 0)$, $(f = 1)$ e $(f = 2)$.

Dobbiamo ancora spiegare i primi tre parametri di `contour`. `x` e `y` sono vettori di numeri che corrispondono matematicamente a due successioni finite $(x_1, \dots, x_m)$ e $(y_1, \dots, y_n)$. Il terzo argomento (il nostro `valori`) deve essere una matrice

$$\begin{pmatrix} v_{11} & \dots & v_{1m} \\ \dots & \dots & \dots \\ v_{n1} & \dots & v_{nm} \end{pmatrix}$$

di n righe ed m colonne; il coefficiente v_{ij} è il valore nel punto (x_j, y_i) della funzione di cui vogliamo disegnare i livelli. Questa matrice è il risultato di un comando `outer(x,y,f)`.

La funzione `Gr.livelli` è stata predisposta per essere utilizzata in due modi: possiamo o indicare la funzione `f` (e allora `outer` viene chiamato all'interno di `Gr.livelli`), oppure usare come parametro `valori` una matrice generata con un `outer` precedente, ad esempio quando, in più esecuzioni con la stessa matrice, vogliamo colorare le curve di livello ogni volta in modo diverso.

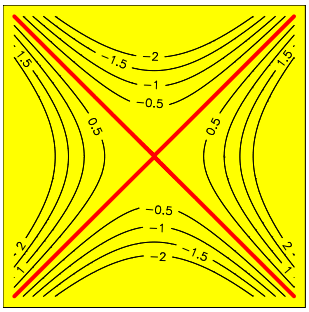
L'iperbole $x^2 - y^2 = k$

Per ogni $k > 0$ questa equazione rappresenta un'iperbole equilatera con asse uguale all'asse delle x. Per $k < 0$ invece abbiamo un'equazione dello stesso tipo con x ed y invertite, che quindi corrisponde a un'iperbole con asse uguale all'asse delle y.

Per $k = 0$ infine l'equazione diventa

$$x^2 - y^2 = (x - y)(x + y) = 0$$

e rappresenta le due rette $y = x$ ed $y = -x$.



Otteniamo la figura che mostra le curve di livello della funzione definita da $f(x, y) = x^2 - y^2$ con questo programma in R:

```
Gr.postscript('28-iperboli.ps',4,4)
lato=c(-2,2); par(bg='yellow')
Gr(lato,lato,cornice=T)
t=seq(-2,2,by=0.01)
f = function (x,y) x*x-y*y
valori=outer(t,t,f)
Gr.livelli(t,t,valori=valori,
  scritte=0.5,
  livelli=setdiff(seq(-2,2,by=0.5),0))
Gr.livelli(t,t,valori=valori,
  spessore=2,colore='red',scritte=0)
dev.off()
```

L'analisi dei livelli avviene sul quadrato $[-2, 2] \times [-2, 2]$ suddiviso in ogni coordinata con una risoluzione di 0.01:

```
t=seq(-2,2,by=0.01)
```

Usiamo la funzione `Gr.livelli` due volte. Infatti la prima volta disegniamo in nero e con le scritte tutti i livelli diversi da 0, la seconda volta solo le due rette $y = \pm x$, in rosso e senza scritte. Nella prima istruzione i livelli scelti corrispondono all'insieme

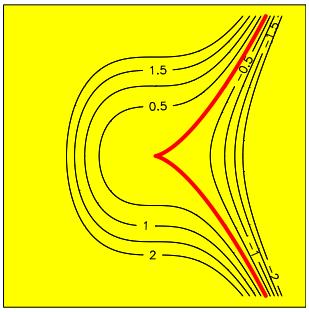
$$\{-2, -1.5, -1, -0.5, 0.5, 1, 1.5, 2\}$$

usando la funzione `setdiff` vista a pagina 7.

In questo numero

$y^2 = x^3$

Questa curva è la cuspide o *parabola di Neill*:

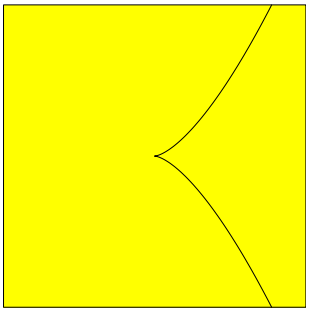


La possiamo anche ottenere con la parametrizzazione

$$x = t^2$$
$$y = t^3$$

mediante il semplicissimo programma

```
Gr.postscript(28-cuspide-par.ps',4,4)
lato=c(-2,2); par(bg="yellow")
Gr(lato,lato,cornice=T)
t=seq(-10,10,by=0.01)
lines(t^2,t^3)
dev.off()
```



Il nodo $y^2 = x^3 + x^2$

Con lo stesso programma che a pagina 28 abbiamo usato per disegnare le iperboli, sostituendo solo la riga riguardante la funzione con

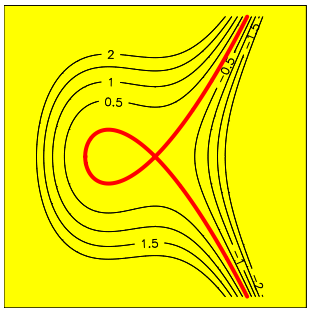
```
f = function (x,y) y^2-x^3-x^2
```

possiamo disegnare la famiglia di curve le cui equazioni hanno la forma

$$y^2 = x^3 + x^2 + k$$

In questo caso la curva singolare, in rosso, è la curva

$$y^2 = x^3 + x^2$$



I livelli indicati sono i livelli della funzione $f := \bigcirc_{(x,y)} y^2 - x^3 - x^2$. Si vede che la f assume valori positivi alla destra della curva singolare e valori negativi alla sua sinistra.

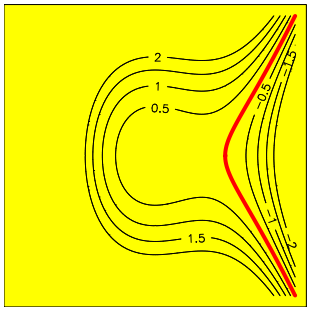
Provare da soli, eventualmente ingrandendo la figura, a scoprire il segno di $f(x,y)$ all'interno del nodo.

$y^2 = x^3 - x^2$

Purtroppo talvolta il software per la rappresentazione di curve può ingannare, e ciò non vale solo per R, ma anche per Maple, Mathematica ecc. In questo caso per esempio è chiaro che l'origine è una soluzione dell'equazione

$$y^2 = x^3 - x^2$$

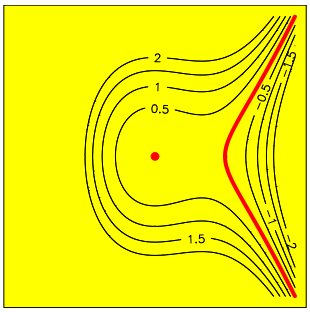
eppure non appare nella figura (dovrebbe essere un punto rosso nell'origine). Quindi non ci si può fidare ciecamente dei disegni che si ottengono con questi programmi.



Non siamo riusciti ad ottenere l'origine come punto della curva mediante il programma; dobbiamo quindi, appellandoci ai risultati della matematica, aggiungerlo a mano:

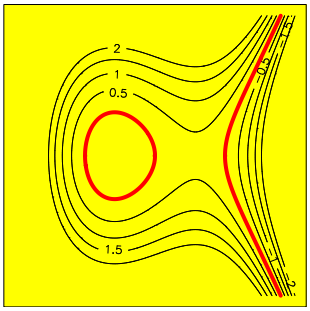
```
f = function (x,y) y^2-x^3+x^2
...
points(0,0,pch=20,cex=0.8,col='red')
dev.off()
```

ottenendo così



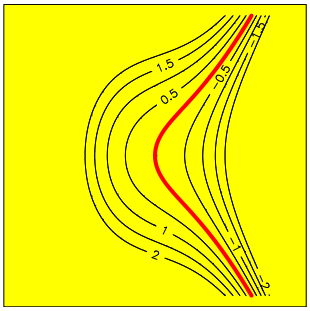
$y^2 = x^3 + x$

Anche la curva $y^2 = x^3 + x$ la possiamo ottenere cambiando semplicemente la funzione nel nostro programma.



$y^2 = x^3 - x$

Vediamo qui che una semplice modificazione dell'equazione implica un cambiamento sostanziale nella forma della curva. Alcune differenze scompaiono comunque se si considerano queste curve come curve definite sul campo dei numeri complessi. Come una retta complessa corrisponde a un piano reale, così una curva complessa corrisponde a una superficie reale. Completate nel senso della geometria proiettiva le cubiche finora viste diventano *superficie di Riemann*.



