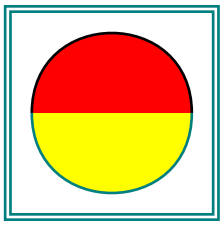
ALGORITMI E STRUTTURE DI DATI

Corso di laurea in matematica | Anno accademico 2005/06 | Numero 1

PDF

La sigla PDF è un'abbreviazione per *Portable Document Format*, un formato per documenti portatili nelle vie di comunicazione telematiche. Questo formato conserva ancora tracce delle sue origini da PostScript, come adesso vedremo; non può essere più considerato un linguaggio di programmazione, ma piuttosto un sofisticato linguaggio per la descrizione di documenti strutturati con il modello grafico di PostScript incorporato in buona parte. Non ci occuperemo molto di PDF, ma vogliamo almeno vedere la struttura tipica di un file PDF in un esempio semplice, quasi minimale. Nella realtà i file PDF sono sempre molto più complicati (ciò non vale invece necessariamente per i file PostScript, come scopriremo forse a sorpresa).

```
%PDF-1.4

1 0 obj
<< /Type /Catalog
/Pages 2 0 R >>
endobj

2 0 obj
<< /Type /Pages
/Kids [3 0 R] /Count 1 >>
endobj

3 0 obj
<< /Type /Page
/Parent 2 0 R
/MediaBox [0 0 100 100]
/Contents 4 0 R >>
endobj

4 0 obj
<< /Length 235 >>
stream
0 0.5 0.5 RG
10 10 m 90 10 l 90 90 l 10 90 l 10 10 l s
12 12 m 88 12 l 88 88 l 12 88 l 12 12 l s
1 0 0 rg
20 50 m
20 90 80 90 80 50 c f
20 50 m
20 90 80 90 80 50 c s
1 1 0 rg
20 50 m
20 10 80 10 80 50 c f
20 50 m
20 10 80 10 80 50 c s
endstream
endobj

xref
0 5
0000000000 65535 f
0000000010 00000 n
0000000061 00000 n
0000000119 00000 n
0000000207 00000 n
trailer
<< /Size 5 /Root 1 0 R >>
startxref
493
%%EOF
```



La prima riga identifica la versione di PDF utilizzata; il comando `%%EOF` alla fine indica la fine del file; alcuni programmi richiedono che esso non sia seguito da una nuova riga o altri caratteri.

Vediamo poi quattro blocchi compresi tra `i 0 obj` ed `endobj`. Questi blocchi sono gli *oggetti* (in inglese *objects*) di cui si compone il file. Il blocco che inizia con `xref` e termina con `493` è di per sé soltanto ausiliario; contiene il numero degli oggetti e, in un formato stabilito, le posizioni degli oggetti (il numero dei bytes dall'inizio del file all'inizio dell'oggetto) ed è piuttosto delicato perché ogni volta che togliamo o inseriamo un carattere le posizioni cambiano. Questa tabella in inglese è detta *cross-reference table* e la `x` in `xref` vuole indicare una crocetta (*cross*).

Torniamo alle righe della forma `i 0 obj` che indicano l'inizio dell'`i`-esimo oggetto del file. Nei primi tre oggetti troviamo espressioni della forma `j 0 R`; qui la `R` sta per riferimento, infatti ad esempio `2 0 R` è un riferimento al secondo oggetto che in qualche modo deve essere inserito in quel punto.

In questo numero

Quindi nel nostro caso il primo oggetto è il catalogo del file che rimanda al secondo oggetto che corrisponde all'albero delle pagine che però stavolta consiste di una pagina sola la cui struttura è descritta nel terzo oggetto che a sua volta rimanda per il contenuto al quarto oggetto. Quest'ultimo contiene, nella parte tra `stream` e `endstream`, le istruzioni di disegno che vediamo nella visualizzazione del file. La lunghezza (`/Length`) non è altro che la lunghezza complessiva in bytes di queste istruzioni.

Le istruzioni di disegno sono simili a quelle del PostScript, ma abbreviate nella notazione e molto meno complete. Raffrontiamo i comandi usati nell'esempio e i comandi corrispondenti di PostScript:

```
m   moveto
l   lineto
s   stroke
RG  setrgbcolor (tracciato)    rg  setrgbcolor (riempimenti)
c   curveto
f   fill
```

Le istruzioni usate per la figura possono essere così raggruppate:

```
Colore di disegno ciano scuro.
Due quadrati concentrici.
Colore di riempimento rosso.
Semicerchio rosso mediante una curva di Bezier.
Bordo del semicerchio usando s al posto di f.
Colore di riempimento giallo.
Semicerchio inferiore giallo.
Bordo del semicerchio inferiore.
```

Esercizio: Individuare le istruzioni che corrispondono a queste operazioni grafiche.