Nelle applicazioni, ad esempio in statistica, ancora più importanti sono però le forme reali, la cui classificazione è molto difficile.

Il foglio di Cartesio

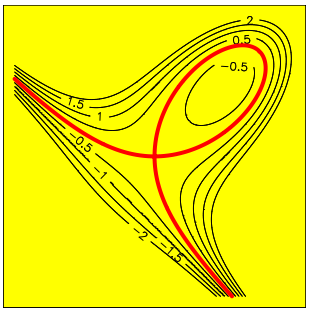
Questa curva ha l'equazione

$$x^3 + y^3 - 3xy = 0$$

Come si verifica facilmente, la curva può essere rappresentata in forma parametrica mediante

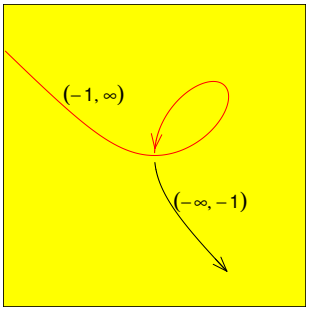
$$x = \frac{3t}{1+t^3}$$

$$y = \frac{3t^2}{1+t^3}$$



Già in questo esempio si vede però che il legame tra equazione e rappresentazione parametrica può essere abbastanza complicato. Infatti il denominatore $1 + t^3$ ha uno zero in $t = -1$, per cui il dominio dei parametri si decompone nei due intervalli aperti $(-\infty, -1)$ e $(-1, \infty)$.

La parte della curva che corrisponde a $(-1, \infty)$, colorato in rosso nella figura che segue, inizia nell'infinito a sinistra, passa con $t = 0$ per l'origine, formando un cappio che ritorna all'origine senza raggiungerlo. La parte che corrisponde a $(-\infty, -1)$, colorata in nero, inizia (per $t << 0$) vicino all'origine e si estende verso destra con t che converge a -1 da sinistra. Anche questa situazione può essere riformulata nel linguaggio della geometria proiettiva.



Il programma utilizza le funzioni `arrows` per le frecce e `text` per l'inserimento di testi.

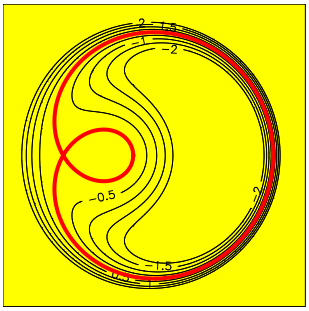
```
phi = function (t)
3*t*(1+1i*t)/(1+t^3)

Gr.postscript('29-cartesio-par.ps',4,4)
lato=c(-3,3); par(bg='yellow')
Gr(lato,lato,cornice=T)
x1=Re(phi(15)); y1=Im(phi(15))
x2=Re(phi(20)); y2=Im(phi(20))
arrows(x1,y1,x2,y2,
  length=0.08,angle=20,col='red')
x1=Re(phi(-1.7)); y1=Im(phi(-1.7))
x2=Re(phi(-1.6)); y2=Im(phi(-1.6))
arrows(x1,y1,x2,y2,
  length=0.08,angle=20)
t=seq(-0.7,50,by=0.01)
lines(phi(t),col='red')
t=seq(-20,-1.6,by=0.01)
text(-1.3+1.3i,
  expression(group('(',
  list(-1,infinity),')')),cex=0.6)
text(1.2-1i,
  expression(group('(',
  list(-infinity,-1),')')),cex=0.6)
lines(phi(t))
dev.off()
```

La chiocciola di Pascal

La chiocciola di Pascal ha l'equazione

$$(x^2 + y^2 - 2x)^2 = x^2 + y^2$$




```
Gr.postscript('30-pascal.ps',4,4)
latox=c(-0.7,3.3); latoy=c(-2,2);
par(bg='yellow')
Gr(latox,latoy,cornice=T)
x=seq(-0.7,3.3,by=0.01)
y=seq(-2,2,by=0.01)
f = function (x,y)
{u=x^2+y^2-2*x; u^2-x^2-y^2}
valori=outer(x,y,f)
Gr.livelli(x,y,valori=valori,
  scritte=0.5,
  livelli=setdiff(seq(-2,2,by=0.5),0))
Gr.livelli(x,y,valori=valori,spessore=2,
  colore='red',scritte=0)
dev.off()
```

Lemniscate

Una *lemniscata* è una curva con equazione

$$(x^2 + y^2)^2 = f(x, y)$$

dove

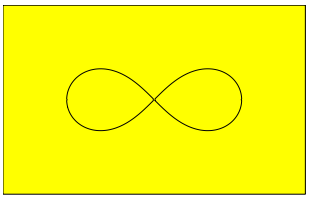
$$f(x, y) = \alpha x^2 + 2\beta xy + \gamma y^2$$

è una forma quadratica ellittica o iperbolica.

Si dimostra facilmente che la lemniscata si ottiene dalla conica $f(x, y) = 1$ mediante inversione al cerchio unitario; nel caso ellittico anche l'origine (che non può essere ottenuto dall'ellisse mediante inversione) soddisfa l'equazione.

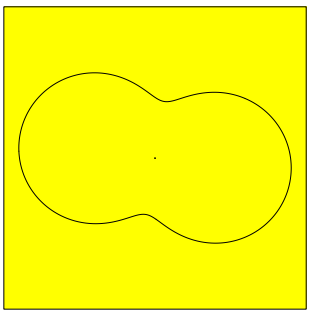
La lemniscata si dice *iperbolica* (o di Bernoulli) se la conica $f(x, y) = 1$ è un'iperbole, ed *ellittica* se la conica è un'ellisse. La lemniscata ellittica ha varie applicazioni interessanti, ad esempio in statistica.

$$(x^2 + y^2)^2 = x^2 - y^2$$



```
Gr.postscript('30-lem-iperbolica.ps',
  4,2.5)
latox=c(-1.5,1.5); latoy=c(-1,1)
par(bg='yellow')
Gr(latox,latoy,cornice=T)
t=seq(-1.5,1.5,by=0.01)
f = function (x,y) (x^2+y^2)^2-(x^2-y^2)
valori=outer(t,t,f)
Gr.livelli(t,t,valori=valori,scritte=0)
dev.off()
```

$$(x^2 + y^2)^2 = 15x^2 - 4xy + 3y^2$$



Si noti l'origine che soddisfa l'equazione.

```
Gr.postscript('30-lem-ellittica.ps',4,4)
latox=c(-4,4); latoy=c(-4,4)
par(bg="yellow")
Gr(latox,latoy,cornice=T)
t=seq(-4,4,by=0.01)
f = function (x,y)
  (x^2+y^2)^2-(15*x^2-4*x*y+3*y^2)
valori=outer(t,t,f)
Gr.livelli(t,t,valori=valori,scritte=0)
dev.off()
```

L'equazione simmetrica

$$(x^2 + y^2)\, 2 = x^2 + y^2$$

può essere scritta nella forma

$$(x^2 + y^2)(x^2 + y^2 - 1) = 0$$

e rappresenta un cerchio unito all'origine.

Spirale di Archimede

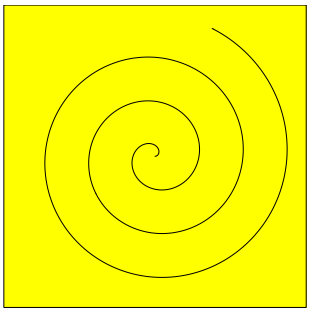
Una curva in *coordinate polari* data da una relazione $r = f(\alpha)$ può essere immediatamente riscritta in forma parametrica:

$$x = f(t)\cos t$$
$$y = f(t)\sin t$$

La *spirale di Archimede* $r = \alpha$ corrisponde quindi a

$$x = t\cos t$$
$$y = t\sin t$$

per $t \geq 0$.

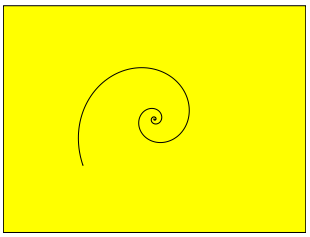


```
Gr.postscript('30-archimede.ps',4,4)
lato=c(-20,20); par(bg='yellow')
Gr(lato,lato,cornice=T)
t=seq(0,20,by=0.01)
lines(t*cos(t),t*sin(t))
dev.off()
```

Spirale logaritmica

La *spirale logaritmica* $r = e^{\frac{\alpha}{4}}$ possiede la rappresentazione parametrica

$$x = e^{\frac{t}{4}}\cos t$$
$$y = e^{\frac{t}{4}}\sin t$$



```
Gr.postscript('30-logspir.ps',4,3)
latox=c(-20,20); latoy=c(-15,15);
par(bg='yellow')
Gr(latox,latoy,cornice=T)
t=seq(-10,10,by=0.01)
lines(exp(t/4)*cos(t),exp(t/4)*sin(t))
dev.off()
```

La cissoide

La *cissoide* corrisponde in coordinate polari alla relazione

$$r = \sin\alpha\tan\alpha$$

e possiede quindi la rappresentazione parametrica

$$x = \sin^2\alpha$$
$$y = \sin^2\alpha\cdot\tan\alpha$$

Ponendo $t = \tan\alpha$ abbiamo

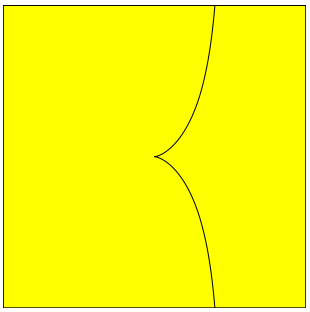
$$t^2 - t^2\sin^2\alpha = \sin^2\alpha$$

cosicché

$$\sin^2\alpha = \frac{t^2}{1+t^2}$$

e la rappresentazione parametrica diventa

$$x = \frac{t^2}{1+t^2}$$
$$y = \frac{t^3}{1+t^2}$$

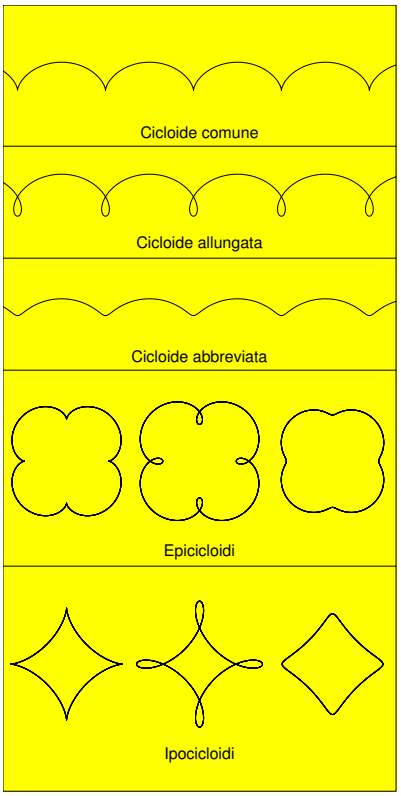


```
Gr.postscript('30-cissoide.ps',4,4)
lato=c(-2,2); par(bg='yellow')
Gr(lato,lato,cornice=T)
t=seq(-4,4,by=0.01)
u=1+t^2
lines(t^2/u,t^3/u)
dev.off()
```

Cicloidi

Cicloidi sono curve che si ottengono facendo rotolare un cerchio su una curva guida. Se la distanza del punto mobile è maggiore del raggio del cerchio, si ottiene una cicloide *allungata*, se è minore una cicloide *abbreviata*. Una cicloide *comune* è una cicloide in cui la curva guida è una retta. Se la curva guida è un cerchio, si ottengono *epicicloidi* facendo rotolare il cerchio mobile all'esterno della guida, e *ipocicloidi* facendolo rotolare internamente. Le cicloidi sono già state studiate da Albrecht Dürer e rivestono grande importanza nella costruzione delle macchine, ad esempio nella teoria degli ingranaggi. Presentiamo solo alcune delle molte forme possibili:

Cicloide comune	$x = t - \sin t$ $y = 1 - \cos t$
Cicloide all.	$x = t - 1.5 \sin t$ $y = 1 - 1.5 \cos t$
Cicloide abbr.	$x = t - 0.6 \sin t$ $y = 1 - 0.6 \cos t$
Epicicloide	$x = 5 \cos t - \cos 5t$ $y = 5 \sin t - \sin 5t$
Epicicloide all.	$x = 5 \cos t - 1.5 \cos 5t$ $y = 5 \sin t - 1.5 \sin 5t$
Epicicloide abbr.	$x = 5 \cos t - 0.6 \cos 5t$ $y = 5 \sin t - 0.6 \sin 5t$
Ipocicloide	$x = 3 \cos t + \cos 3t$ $y = 3t - \sin 3t$
Ipocicloide all.	$x = 3 \cos t + 1.5 \cos 3t$ $y = 3t - 1.5 \sin 3t$
Ipocicloide	$x = 3 \cos t + 0.6 \cos 3t$ $y = 3t - 0.6 \sin 3t$




```
Gr.postscript('31-cicloidi.ps',5.2,10.4)
latox=c(0,26); latoy=c(0,52)
par(cex=0.5,bg='yellow')
Gr(latox,latoy,cornice=T)
for (k in c(14,28,36,44)) abline(h=k)
t=seq(-2*pi,10*pi,by=0.01)
lines(48i+(t-sin(t))+1i*(1-cos(t)))
text(13+45i,'Cicloide comune')
lines(39.5i+(t-1.5*sin(t))+
  1i*(1-1.5*cos(t)))
text(13+37i,'Cicloide allungata')
lines(31.5i+(t-0.6*sin(t))+
  1i*(1-0.6*cos(t)))
text(13+29i,'Cicloide abbreviata')
lines(3.5+21.5i+0.75*(5*cos(t)-cos(5*t)
  +1i*(5*sin(t)-sin(5*t))))
text(13+15i,'Epicicloidi')
lines(13+21.5i+0.75*(5*cos(t)-
  1.5*cos(5*t)
  +1i*(5*sin(t)-1.5*sin(5*t))))
lines(22.5+21.5i+0.75*(5*cos(t)-
  0.6*cos(5*t)
  +1i*(5*sin(t)-0.6*sin(5*t))))
lines(3.5+7i+3*cos(t)+cos(3*t)
  +1i*(3*sin(t)-sin(3*t)))
lines(13+7i+3*cos(t)+1.5*cos(3*t)
  +1i*(3*sin(t)-1.5*sin(3*t)))
lines(22.5+7i+3*cos(t)+0.6*cos(3*t)
  +1i*(3*sin(t)-0.6*sin(3*t)))
text(13+0.5i,'Ipocicloidi')
dev.off()
```

Proiezioni lineari $\mathbb{R}^n \longrightarrow \mathbb{R}^2$

Un'applicazione lineare $P : \mathbb{R}^n \longrightarrow \mathbb{R}^2$ è determinata dalle immagini

$$w_1 := P(\delta_1), \ldots, w_n := P(\delta_n)$$

dei vettori $\delta_1, \ldots, \delta_n$ della base standard. Per $x = (x_1, \ldots, x_n) \in \mathbb{R}^n$ si ha allora

$$x = x_1\delta_1 + \ldots + x_n\delta_n$$

e quindi

$$P(x) = x_1 w_1 + \ldots + x_n w_n$$

La traduzione in R è immediata e incredibilmente semplice:

```
G.prolin = function (w)
function (x) sum(x*w)
```

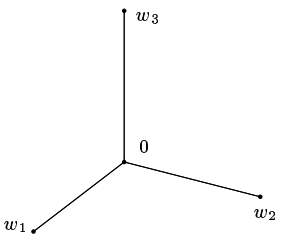
In questa funzione `w` deve essere un vettore di numeri complessi.