Traduzione in PostScript

Per far apparire il disegno a pagina 1 abbiamo inserito queste istruzioni nella sorgente Latex degli appunti:

```
\special {"
120 10 translate
0 0.5 0.5 setrgbcolor
10 10 moveto 90 10 lineto 90 90 lineto
  10 90 lineto 10 10 lineto stroke
12 12 moveto 88 12 lineto 88 88 lineto
  12 88 lineto 12 12 lineto stroke
1 0 0 setrgbcolor 20 50 moveto
20 90 80 90 80 50 curveto fill
0 0 0 setrgbcolor 20 50 moveto
20 90 80 90 80 50 curveto stroke
1 1 0 setrgbcolor 20 50 moveto
20 10 80 10 80 50 curveto fill
0 0.5 0.5 setrgbcolor 20 50 moveto
20 10 80 10 80 50 curveto stroke}
```

Il comando `\special` (si noti la sintassi con la virgoletta singola) contiene le stesse istruzioni che abbiamo usato nel PDF, tenendo conto della tabella per le corrispondenze tra istruzioni PDF e istruzioni PostScript con la sola aggiunta di `120 10 translate` per posizionare il disegno sulla pagina.

Lavorare con PDF

Difficilmente si lavora a mano con PDF e si impiegano invece i numerosi strumenti disponibili, tra cui segnaliamo *pdftk*, un programma che permette tra l'altro di modificare, unire o correggere file PDF. Molte indicazioni si trovano nel libro di Steward e nel sito web (*AccessPDF*) dello stesso autore, oppure, in tedesco, nel libro di Merz/Drümmer. Una presentazione completa delle specifiche PDF è contenuta nel manuale della Adobe (Geschke/Warnock).

C. Geschke/J. Warnock (ed.): PDF reference. Addison-Wesley 2001.
T. Merz/O. Drümmer: Die PostScript- und PDF-Bibel. PDFlib 2002.
S. Steward: PDF hacks. O'Reilly 2004.

ImageMagick e GQview

ImageMagick è un programma di elaborazione delle immagini che si presenta come collezione di vari strumenti di cui in lezione useremo soprattutto *convert* per la conversione tra formati di immagini, in particolare da PostScript (o PDF) a *.png* (i formati supportati sono però veramente molti).

Le istruzioni di cui avremo bisogno saranno soprattutto

```
convert imm.ps imm.png
convert -transparent white imm.ps imm.png
```

L'opzione `-transparent white` fa in modo che il bianco (che tipicamente è il colore di fondo) diventi trasparente; ciò è utile quando l'immagine deve essere inserita su uno sfondo di altro colore.

convert può essere usato anche per convertire più immagini in una sola, ad esempio per creare un'immagine animata in formato *.gif*, come vedremo.

La pagina web di ImageMagick contiene una guida alle molte opzioni di *convert*.

M. Still: The definitive guide to ImageMagick. Apress 2005.

GQview è un bellissimo programma per la visualizzazione di immagini, adatto soprattutto per vedere in fila tutte le immagini di una cartella. Cliccando con il tasto sinistro del mouse sull'immagine visualizzata, fa vedere la prossima; si torna all'immagine precedente invece con il tasto medio. Per ogni immagine si possono (sotto Linux) invocare direttamente *xv*, *xpaint* o *gimp*.

Si consiglia quindi di raccogliere le immagini che creiamo con i nostri programmi in un'apposita cartella per poterle vedere insieme con GQview.

Forth

Il *Forth* venne inventato all'inizio degli anni '60 da Charles Moore per piccoli compiti industriali, ad esempio il pilotaggio di un osservatorio astronomico. È allo stesso tempo un linguaggio semplicissimo e estremamente estendibile - dipende solo dalla pazienza del programmatore quanto voglia accrescere la biblioteca delle sue funzioni (o meglio macroistruzioni). Viene usato nel controllo automatico, nella programmazione di sistemi, nell'intelligenza artificiale. L'interprete di Forth è basato su una *macchina virtuale* (piuttosto simile concettualmente a quelle usata per Java) e quindi facilmente portatile. Qualche passo dal libro di Philip Koopman:

"One of the characteristics of Forth is its very high use of subroutine calls. This promotes an unprecedented level of modularity, with approximately 10 instructions per procedure being the norm. Tied with this high degree of modularity is the interactive development environment used by Forth compilers ...

This interactive development of modular programs is widely claimed by experienced Forth programmers to result in a factor of 10 improvement in programmer productivity, with improved software quality and reduced maintenance costs ... Forth programs are usually quite small ...

Good programmers become exceptional when programming in Forth. Excellent programmers can become phenomenal. Mediocre programmers generate code that works, and bad programmers go back to programming in other languages. Forth ... is different enough from other programming languages that bad habits must be unlearned ... Once these new skills are acquired, though, it is a common experience to have Forth-based problem solving skills involving modularization and partitioning of programs actually improve a programmers effectiveness in other languages as well."

L. Brodie: Starting Forth. Prentice Hall 1981.
L. Brodie: Thinking Forth. Forth Interest Group 1994.
E. Conklin/E. Rather: Forth programmer's handbook. Mass Market 1998.
P. Koopman: Stack computers. Ellis Horwood 1989.

PostScript

Uno stretto parente e discendente del Forth è il *PostScript*, un sofisticato linguaggio per stampanti che mediante un interprete (il più diffuso è *ghostscript*) può essere utilizzato anche come linguaggio di programmazione per altri scopi.

Forth e PostScript presentano alcune caratteristiche che li distinguono da altri linguaggi di programmazione:

(1) Utilizzano la notazione polacca inversa (RPN, reverse Polish notation) come alcune calcolatrici tascabili (della Hewlett Packard per esempio); ciò significa che gli argomenti precedono gli operatori. Invece di `a+b` si scrive ad esempio `a b +` (in PostScript `a b add`) e quindi `(a+3)*5+1` diventa `a 3 add 5 mul 1 add`. Ciò comporta una notevole velocità di esecuzione perché i valori vengono semplicemente prelevati da uno *stack* e quindi, benché interpretati, Forth e PostScript sono linguaggi veloci con codice sorgente molto breve.

(2) Entrambi i linguaggi permettono e favoriscono un uso estensivo di macroistruzioni (abbreviazioni) che nel PostScript possono essere addirittura organizzate su più dizionari (fornendo così una via alla programmazione orientata agli oggetti in questi linguaggi apparentemente quasi primitivi). Tranne pochi simboli speciali quasi tutte le lettere possono far parte dei nomi degli identificatori, quindi se ad esempio anche in PostScript volessimo usare `+` e `*` al posto di `add` e `mul` basta definire

```
/+ {add} def
/* {mul} def
```

(3) In pratica non esiste distinzione tra procedure e dati; tutti gli oggetti sono definiti mediante abbreviazioni e la programmazione acquisisce un carattere fortemente logico-semantico.

J. Warnock/C. Geschke (ed.): PostScript language reference. Addison-Wesley 1999. John Warnock e Charles Geschke, laureati in matematica, sono gli inventori di PostScript. Nel 1982 hanno fondato Adobe.

www.adobe.com/. Sito di Adobe, con manuali e guide alla programmazione in PostScript.

www.cs.indiana.edu/docproject/programming/postscript/postscript.html. Una prima introduzione, di P. Weingartner.

Lo stack

Una *pila* (in inglese *stack*) è una delle più elementari e più importanti strutture di dati. Uno stack è una successione di dati in cui tutte le inserzioni, cancellazioni ed accessi avvengono a una sola estremità. Gli interpreti e compilatori di tutti i linguaggi di programmazione utilizzano uno o più stack per organizzare le chiamate annidate di funzioni; in questi casi lo stack contiene soprattutto gli indirizzi di ritorno, i valori di parametri che dopo un ritorno devono essere ripristinati, le variabili locali di una funzione.

Descriviamo brevemente l'esecuzione di un programma in PostScript (o Forth). Consideriamo ancora la sequenza

```
40 3 add 5 mul
```

Assumiamo che l'ultimo elemento dello stack degli operandi sia `x`. L'interprete incontra prima il numero `40` e lo mette sullo stack degli operandi, poi legge `3` e pone anche questo numero sullo stack (degli operandi, quando non specificato altrimenti). In questo momento il contenuto dello stack è `... x 40 3`. Successivamente l'interprete incontra l'operatore `add` che richiede due argomenti che l'interprete preleva dallo stack; adesso viene calcolata la somma 40+3=43 e posta sullo stack i cui ultimi elementi sono così `x 43`. L'interprete va avanti e trova `5` e lo pone sullo stack che contiene così `... x 43 5`. Poi trova di nuovo un operatore (`mul`), preleva i due argomenti necessari dallo stack su cui ripone il prodotto (43·5=215). Il contenuto dello stack degli operandi adesso è `... x 215`.