Per le proiezioni $\mathbb{R}^3 \longrightarrow \mathbb{R}^2$ nel seguito useremo l'applicazione determinata da

$$w_1 = (-0.6, -0.46)$$
$$w_2 = (0.9, -0.23)$$
$$w_3 = (1, 0)$$

che otteniamo mediante l'istruzione

```
Pro32 = G.prolin(c(-0.6-0.46i,
  0.9-0.23i,1))
```



Assumiamo quindi di voler proiettare una curva spaziale

$$x = x(t)$$
$$y = y(t)$$
$$z = z(t)$$

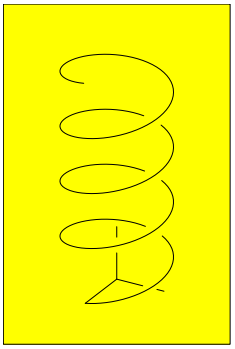
sul piano. In R allora usiamo le seguenti istruzioni:

```
f = function (t)
  Pro32(c(x(t),y(t),z(t))
u=sapply(t,f); lines(u)
```

L'elica

$$x = \cos t$$
$$y = \sin t$$
$$z = \frac{t}{6}$$

Con il procedimento dell'articolo precedente otteniamo la figura



```
w=c(-0.6-0.46i,0.9-0.23i,1i)

Pro32 = G.prolin(w)

Gr.postscript('31-elica.ps',3,4.5)
latox=c(-2,2); latoy=c(-1,5);
par(bg='yellow')
Gr(latox,latoy,cornice=T)
t=seq(0,8*pi,by=0.1)
t=c(t[t<pi-0.3],NA,t[t>pi])     #
t=c(t[t<3*pi-0.3],NA,t[t>3*pi]) #
t=c(t[t<5*pi-0.3],NA,t[t>5*pi]) #
f = function (t)
  Pro32(c(cos(t),sin(t),t/6))
u=sapply(t,f)
lines(u)
lines(c(0,w[1])); lines(c(0,0.55*w[2]))
lines(c(0.85*w[2],w[2]))
lines(c(0,0.5*w[3]))
lines(c(0.8*w[3],w[3]))
dev.off()
```

Nel programma le tre righe che terminano con commento sono usate per eliminare le sovrapposizioni sull'elica; sfruttiamo qui che `lines` interrompe il disegno di un poligono quando incontra un valore `NA`. Abbiamo perciò introdotto valori `NA` nella sequenza `t` dei parametri che vengono successivamente tramandati all'elenco dei punti da congiungere.

Data e tempo

Per la data attuale esistono le funzioni `date`, `Sys.time` e `Sys.Date`. La prima restituisce una stringa come in

```
d=date()
print(d)
# Fri May 27 19:05:33 2005
```

le altre due utilizzano un formato speciale per il quale bisogna consultare l'aiuto o gli articoli di Grothendieck/Petzold e Ripley/Hornik. Non confondere `Sys.time` con `system.time` vista a pagina 17.

Liste

Abbiamo già usato più volte le *liste*. Esse, a differenza dai vettori, possono contenere elementi di tipo diverso, anche altre liste e vettori. Per creare una lista si usa la funzione `list`:

```
a=list(1,2,c(3,4),list(5,6,7),8)
```

Se vogliamo, usando `list` ed `eval`, possiamo quindi facilmente imitare il Lisp con R. L'i-esimo elemento di una lista `a` è `a[[1]]`; gli elementi di una lista (come peraltro quelli di un vettore) possono anche essere indicati mediante *voci* e nel caso delle liste l'accesso può allora avvenire mediante l'operatore `$`:

```
a=list(nome='Rossi',voti=c(27,25,30))
print(a$nome)
# "Rossi"
print(a$voti)
# 27 25 30
```

Le doppie parentesi quadre possono essere usate solo per estrarre una singola componente della lista, perciò `a[[1:2]]` non è corretto. Se usiamo solo parentesi quadre singole, otteniamo una lista, ad esempio

```
a=list(c(1,2),c(2,3),c(4,5))
print(a[2:3])

# [[1]]
# [1] 2 3

# [[2]]
# [1] 4 5
```

Anche `a[1]` è una lista, come si vede dalla forma dell'output:

```
print(a[1])

# [[1]]
# [1] 1 2
```

Liste possono essere concatenate con `c`. Infatti, se uno degli argomenti di `c` è una lista, `c` restituisce una lista tranne nel caso che si abbia scelto l'opzione `recursive=T`.

```
a=list('alfa',7)
b=c(a,c(1,2,3))
print(b)

[[1]]
[1] "alfa"

[[2]]
[1] 7

[[3]]
[1] 1

[[4]]
[1] 2

[[5]]
[1] 3
```

Invece

```
b=c(a,c(1,2,3),recursive=T)
print(b)

# [1] "alfa" "7" "1" "2" "3"
```

`unlist` trasforma una lista in un vettore, conserva però gli attributi; questi possono essere rimossi con `as.vector`:

```
a=list(ab=80, dm=c(7,1,2))
u=unlist(a)
print(u)

# ab dm1 dm2 dm3
# 80  7   1   2

v=as.vector(u)
print(v)
# 80 7 1 2
```

L'elenco delle voci di una lista o di un vettore lo si ottiene con `names`:

```
z=c(x=2,y=5)
print(names(z))
# "x" "y"
```

Files e cartelle

Abbiamo visto a pagina 10 come scrivere su un file con `cat`.

Con `file.info(nome)` si ottiene una lista con informazioni sul file `nome`, di cui le più importanti sono:

`$size` ... lunghezza del file in bytes;
`$isdir` ... `T` se si tratta di una cartella;
`$uname` ... nome dell'utente.

Per l'esistenza di un file di testo (in verità controlliamo solo se il file esiste e che non si tratta di una cartella) possiamo usare

```
# F se si tratta di una cartella.
File.esiste = function (nome)
{info=file.info(nome)
if (is.na(info$size)) F
else !info$isdir}
```

Con `readChar` possiamo leggere caratteri da un file; il secondo parametro di questa funzione è un vettore di interi che determinala lunghezza delle stringhe lette. Con `readChar(nome,2000)` si ottiene un'unica stringa che contiene i primi 2000 caratteri del file `nome`, con `readChar(nome,c(10,5,8))` un vettore di tre stringhe di lunghezza complessiva 23, di cui la prima contiene i primi 10 caratteri del file ecc. Possiamo così definire una nostra funzione:

```
File.leggi = function (nome,dir='')
{if (Tr.pos('/$',dir)>0) sep=''
else if (dir!='') sep='/' else sep=''
nome=paste(dir,nome,sep=sep)
n=file.info(nome)$size;
readChar(nome,n)}
```

Con il secondo parametro, opzionale, possiamo indicare una cartella che contiene il file; per una corretta combinazione con il nome del file usiamo la nostra funzione `Tr.pos`.

Si osservi che anche sotto Windows il separatore nei nomi dei files deve essere `/` (oppure `\\`) e non `\`.

Con `readLines(nome)` si ottiene un vettore di parole che contiene le righe del file `nome`.

Già nel file *alfa* a pagina 2 abbiamo usato la funzione `dir` che restituisce un vettore di parole che, con l'opzione `recursive=T`, contiene i nomi dei files contenuti in una cartella e nelle sue sottocartelle, ma non i nomi delle sottocartelle. La sintassi usata era

```
dir('Libreria',full.names=T,recursive=T)
```

Numeri esadecimali

Abbiamo già imparato a pagina 11 come ottenere la rappresentazione esadecimale di un numero naturale con `sprintf`. Possiamo adesso risolvere anche il compito inverso, cioè la conversione di una stringa che contiene la rappresentazione esadecimale di un numero. Assumiamo che sia data la stringa

```
'23f5d'
```

Mediante `strsplit` da essa otteniamo il vettore

```
c('2','3','f','5','d')
```

Formiamo adesso il vettore

```
c('0',...,'9','a','b','c','d','e','f')
```

La posizione di una cifra in questo vettore può essere calcolata con `match` ed è uguale al numero che corrisponde a questa cifra (quindi 7 per `'7'` e 11 per `'b'`), diminuita di uno. A questo punto possiamo applicare `M.horner`. Definiamo quindi la seguente funzione:

```
Ma.esa = function (rapp)
{rapp=tolower(rapp)
u=strsplit(rapp,'',perl=T)[[1]]
lettercifre=c(0:9,
  'a','b','c','d','e','f')
v=sapply(u,function (x)
  match(x,lettercifre)-1)
v=as.numeric(v)
M.horner(v,16)}
```

La possiamo provare con

```
for (x in c('0','a0','10e','f0ae'))
print(Hex(x))
# 0
# 160
# 270
# 61614
```

La funzione `match` è già stata introdotta a pagina 9.

Bibliografia

440/1 **I. Bronshtein/K. Semendyayev:** Taschenbuch der Mathematik I. Deutsch 1987.

(2673) **M. do Carmo:** Differential geometry of curves and surfaces. Prentice-Hall 1976.

17124 **G. Grothendieck/T. Petzold:** Date and time classes in R.
R News 4/1 (2004), 29-31.

(1451) **E. Kreyszig:** Differential geometry. Dover 1991.

17057 **U. Ligges:** Programmieren mit R. Springer 2005.

16644 **H. Pottmann/J. Wallner:** Computational line geometry. Springer 2001.

17125 **B. Ripley/K. Hornik:** Date-time classes. R News 1/2 (2001), 8-11.

FONDAMENTI DI INFORMATICA

Tabelle

La struttura fondamentale per rappresentare dati statistici in R sono le *tabelle* (in inglese *data frames*). Molte funzioni statistiche di R operano su tabelle. Esse sono simili alle matrici dalle quali si distinguono per il fatto che le colonne possono corrispondere a tipi diversi. Gli elementi in ogni colonna invece sono dello stesso tipo.

Formalmente una tabella è una *lista* di vettori della stessa lunghezza con nomi distinti per le colonne con cui queste possono essere identificate.

Per creare una tabella si usa la funzione `data.frame` che accetta, in forma di matrici o vettori della stessa lunghezza, i nomi e i contenuti delle colonne della tabella. Esempi:

```
tab = data.frame(a=1:5, b=11:15)
print(tab)

#   a  b
# 1 1 11
# 2 2 12
# 3 3 13
# 4 4 14
# 5 5 15
```

```
A=Mm(1:15, col=3)
tab=data.frame(A)
print(tab)

#    X1 X2 X3
#  1  1  2  3
#  2  4  5  6
#  3  7  8  9
#  4 10 11 12
#  5 13 14 15
```

```
A=Mm(1:15, col=3)
tab=data.frame(A,z=21:25)
print(tab)

#   X1 X2 X3  z
# 1  1  2  3 21
# 2  4  5  6 22
# 3  7  8  9 23
# 4 10 11 12 24
# 5 13 14 15 25
```

```
A=Mm(1:15, col=3)
tab=data.frame(U=A,z=21:25)
print(tab)

#   U.1 U.2 U.3  z
# 1   1   2   3 21
# 2   4   5   6 22
# 3   7   8   9 23
# 4  10  11  12 24
# 5  13  14  15 25
```

```
A=Mm(1:15, col=3)
tab=data.frame(comune=c('Parma','Pisa',
  'Roma','Crema','Trento'),A=A)
print(tab)

#   comune A.1 A.2 A.3
# 1  Parma  1  2  3
# 2   Pisa  4  5  6
# 3   Roma  7  8  9
# 4  Crema 10 11 12
# 5 Trento 13 14 15
```

Tabelle possono essere indicizzate sia come liste che come matrici, con qualche variazione; dipende dal contesto quale variante si preferirà.

```
tab=data.frame(comune=c('Parma','Pisa',
  'Bari','Crema','Trento','Lana'),
  ab=c(170,92,332,34,106,10),
  alt=c(55,4,5,79,194,316))

print(tab)

#   comune  ab alt
# 1  Parma 170  55
# 2   Pisa  92   4
# 3   Bari 332   5
# 4  Crema  34  79
# 5 Trento 106 194
# 6   Lana  10 316

print(tab$alt)
# 55 4 5 79 194 316

print(tab[,3])
# Come precedente.

print(tab[3])
#   alt
# 1  55
# 2   4
# 3   5
# 4  79
# 5 194
# 6 316

print(tab[[3]])
# 55 4 5 79 194 316

print(tab[2:3])
#    ab alt
# 1 170  55
# 2  92   4
# 3 332   5
# 4  34  79
# 5 106 194
# 6  10 316

print(tab[,2:3])
# Come precedente.

print(tab[1,3])
# 55

print(tab[[1,3]])
# 55
```

Colonne non numeriche di una tabella vengono considerate come *fattori* (variabili non-numeriche, dette anche *categoriali*, di cui R registra la frequenza di apparizione):

```
print(tab$comune)
# [1] Parma Pisa Bari Crema Trento Lana
# Levels: Bari Crema Parma Pisa Trento
            Lana

print(tab[,1])
# Come precedente.

print(tab[1])
#   comune
# 1  Parma
# 2   Pisa
# 3   Bari
# 4  Crema
# 5 Trento
# 6   Lana
```

In questo numero

attach

Creiamo una nuova tabella:

```
tab=data.frame(cat=c('a','b','a',
    'a','p','b'),
  classe=c('I','III','III','I','II',
    'III'),
  prezzo=c(88,62,69,85,70,67))

print(tab)
#   cat classe prezzo
# 1   a      I     88
# 2   b    III     62
# 3   a    III     69
# 4   a      I     85
# 5   p     II     70
# 6   b    III     67

print(tab$cat)
# [1] a b a a p b
# Levels: a b p

print(tab$classe)
# [1] I III III I II III
# Levels: I II III
```

Mediante l'istruzione `attach(tab)` i nomi delle colonne di una tabella diventano direttamente accessibili e non è più necessario indicare la tabella. Bisogna però stare attenti perché aumenta il rischio che i nomi delle colonne si sovrappongano a nomi di altre variabili che dobbiamo ancora usare. Il comando è revocato con `detach(tab)`. I nomi abbreviati sono usati però solo in lettura e, quando appaiono alla sinistra di un assegnamento, questo non ha effetto sulla tabella.

```
print(tab$prezzo)
# 88 62 69 85 70 67

attach(tab)

print(prezzo)
# Come precedente.

prezzo=c(0,0,0,0,0)
print(prezzo)
# 0 0 0 0 0

print(tab)
#   cat classe prezzo
# 1   a      I     88
# 2   b    III     62
# 3   a    III     69
# 4   a      I     85
# 5   p     II     70
# 6   b    III     67

detach(tab)
```

subset

Righe e colonne di una tabella possono essere selezionate con la funzione `subset`.
Per selezionare *righe* da una tabella `tab` si usa la sintassi `righe=subset(tab,cond)`, dove `cond` è un vettore booleano. Il risultato `righe` è a sua volta una tabella. `tab` sia l'ultima tabella definita a pagina 33:

```
righe=subset(tab,cat=='a')
print(righe)
#   cat classe prezzo
# 1   a      I     88
# 3   a    III     69
# 4   a      I     85

print(class(righe))
# "data.frame"

righe=subset(tab,prezzo<80)
print(righe)
#   cat classe prezzo
# 2   b    III     62
# 3   a    III     69
# 5   p     II     70

righe=subset(tab,
  (prezzo<80)&(cat=='a'))
# Non funziona invece &&.
print(righe)
#   cat classe prezzo
# 3   a    III     69

righe=subset(tab,F)
print(righe)
# cat classe prezzo
# <0 rows> (or 0-length row.names)
# Tabella vuota.
```

Per selezionare *colonne* da una tabella `tab` si usa invece la sintassi

```
col=subset(tab,select=v)
```

dove `v` è un vettore che contiene gli indici o i nomi delle colonne selezionate:

```
col=subset(tab,select=c(cat,prezzo))
print(col)
#   cat prezzo
# 1   a     88
# 2   b     62
# 3   a     69
# 4   a     85
# 5   p     70

col=subset(tab,select=c(cat,3))
print(col)
# Come precedente.
```

transform

Con la funzione `transform` si possono aggiungere nuove colonne a una tabella o sostituire colonne esistenti; questa funzione viene usata soprattutto per aggiungere valori derivati dalle colonne già presenti.

```
tab=data.frame(a=1:5,b=c(2,0,1,7,3))
print(tab)
#   a b
# 1 1 2
# 2 2 0
# 3 3 1
# 4 4 7
# 5 5 3

tab1=transform(tab,ab=a*b,qb=b^2)
print(tab1)
#   a b ab qb
# 1 1 2  2  4
# 2 2 0  0  0
# 3 3 1  3  1
# 4 4 7 28 49
# 5 5 3 15  9
```