Abbreviazioni con def

Le abbreviazioni vengono definite secondo la sintassi

```
/abbreviazione significato def
```

Se il significato è un operatore eseguibile bisogna racchiudere le operazioni tra parentesi graffe come abbiamo fatto a pagina 2 per `add` e `mul`, per impedire che vengano eseguite già la prima volta che l'interprete le incontra, cioè nel momento in cui legge l'abbreviazione. In questo modo mediante abbreviazioni si possono anche definire *macroistruzioni* che in PostScript svolgono il ruolo delle funzioni definibili in altri linguaggi.

Quando il significato invece non è eseguibile, non bisogna mettere parentesi graffe, ad esempio

```
/e 2.7182818 def
/pi 3.14159 def
```

Operatori matematici

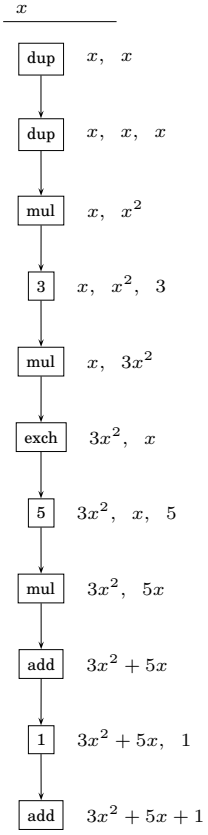
$x\ y$	`add`	$x + y$
$x\ y$	`sub`	$x - y$
x	`neg`	$-x$
$x\ y$	`mul`	xy
$x\ y$	`div`	x/y
$x\ y$	`idiv`	$[x/y]$ (quoziente intero) per $y > 0$
$x\ y$	`mod`	resto di x modulo y per $y > 0$
x	`abs`	$\lvert x \rvert$ (valore assoluto)
x	`floor`	$[x]$ (parte intera)
x	`ceiling`	intero più vicino a destra
x	`round`	arrotondamento di x (*)
x	`sqrt`	$\sqrt{x}$
x	`sin`	$\sin x$
x	`cos`	$\cos x$
x	`exp`	e^x
x	`log`	$\log_{10} x$ (logaritmo in base 10)
x	`ln`	$\log x$ (logaritmo naturale)
$y\ x$	`atan`	angolo del punto (x, y) (**)

(*) Per $x \in \mathbb{Z} + 0.5$ l'arrotondamento avviene verso destra.

(**) `y x atan` è, per un punto (x, y) diverso dall origine $(0, 0)$, il suo angolo α in coordinate polari e misurato in gradi, preso con $0 \le \alpha < 360°$. Attenzione che negli argomenti viene prima la `y`!

Diagrammi di flusso per lo stack

Vogliamo scrivere una macroistruzione per la funzione f definita da $f(x) = 3x^2 + 5x + 1$. Per tale compito in PostScript (e in Forth) si possono usare diagrammi di flusso per lo stack, dove indichiamo ogni volta gli elementi più a destra dello stack rimasti dopo che l'istruzione è stata eseguita. Utilizziamo due istruzioni `dup`, che duplicano l'ultimo elemento dello stack (vedremo subito perché), ed il comando `exch`, che scambia gli elementi più a destra dello stack. All'inizio l'ultimo elemento dello stack è x.



Il diagramma di flusso corrisponde alla macroistruzione

```
/f {dup dup mul 3 mul exch 5 mul add 1 add} def
```

Spesso scriveremo i diagrammi di flusso semplicemente riga per riga senza le caselle e usando virgole solo se necessario.

Operatori per lo stack

Seguendo la notazione del manuale di Adobe, indichiamo, quando è appropriato, gli operatori nella forma α `op` β, dove α è la parte considerata dello stack prima dell'esecuzione dell'operatore `op`, mentre β descrive lo stato dopo l'esecuzione, eventualmente seguito da un commento. `clear` vuota tutto lo stack.

x	`pop`	- (toglie l'ultimo elemento dello stack)
x y	`exch`	y x
x	`dup`	x x
x_1 ... x_n n	`copy`	x_1 ... x_n x_1 ... x_n
a 0	`index`	a a
a b 1	`index`	a b a
a b c 2	`index`	a b c a (ecc.)
a b c 3 1	`roll`	c a b
a b c d 4 1	`roll`	d a b c
a b c d e 5 2	`roll`	d e a b c
a b c 3 -1	`roll`	b c a (ecc.)