Se i nomi di colonna indicati sono nomi di colonne già esistenti nella tabella, queste vengono sostituite dai nuovi valori. Usiamo ancora la stessa tabella `tab`:

```
tab1=transform(tab,a=a^2,apb=a+b)
print(tab1)
#    a b apb
# 1  1 2   3
# 2  4 0   2
# 3  9 1   4
# 4 16 7  11
# 5 25 3   8
```

rbind e cbind per tabelle

Per una semplice unione di tabelle si possono usare `rbind` e `cbind` come per matrici.

```
lazio1=data.frame(capoluogo=
  c('Frosinone','Latina','Rieti',
  'Roma','Viterbo'),
  comuni=c(91,33,73,121,60))
print(lazio1)
#   capoluogo comuni
# 1 Frosinone     91
# 2    Latina     33
# 3     Rieti     73
# 4      Roma    121
# 5   Viterbo     60

lazio2=data.frame(sup=
  c(3244,2250,2749,5352,3612),
  dens=c(152,228,55,719,81))
print(lazio2)
#    sup dens
# 1 3244  152
# 2 2250  228
# 3 2749   55
# 4 5352  719
# 5 3612   81

lazio=cbind(lazio1,lazio2)
print(lazio)
#   capoluogo comuni  sup dens
# 1 Frosinone     91 3244  152
# 2    Latina     33 2250  228
# 3     Rieti     73 2749   55
# 4      Roma    121 5352  719
# 5   Viterbo     60 3612   81

basilicata=data.frame(capoluogo=
  c('Matera','Potenza'),
  comuni=c(31,100),sup=c(3446,6548),
  dens=c(60,61))
tab=rbind(lazio,basilicata)
print(tab)
#    capoluogo comuni  sup dens
# 1  Frosinone     91 3244  152
# 2     Latina     33 2250  228
# 3      Rieti     73 2749   55
# 4       Roma    121 5352  719
# 5    Viterbo     60 3612   81
# 11    Matera     31 3446   60
# 21   Potenza    100 6548   61
```

merge

Per unire due tabelle che hanno colonne in comune si usa la funzione `merge`. Essa corrisponde alla *combinazione* (in inglese *join*) di due tabelle nel senso della teoria delle basi di dati, un'operazione fondamentale che può essere descritta matematicamente e implementata in SQL. La sintassi più semplice è illustrata negli esempi; consultare `?merge` per le vari opzioni.

```
lazio1=data.frame(capoluogo=
  c('Frosinone','Latina','Rieti',
  'Roma','Viterbo'),
  comuni=c(91,33,73,121,60))
print(lazio1)
#   capoluogo comuni
# 1 Frosinone     91
# 2    Latina     33
# 3     Rieti     73
# 4      Roma    121
# 5   Viterbo     60

lazio3=data.frame(capoluogo=
  c('Roma','Latina','Rieti',
  'Frosinone','Viterbo'),sup=
  c(5352,2250,2749,3244,3612))
print(lazio3)
#   capoluogo  sup
# 1      Roma 5352
# 2    Latina 2250
# 3     Rieti 2749
# 4 Frosinone 3244
# 5   Viterbo 3612

lazio4=merge(lazio1,lazio3)
print(lazio4)
#   capoluogo comuni  sup
# 1 Frosinone     91 3244
# 2    Latina     33 2250
# 3     Rieti     73 2749
# 4      Roma    121 5352
# 5   Viterbo     60 3612
```

```
tab1=data.frame(a=1:5,b=11:15,c=21:25,
  d=31:35,e=41:45)
print(tab1)
#   a  b  c  d  e
# 1 1 11 21 31 41
# 2 2 12 22 32 42
# 3 3 13 23 33 43
# 4 4 14 24 34 44
# 5 5 15 25 35 45

tab2=data.frame(a=3:5,c=c(23,24,28),
  f=53:55)
print(tab2)
#   a  c  f
# 1 3 23 53
# 2 4 24 54
# 3 5 28 55

tab=merge(tab1,tab2)
print(tab)
#   a  c  b  d  e  f
# 1 3 23 13 33 43 53
# 2 4 24 14 34 44 54
```

Si noti che sono state usate solo le righe compatibili nelle due tabelle di partenza.

Con l'opzione `all=T` anche righe non corrispondenti vengono usate nell'unione; i valori mancanti sono posti uguali a `NA`.

```
tab=merge(tab1,tab2,all=T)
print(tab)
#   a  c  b  d  e  f
# 1 1 21 11 31 41 NA
# 2 2 22 12 32 42 NA
# 3 3 23 13 33 43 53
# 4 4 24 14 34 44 54
# 5 5 25 15 35 45 NA
# 6 5 28 NA NA NA 55
```

Si può chiedere anche solo l'inclusione di righe non corrispondenti della prima tabella con `all.x=T` o della seconda con `all.y=T`.

Nell'impostazione iniziale la tabella unita che si ottiene è ordinata rispetto alle colonne comuni. Per questa ragione usiamo talvolta un'istruzione `tab=merge(tab,tab)` anche soltanto per ordinare la tabella.

read.table

Questa funzione è usata per leggere una tabella da un file di testo. A ogni riga del file corrisponde una riga della tabella. Numerosi altri parametri possono essere impostati. La tipica sintassi è

```
tab=read.table('province',sep=',',
  header=T)
```

Non useremo questa funzione per la nostra banca di dati, perché vogliamo creare le tabelle dal terminale o da un programma, salvarle con `save` e caricarle con `load` come descritto nel seguito.

save e load

Con

```
save(x,y,...,file='filexy')
```

si possono salvare gli oggetti `x,y,...` nel file *filexy* e nel *formato binario* speciale di R. Successivamente (anche in un'altra sessione) è sufficiente il comando `load('filexy')` affinché `x,y,...` siano di nuovo disponibili.

Esempio: Assumiamo che sia stata definita una tabella `tab` come in precedenza, che `A` sia una matrice ed `u,v` vettori di numeri. Allora li possiamo salvare file binario *salv* con

```
save(file='salv',tab,A,u,v)
```

Se adesso chiudiamo la sessione, quando riapriamo R possiamo ricaricare questi oggetti con `load('salv')`.

Creare una tabella vuota

Definiremo adesso e all'inizio del prossimo numero (che sarà dedicato alla possibilità di gestire una piccola banca dati con R) alcune funzioni per la nostra libreria nelle sezioni `DT` (funzioni generali per tabelle), `DTC` (funzioni per colonne), `DTR` (funzioni per righe). Per creare una tabella nuova vuota di cui sono indicati i nomi delle colonne nell'argomento `...` della funzione, useremo

```
Dt.nuova = function (...)
{a=c(...); m=length(a)
naxm=matrix(rep(NA,m),byrow=T,ncol=m)
tab=data.frame(naxm)
colnames(tab)=a; subset(tab,F)}
```

Esempio:

```
lazio=Dt.nuova('capoluogo','comuni',
  'sup','dens')
print(lazio)
# capoluogo comuni    sup       dens
# <0 rows> (or 0-length row.names)
```

I nomi delle colonne di una matrice o di una tabella si ottengono con `colnames`, i nomi delle righe con `rownames`. Queste funzioni possono essere usate anche alla sinistra di un assegnamento.

Aggiunta o sostituzione di righe

Creiamo una funzione per aggiungere una riga a una tabella esistente. Questa funzione a sua volta utilizza `rbind`. Allo scopo di riordinare le righe dopo l'aggiunta della nuova riga la funzione restituisce la tabella che si ottiene con `merge(tab,tab)`.

```
Dtr.agg = function (tab,a,b,...)
{if (!missing(b)) a=list(a,b,...)
names(a)=colnames(tab)
tab=rbind(tab,data.frame(a))
merge(tab,tab)}
# Per riordinare le righe.
```

Supponiamo di avere creato una tabella vuota `lazio` come nell'articolo precedente. Allora possiamo aggiungere righe nel seguente modo.

```
lazio=Dtr.agg(lazio,
  'Latina',33,2250,228)
print(lazio)
#   capoluogo comuni  sup dens
# 1    Latina     33 2250  228

lazio=Dtr.agg(lazio,
  'Viterbo',60,3612,81)
print(lazio)
#   capoluogo comuni  sup dens
# 1    Latina     33 2250  228
# 2   Viterbo     60 3612   81

lazio=Dtr.agg(lazio,
  'Roma',121,5000,719) # errore
lazio=Dtr.agg(lazio,
  'Rieti',73,2749,55)
lazio=Dtr.agg(lazio,
  'Frosinone',91,3244,152)
print(lazio)
#   capoluogo comuni  sup dens
# 1 Frosinone     91 3244  152
# 2    Latina     33 2250  228
# 3     Rieti     73 2749   55
# 4      Roma    121 5000  719
# 5   Viterbo     60 3612   81
```

Nell'inserimento della riga per *Roma* abbiamo però commesso un errore. Creiamo quindi una funzione `Dtr.sost` per sostituire una riga.

```
Dtr.sost = function (tab,n,a,b,...)
{if (!missing(b)) a=list(a,b,...)
tab=subset(tab,row.names(tab)!=n)
Dtr.agg(tab,a)}
```

Possiamo così sostituire la quarta riga con i valori corretti:

```
lazio=Dtr.sost(lazio,4,
  'Roma',121,5352,719)
print(lazio)
#   capoluogo comuni  sup dens
# 1 Frosinone     91 3244  152
# 2    Latina     33 2250  228
# 3     Rieti     73 2749   55
# 4      Roma    121 5352  719
# 5   Viterbo     60 3612   81
```

Per togliere righe da una tabella usiamo

```
Dtr.togli = function (tab,...)
{tab=subset(tab,
  !(row.names(tab) %in% c(...)))
merge(tab,tab)}
# Per riordinare le righe.
```

Selezione di righe

Per selezionare righe da una tabella usiamo le funzioni `Dtr` e `Dtr.cond`. Nella prima dobbiamo indicare i numeri (o più generalmente i nomi) delle righe richieste, nella seconda una condizione. Anche queste funzioni restituiscono nuove tabelle con le righe rinumerate.

```
Dtr = function (tab,...)
{tab=subset(tab,
  row.names(tab) %in% c(...))
merge(tab,tab)}
# Per riordinare le righe.

Dtr.cond = function (tab, cond)
{tab=subset(tab,cond)
merge(tab,tab)}
# Per riordinare le righe.
```

Esempi:

```
u=Dtr(lazio,1,5)
print(u)
#   capoluogo comuni  sup dens
# 1 Frosinone     91 3244  152
# 2   Viterbo     60 3612   81

u=Dtr.cond(lazio,lazio$sup>3000)
# Oppure Dtr(lazio,sup>3000)
# dopo attach(lazio).
print(u)
#   capoluogo comuni  sup dens
# 1 Frosinone     91 3244  152
# 2      Roma    121 5352  719
# 3   Viterbo     60 3612   81
```

Aggiunta di colonne

Per poter aggiungere nuove colonne a una tabella esistente creiamo una nostra funzione (molto semplice) che a sua volta utilizza `cbind`.

```
Dtc.agg = function (tab,...)
cbind(tab,data.frame(...))
```

Esempio:

```
lazio=Dtc.agg(lazio,
  sigla=c('FR','LT','RI','ROMA','VT'))
print(lazio)
#   capoluogo comuni  sup dens sigla
# 1 Frosinone     91 3244  152    FR
# 2    Latina     33 2250  228    LT
# 3     Rieti     73 2749   55    RI
# 4      Roma    121 5352  719  ROMA
# 5   Viterbo     60 3612   81    VT
```

Per togliere colonne da una tabella usiamo la funzione

```
Dtc.togli = function (tab,...)
{a=c(...)
if (is.character(a))
subset(tab,
  select=-match(a,colnames(tab)))
else subset(tab,select=-a)}
```

Per selezionare colonne da una tabella usiamo semplicemente

```
Dtc = function (tab,...)
subset(tab,select=c(...))
```

FONDAMENTI DI INFORMATICA

Corso di laurea in matematica | Anno accademico 2004/05 | Numero 10

Gestire una banca dati con R

Creeremo in questo numero la sezione DB della nostra libreria, che conterrà funzioni che ci permettono di gestire una piccola banca dati. Questa contiene tabelle salvate in formato binario con `save` (o con la nostra funzione `Db.salva` che utilizza `save` al suo interno) nella cartella a cui corrisponde la variabile `CARTELLADATI` definita nel file *alfa*. Una versione in formato testo della tabella viene salvata nella cartella `CARTELLAWRITE`, ma ciò è meno importante. La cartella `CARTELLACOMM` contiene, per alcune tabelle, informazioni (commenti) di carattere generale. Quando una tabella viene caricata, essa viene assegnata come valore alla variabile `TABELLA`. La tabella che corrisponde a questa variabile nel seguito sarà indicata con il termine *tabella di lavoro*.

Un elenco di tutte le tabelle contenute in una cartella `cart` si ottiene dal terminale con il comando `Dbe(cart)`. Con `Dbe()` si ottiene l'elenco di tutte le tabelle disponibili. La funzione `Dbe` è definita nel modo seguente:

```
Dbe = function (cartella)
{if (missing(cartella))
cartella=CARTELLADATI
else cartella=paste(CARTELLADATI,
  cartella,'/',sep='')
as.vector(dir(cartella,recursive=T),
  mode='character')}
```

Con `Dbe()` otteniamo

```
"economia/disoccupati-01"
"geografia/quindici-comuni"
"geografia/stati-africa"
"geografia/stati-america"
"geografia/stati-asia"
"geografia/stati-australia-isole"
"geografia/stati-europa"
```

con `Dbe("economia")` soltanto

```
"disoccupati-01"
```

Per sostituire la tabella di lavoro con un'altra, utilizziamo la funzione `Db.sostcon`. Essa cambia evidentemente il valore di `TABELLA`, ma anche quello di un'altra variabile globale `MODIFICA` che indica se ci sono state modifiche nella tabella di lavoro di cui tener conto per un eventuale salvataggio quando si carica una nuova tabella di lavoro o si termina la sessione. Per assegnare valori a una variabile globale usiamo la tecnica vista a pagina 3.

```
Db.sostcon = function (tab)
{assign('MODIFICA',T,pos=1)
assign('TABELLA',tab,pos=1); Db()}
```

Chiamiamo *modificante* una funzione che modifica la tabella di lavoro.

Il sistema di gestione di una banca dati così realizzato è certamente molto semplice, eppure, utilizzando le funzioni per tabelle che abbiamo imparato nel numero precedente, soprattutto `subset` e `merge`, possiamo già imitare molte operazioni di SQL in un ambiente di nostra creazione.

La sezione DBC

Della sezione DBC fanno parte le funzioni per le colonne della tabella di lavoro; esse corrispondono per la maggior parte alle funzione in DTC; alcune di esse modificano la tabella di lavoro, utilizzando `Db.sostcon`, assegnando in questo modo il valore `TRUE` alla variabile `MODIFICA` cosicché, quando opportuno, verrà chiesto il salvataggio della tabella. Le funzioni in DBC sono:

Dbc – Seleziona le colonne indicate della tabella di lavoro, utilizzando `Dtc`.

Dbc.a – Aggiunta di colonne. Modificante.

Dbc.cambianomi – Cambio dei nomi delle colonne. Modificante.

Dbc.n – Restituisce il numero delle colonne della tabella di lavoro di lavoro.

Dbc.selsost – Sostituisce la tabella di lavoro con la tabella che si ottiene selezionando le colonne indicate. Modificante.

Dbc.sost – Sostituisce una colonna. Modificante.

Dbc.togli – Tabella che si ottiene dalla tabella di lavoro togliendo le colonne indicate.

Dbc.toglisost – Sostituisce la tabella di lavoro con la tabella che si ottiene togliendo le colonne indicate. Modificante.

```
Dbc = function (...)
Dtc(TABELLA,...)

Dbc.a = function (...)
{tab=Dtc.agg(TABELLA,...)
Db.sostcon(tab)}

Dbc.cambianomi = function (...)
{tab=TABELLA; colnames(tab)=c(...)
Db.sostcon(tab)}

Dbc.n = function () ncol(TABELLA)

Dbc.selsost = function (...)
Db.sostcon(Dbc(...))

Dbc.sost = function (j,x)
{tab=Dtc.sost(Db.tab(),j,x)
Db.sostcon(tab)}

Dbc.togli = function (...)
Dtc.togli(TABELLA,...)

Dbc.toglisost = function (...)
Db.sostcon(Dbc.togli(...))
```

In questo numero

Sostituzione di colonne

Per sostituire una colonna in una tabella creiamo la funzione `Dtc.sost`, meno complicata di quanto sembri. Infatti dobbiamo usare `cbind` con parametri diversi per indicare se la colonna da sostituire si trova all'inizio, nel mezzo o alla fine della tabella.

```
Dtc.sost = function (tab,j,x)
{m=ncol(tab); a=matrix(x,ncol=1)
tabj=data.frame(a)
colnames(tabj)=colnames(tab)[j]
if (j==1) cbind(tabj,Dtc.sel(tab,2:m))
else if (j==m) cbind(Dtc.sel(tab,
  1:(m-1)),tabj)
else cbind(Dtc.sel(tab,1:(j-1)),tabj,
  Dtc.sel(tab,(j+1):m))}
```

Ordinamento di una tabella

Con `Dt.ord` otteniamo una tabella ordinata rispetto a una determinata colonna. Inizialmente è impostato l'ordine decrescente.

```
Dt.ord = function (tab,x,crescente=F)
{a=tab[,x]
tab=tab[order(a,
  decreasing=!crescente),]
merge(tab,tab)}
```

La matrice dei dati

Possiamo infine ottenere una matrice di dati da una tabella che, tranne le colonne indicate in `senza`, è omogenea, con la seguente funzione:

```
Dt.matrice = function (tab,senza=1)
{nomi=colnames(tab)
if (!identical(senza,0))
nomi=nomi[-senza]; a=tab[nomi]
v=unlist(a,use.names=F)
matrix(v,ncol=ncol(a))}
```

Esempio:

```
A=Dt.matrice(lazio,senza=c(1,5))
print(A)
#  91 3244 152
#  33 2250 228
#  73 2749  55
# 121 5352 719
#  60 3612  81
```

La sezione DBR

Definiamo nella sezione DBR le funzioni per le righe della tabella di lavoro.