Visualizzazione di stringhe

Stringhe in PostScript sono racchiuse tra parentesi tonde e vengono visualizzate con il comando `show`. Prima di poter utilizzare questo comando è però necessario che sia definito il punto in cui la stringa deve iniziare e che sia impostato un insieme di caratteri (in inglese *font*).

Il punto iniziale può essere definito direttamente mediante un'istruzione `moveto` oppure indirettamente mediante una precedente operazione di disegno.

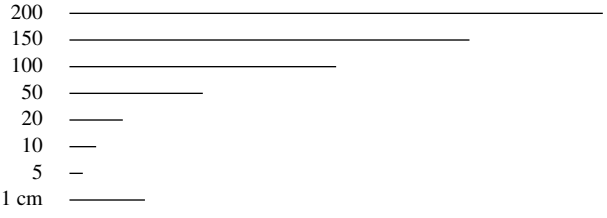
Dagli esempi che seguono si deduce il modo in cui devono essere impostati gli insiemi di caratteri:

```
/normale /Times-Roman findfont 12 scalefont def
/normale8 /Times-Roman findfont 8 scalefont def
/grassetto /Times-Bold findfont 12 scalefont def
/corsivo /Times-Italic findfont 12 scalefont def
/grassettocorsivo /Times-BoldItalic findfont 12 scalefont def
/helveticanormale /Helvetica findfont 12 scalefont def
/couriergrassetto /Courier-Bold findfont 10 scalefont def
/symbol /Symbol findfont 10 scalefont def

normale setfont 40 84 moveto (scritta normale (Times 12)) show
normale8 setfont 40 72 moveto (scritta normale (Times 8)) show
grassetto setfont 40 60 moveto (grassetto (Times 12)) show
corsivo setfont 40 48 moveto (corsivo (Times 12)) show
grassettocorsivo setfont 40 36 moveto
  (grassetto corsivo (Times 12)) show
helveticanormale setfont 40 24 moveto (Helvetica 12) show
couriergrassetto setfont 40 12 moveto
(Courier grassetto 10) show
symbol setfont 40 0 moveto (Sumbolikh 10) show
```

scritta normale (Times 12)
scritta normale (Times 8)
grassetto (Times 12)
corsivo (Times 12)
grassetto corsivo (Times 12)
Helvetica 12
Courier grassetto 10
Συμβολικη 10

Coordinate

Le coordinate in PostScript vengono espresse in *punti*; un punto è un 72-esimo di pollice. Siccome un pollice è uguale a 2.54 cm, 100 punti corrispondono a 3.53 cm. D'altra parte un pollice è uguale a 72 punti e quindi un centimetro corrisponde a 28.3 punti. Esempi:



Il disegno è stato ottenuto con

```
%%BoundingBox: 0 0 500 200
2 2 scale
/normale /Times-Roman findfont 8 scalefont def
normale setfont 0.5 setlinewidth
/h 30 def /v 80 def /a 10 string def
10 v moveto [200 150 100 50 20 10 5]
{dup a cvs 20 1 index stringwidth pop sub v moveto show
h v 2 add moveto 0 rlineto stroke
/v v 10 sub def} forall
20 (1 cm) stringwidth pop sub v moveto (1 cm) show
h v 2 add moveto 28.3 0 rlineto stroke
```

Se `a` è una stringa, `a stringwidth` mette sullo stack una coppia di numeri, di cui il primo è la larghezza della stringa, il secondo l'altezza. Nell'esempio quest'ultima non serve e la eliminiamo con `pop`. Spiegheremo altri dettagli fra poco.

scale e translate

Con il comando `scale` si possono ridefinire le unità di misura (indipendentemente per le coordinate x ed y, anche se quasi sempre si userà la stessa scala per entrambe). Siccome un millimetro corrisponde a 2.83 punti, se all'inizio di un programma di disegno poniamo

```
2.83 2.83 scale
```

possiamo lavorare con millimetri come unità di misura.

Il comando `translate`, che abbiamo già incontrato a pagina 2, permette di spostare l'origine delle coordinate cartesiane e quindi, in particolare, di spostare fisicamente una parte del disegno.

Il comando cvs

Per visualizzare con `show` un oggetto `x` che non sia una stringa, ad esempio un numero, bisogna trasformare `x` in una stringa utilizzando l'operatore `cvs` (abbreviazione per *convert to string*), che viene usato nel formato

```
x s cvs
```

dove `s` è una stringa la cui lunghezza è sufficiente per contenere i caratteri necessari per rappresentare `x`. `s` può essere una stringa già esistente oppure definita appositamente con il comando `n string` che crea una stringa di caratteri con codice ASCII 0 di lunghezza n. Così nel programma per il disegno nella prima colonna di questa pagina abbiamo prima creato con `/a 10 string def` la stringa `a` che successivamente è stata utilizzata nei comandi di conversione `a cvs`.

Se l'oggetto `x` stesso è una stringa, con `x s cvs` si ottiene una copia *indipendente* di `x`, cioè una copia `y` che non risente delle modifiche che successivamente eseguiremo sull'originale `x`, mentre con `/y x def` `x` e /verb/y/ sono semplicemente nomi diversi per la stessa stringa. Esempio:

```
%%BoundingBox: 0 0 200 200
2 2 scale
/normale /Times-Roman findfont 10 scalefont def
normale setfont /s 20 string def
/x (Roma) def /y x def /z x s cvs def
x 2 115 put
10 72 moveto (x = ) show x show
10 60 moveto (y = ) show y show
10 48 moveto (z = ) show z show
/x (Ferrara) def
10 36 moveto (x = ) show x show
10 24 moveto (y = ) show y show
10 12 moveto (z = ) show z show
```

x = Rosa
y = Rosa
z = Roma
x = Ferrara
y = Rosa
z = Roma

Come si vede, la copia indipendente `z` (creata con `cvs`) non è mai stata influenzata dai cambiamenti di `x`, mentre `y`, che corrisponde alla stessa locazione in memoria di `x`, viene modificata se operiamo in quella porzione di memoria (ad esempio con `x 2 115 put`, comando con cui si pone il terzo carattere di `x` uguale al carattere con codice ASCII 115, cioè ad *s*). `y` non è invece modificato se con `x (Ferrara) def` alla variabile `x` viene assegnato un nuovo significato in un nuovo indirizzo.

„Please do not call PostScript a page description language ... PostScript from Adobe Systems is an underappreciated yet superb general purpose computing language.“ (Don Lancaster)

D. Lancaster: PostScript insider secrets. Byte May 1990, 381-389.

www.tinaja.com/post01.asp. Saggi su PostScript, di Don Lancaster, riguardanti sia la programmazione che le applicazioni tipografiche.

Operatori per stringhe

`a` e `b` sono stringhe:

n	`string`	stringa nulla di lunghezza n
`a`	`length`	lunghezza di `a`
`a` i	`get`	i-esimo elemento (un intero!) di `a`
`a` i `x`	`put`	pone l'i-esimo elemento di `a` uguale ad `x`
`a` i n	`getinterval`	stringa che consiste degli n elementi che si trovano in `a` a partire dall'indice i
`a` i `b`	`putinterval`	sostituisce, a partire dall'indice i, una parte di `a` con `b`
`a f`	`forall`	esegue l'operazione `f` su tutti gli elementi di `a`

Alcune di queste operazioni sono identiche alle omonime operazioni per vettori. Esistono anche funzioni di ricerca per cui si rimanda al manuale di Adobe, se non verranno trattate nel corso.

Stringhe sono molto simili a vettori di bytes, cioè di interi compresi tra 0 e 255. Per questa ragione i caratteri di una stringa, quando vengono estratti, compaiono come interi:

```
/normale /Times-Roman findfont 9 scalefont def
normale setfont
/s 3 string def 10 12 moveto
(ABC abc) {s cvs show ( ) show} forall
/s 1 string def 10 2 moveto
(ABC abc) {s dup 0 3 index put show pop} forall
```

65 66 67 32 97 98 99
ABC abc

Analizziamo, per ogni singolo passo, l'operazione nell'ultima riga, in cui i numeri x estratti vengono riconvertiti in stringhe s_x a un carattere visualizzate con `show`.

	x
`s`	$x\ s$
`dup`	$x\ s\ s$
`0`	$x\ s\ s\ 0$
`3 index`	$x\ s\ s\ 0\ x$
`n put`	$x\ s_x$
`show`	x
`pop`	

Si noti che `a i x put` toglie la stringa trasformata a_x dallo stack e quindi dobbiamo raddoppiare la stringa con `dup`.