Dbr – Seleziona le righe indicate della tabella di lavoro, utilizzando `Dtr`.

Dbr.a – Aggiunta di una riga rappresentata da una lista o da un argomento variabile. Modificante.

Dbr.cond – Seleziona le righe che soddisfano le condizioni indicate.

Dbr.n – Restituisce il numero delle righe.

Dbr.selsost – Sostituisce la tabella di lavoro con la tabella che si ottiene selezionando le righe indicate. Modificante.

Dbr.sost – Sostituisce una riga. Modificante.

Dbr.togli – Tabella che si ottiene togliendo le righe indicate.

Dbr.toglisost – Sostituisce la tabella di lavoro con la tabella che si ottiene togliendo le righe indicate. Modificante.

```
Dbr = function (...)
Dtr(TABELLA,...)

Dbr.a = function (a,b,...)
{if (!missing(b)) a=list(a,b,...)
Db.sostcon(Dtr.agg(TABELLA,a))}

Dbr.cond = function (cond)
Dtr.cond(TABELLA,cond)

Dbr.n = function () nrow(TABELLA)

Dbr.selsost = function (...)
Db.sostcon(Dbr(...))

Dbr.sost = function (n,...)
Db.sostcon(Dtr.sost(TABELLA,n,...))

Dbr.togli = function (...)
Dtr.togli(TABELLA,...)

Dbr.toglisost = function (...)
Db.sostcon(Dbr.togli(...))
```

Funzioni ausiliarie

Alcune funzioni ausliarie derivano direttamente da funzioni che già conosciamo.

Db.ord – Tabella di lavoro ordinata rispetto alla colonna indicata.

Db.matrice – Matrice numerica che si ottiene dalla tabella di lavoro tralasciando le colonne indicate. Utilizza `Dt.matrice`.

Db.tab – Restituisce la tabella di lavoro. È usata in assegnazioni della forma `tab=Db.tab()`, comunque equivalenti a `tab=TABELLA`.

Db..esiste – Indica se la variabile `TABELLA` è definita.

Db.comm – Visualizza il commento, se presente nella banca dati.

Db..attach – Esegue `attach`.

Db.cancella – Cancella la tab. di lavoro.

O.riga – Stampa una riga di separazione sullo schermo.

```
Db.ord = function (x,crescente=F)
Dt.ord(TABELLA,x,crescente=crescente)

Db.matrice = function (senza=1)
Dt.matrice(TABELLA,senza=senza)

Db.tab = function ()
{Db(); TABELLA}

Db..esiste = function ()
exists('TABELLA')

Db.comm = function ()
{O.riga(inizio='\n')
if (Db..esiste())
{comm=FILECOMM
if (File.esiste(comm))
cat(File.leggi(comm))
else cat('Commento non esiste.\n')}
else cat('Tabella non esiste.\n')}

Db..attach = function ()
T.eval('attach(TABELLA)')

Db.cancella = function ()
{assign('MODIFICA',F,pos=1)
if (Db..esiste()) T.eval('rm(TABELLA)')
else cat('Tabella non esiste.\n')}

O.riga = function (n=48,inizio='')
{a=rep('-',n)
a=paste(a,collapse='')
cat(inizio,a,'\n',sep='')}
```

Caricamento con Db

Per caricare una tabella presente in `CARTELLADATI` usiamo la funzione `Db` che a sua volta chiama `Db..carica`, una di quelle funzioni riconoscibili dall'infisso `..` che non sono destinate ad essere usate dall'utente ma solo all'interno di altre funzioni. In `Db..carica` è eseguita un'altra di queste funzioni, `Db..nome`, che definisce o ridefinisce alcune variabili che contengono i nomi dei files attualmente utilizzati: `FILEDATIBREVE`, il nome breve (cioè senza indicazione del cammino) del file binario; `FILEDATI`, il nome completo del file binario; `FILEWRITE`, il nome del file di testo; `FILECOMM`, il nome dell'eventuale file di commenti.

`Db` può essere chiamata con un nome breve oppure con un numero, ad esempio `Db(5)`. `O.interopositivo` accerta se è stato usato un numero. Se è già caricata una tabella, viene anche proposto il salvataggio con `Db.salva`, funzione che discuteremo per ultima. Si osservi l'utilizzo di `T.eval`.

```
Db = function (nome)
{if (!missing(nome)) Db..carica(nome)
else if (Db..esiste())
{Db..attach()
cat('\n',FILEDATIBREVE,'\n',sep='')
O.riga()
print(TABELLA)}
else cat('Tabella non esiste.\n')}

Db..carica = function (nome)
{e=Dbe()
if (O.interopositivo(nome)) nome=e[nome]
if (Db..esiste()) Db.salva()
Db..nome(nome)
if (FILEDATIBREVE %in% e)
{comando=paste('load("',FILEDATI,'")',
  sep='')
T.eval(comando)
Db..attach(); Db()}
else cat('File non esiste.\n')}

O.interopositivo = function (x)
{if (mode(x)!='numeric') F
else if (x!=floor(x)) F
else (x>0)}

Db..nome = function (nome)
{assign('FILEDATIBREVE',nome,pos=1)
dati=paste(CARTELLADATI,nome,sep='')
assign('FILEDATI',dati,pos=1)
write=paste(CARTELLAWRITE,nome,sep='')
assign('FILEWRITE',write,pos=1)
comm=paste(CARTELLACOMM,nome,sep='')
assign('FILECOMM',comm,pos=1)}

T.eval = function (x)
{com=textConnection(x)
source(com); close(com)}
```

Db.nuova e Db.salva

Per la creazione di una nuova tabella da utilizzare come tabella di lavoro usiamo la funzione `Db.nuova` il cui primo argomento è il nome che verrà assegnato alla tabella nella banca dati. Se una tabella con questo nome è già caricata, la funzione visualizza soltanto un messaggio; altrimenti, se la banca dati contiene già un file dallo stesso nome, viene semplicemente caricata quella tabella come si vede nelle istruzioni che terminano in `#`.

Se è caricata una tabella, viene prima chiesto il salvataggio. Negli argomenti `...` sono indicati i nomi delle colonne; tuttavia si può indicare nell'argomento `tab` un'altra tabella che sostituisce la tabella di lavoro, a cui viene assegnato il nuovo nome nella banca dati (l'inserimento nella banca dati avviene soltanto con il primo salvataggio).

```
Db.nuova = function (nome,...,tab)
{if (Db..stessatabella(nome))
cat('Stessa tabella.\n')
else {if (Db..esiste())
Db.salva (messaggio=
  'Salvare la vecchia tabella? (s/n): ')
Db..nome(nome)                    #
if (FILEDATIBREVE %in% Dbe())     #
Db(nome)                          #
else {if (!missing(tab)) tab=tab
else tab=Dt.nuova(...)
Db.sostcon(tab)}}}

Db..stessatabella = function (nome)
{if (!Db..esiste()) return(F)
(nome==FILEDATIBREVE)}
```

Ogni volta che si cerca di caricare una nuova tabella e quando si vuole uscire da R, viene eseguita la funzione `Db.salva` che controlla se è caricata una tabella di lavoro e se sono state effettuate modifiche non salvate su disco. In caso affermativo viene chiesto il salvataggio.

```
Db.salva = function (messaggio=
  'Vuoi salvare la tabella? (s/n): ')
{if (Db..esiste())
{if (!MODIFICA)
cat('Tabella non modificata.\n\n')
else {Db(); cat('\n'); repeat
{s=readline(messaggio)
if (s=='s') {s=T; break}
if (s=='n') {s=F; break}}
if (s) {comando=paste('save(TABELLA,
  file="',FILEDATI,'")',sep='')
T.eval(comando)
comando=paste('write.table(TABELLA,
  file="',FILEWRITE,'")',sep='')
T.eval(comando)
assign('MODIFICA',F,pos=1)
cat('Tabella salvata.\n')}
else cat('Tabella non salvata.\n')}}
else cat('Tabella non esiste.\n')}
```