Vettori

I componenti di un vettore possono essere di tipo diverso e vengono numerati a partire da sinistra cominciando con 0 (mentre gli elementi dello stack sono numerati a partire da destra). Per elencare gli elementi di un vettore si usano le parentesi quadre - ma ciò, come vedremo, non è una semplice notazione! Gli elementi sono separati da spazi, ad esempio

```
/v [1 2 3 (abc) 4 [5 6]] def
```

Il vettore `v` ha 6 componenti, di cui l'ultimo è sua volta una vettore.

Alcuni degli operatori per vettori a e b:

n	`array`	vettore di oggetti nulli di lunghezza n
`a`	`length`	lunghezza di `a`
`a` i	`get`	i-esimo elemento di `a`
`a` i `x`	`put`	pone l'i-esimo elemento di `a` uguale ad `x`
`a` i n	`getinterval`	vettore che consiste degli n elementi che si trovano in `a` a partire dall'indice i
`a` i `b`	`putinterval`	sostituisce, a partire dall'indice i, una parte di `a` con `b`
`a f`	`forall`	esegue l'operazione `f` su tutti gli elementi di `a`
`a`	`aload`	pone tutti gli elementi di `a` sullo stack e aggiunge come ultimo elemento dello stack `a` stesso
x_1 ... x_n `a`	`astore`	carica gli elementi x_j nel vettore `a` (la cui lunghezza è n) e pone `a` (così trasformato) sullo stack

Mentre i caratteri di una stringa inizializzata con `string` sono tutti uguali all'intero 0, gli elementi di un vettore creato con `array` sono tutti uguali all'elemento `null`, che è diverso dal numero 0.

Concatenazione di stringhe e vettori

Definiamo una procedura `Str.concat` per la concatenazione di stringhe seguendo un diagramma di flusso in cui s, dopo il comando `string` della nona riga è una stringa di zeri di lunghezza $|a| + |b|$, nella quale prima viene inserito b a partire dall'indice $|a|$, poi a a partire dall'indice 0. Con $|a|$ denotiamo la lunghezza di una stringa o di un vettore a.

	$a\ b$
`dup`	$a\ b\ b$
`length`	$a\ b\ \|b\|$
`2 index`	$a\ b\ \|b\|\ a$
`length`	$a\ b\ \|b\|\ \|a\|$
`dup`	$a\ b\ \|b\|\ \|a\|\ \|a\|$
`5 1 roll`	$\|a\|\ a\ b\ \|b\|\ \|a\|$
`add`	$\|a\|\ a\ b,\ \|b\| + \|a\|$
`string`	$\|a\|\ a\ b\ s$
`dup dup`	$\|a\|\ a\ b\ s\ s\ s$
`6 -1 roll`	$a\ b\ s\ s\ s\ \|a\|$
`5 -1 roll`	$a\ s\ s\ s\ \|a\|\ b$
`putinterval`	$a\ s\ s$
`0`	$a\ s\ s\ 0$
`4 -1 roll`	$s\ s\ 0\ a$
`putinterval`	s

Otteniamo così la procedura

```
/Str.concat {dup length 2 index length dup 5 1 roll
  add string dup dup 6 -1 roll 5 -1 roll putinterval
  0 4 -1 roll putinterval} def
```

Lo stesso algoritmo lo possiamo usare per vettori, sostituendo `string` con `array`:

```
/Vett.concat {dup length 2 index length dup 5 1 roll
  add array dup dup 6 -1 roll 5 -1 roll putinterval
  0 4 -1 roll putinterval} def
```

Esempio:

```
/normale /Times-Roman findfont 8 scalefont def
normale setfont
10 10 moveto (alfa) (beta) Str.concat show
10 0 moveto
[1 2 3] [4 5 6 7 8 9 (a) (b)] Vett.concat
[(c) 0 1 2 5 add] Vett.concat {10 string cvs show} forall
```

con output:

alfabeta
123456789abc017

Vedremo più avanti che `Vett.concat` e `Str.concat` possono essere definite senza l'utilizzo delle funzioni `putinterval` e `length`.

repeat

L'istruzione `n f repeat` fa in modo che `f` (spesso un'espressione tra parentesi graffe) venga eseguita n volte. Utilizzato in modo appropriato e sfruttando il meccanismo delle operazioni sullo stack, questo semplice comando è sorprendentemente potente. Esempio:

```
/normale /Times-Roman findfont 10 scalefont def
normale setfont 10 0 moveto
(Roma) ( ) (Firenze) ( ) (Ferrara) ( ) (Padova)
7 {show} repeat
```

con output

Padova Ferrara Firenze Roma

Numeri esadecimali

Nei linguaggi macchina e assembler (ma anche nella rappresentazione RGB di colori) molto spesso si usano i numeri esadecimali o, più correttamente, la rappresentazione esadecimale dei numeri naturali, cioè la loro rappresentazione in base 16.

Per $2789 = 10 \cdot 16^2 + 14 \cdot 16 + 5 \cdot 1$ potremmo ad esempio scrivere

`2789 = (10,14,5)`$_{16}$.

In questo senso `10`, `14` e `5` sono le cifre della rappresentazione esadecimale di `2789`. Per poter usare lettere singole per le cifre si indicano le cifre `10`,...,`15` mancanti nel sistema decimale nel modo seguente:

```
10   A
11   B
12   C
13   D
14   E
15   F
```

In questo modo adesso possiamo scrivere $2789 = (AE5)_{16}$. In genere si possono usare anche indifferentemente le corrispondenti lettere minuscole. Si noti che `(F)`$_{16}$ `= 15` assume nel sistema esadecimale lo stesso ruolo come il `9` nel sistema decimale.

Quindi `(FF)`$_{16}$ `=` $255 = 16^2 - 1 = 2^8 - 1$. Un numero naturale n con $0 \le n \le 255$ si chiama un *byte*, un *bit* è invece uguale a 0 o a 1. Esempi:

	0	$(0)_{16}$
	14	$(E)_{16}$
	15	$(F)_{16}$
	16	$(10)_{16}$
	28	$(1C)_{16}$
2^5	32	$(20)_{16}$
2^6	64	$(40)_{16}$
	65	$(41)_{16}$
	97	$(61)_{16}$
	127	$(7F)_{16}$
2^7	128	$(80)_{16}$
	203	$(CB)_{16}$
	244	$(F4)_{16}$
	255	$(FF)_{16}$
2^8	256	$(100)_{16}$
2^{10}	1024	$(400)_{16}$
2^{12}	4096	$(1000)_{16}$
	65535	$(FFFF)_{16}$
2^{16}	65536	$(10000)_{16}$

Si vede da questa tabella che i byte sono esattamente quei numeri per i quali sono sufficienti al massimo due cifre esadecimali. Nell'immissione di una successione di numeri esadecimali come un'unica stringa spesso si pone uno zero all'inizio di quei numeri (da `0` a `9`) che richiedono una cifra sola, ad esempio la stringa `0532A2014E586A750EAA` può essere usata per rappresentare la successione `(`5,32,A2,1,4E,58,6A,75,E,AA`)` di numeri esadecimali.

Lo schema di Horner

Sia dato un polinomio

$$f = a_0 x^n + a_1 x^{n-1} + \ldots + a_n$$

con coefficienti reali $a_0, \ldots, a_n$. Per $\alpha \in \mathbb{R}$ vogliamo calcolare

$$f(\alpha) = a_0 \alpha^n + \ldots + a_n$$

Definiamo i numeri $b_0, \ldots, b_n$ nel modo seguente:

$$\begin{aligned} b_0 &:= a_0 \\ b_1 &:= a_0\alpha + a_1 \\ b_2 &:= a_0\alpha^2 + a_1\alpha + a_2 \\ &\ldots \\ b_k &= a_0\alpha^k + a_1\alpha^{k-1} + \ldots + a_k \\ &\ldots \\ b_n &:= a_0\alpha^n + \ldots + a_n \end{aligned}$$

Abbiamo quindi $b_n = f(\alpha)$.

D'altra parte è evidente la legge di ricorsione

$$b_k = b_{k-1}\alpha + a_k$$

per $k = 1, \ldots, n$. Queste relazioni forniscono un algoritmo che è detto *schema di Horner* o *schema di Ruffini*, molto più veloce del calcolo separato delle potenze di α (tranne nel caso che il polinomio consiste di una sola o di pochissime potenze, in cui si usa l'algoritmo del contadino russo, che vedremo fra poco).

Lo schema di Horner si presta bene anche al calcolo a mano, come vediamo in un esempio: Siano

$$f = 3x^4 + 15x^3 + 26x^2 - 8x + 2 \text{ ed } \alpha = 10$$

a_k	b_k $\qquad\qquad \alpha = 10$
3	3
15	$30 + 15 = 45$
26	$450 + 26 = 476$
-8	$4760 - 8 = 4752$
2	$47520 + 2 = 47522 = f(10)$

Vogliamo adesso scrivere una procedura in PostScript che realizza lo schema di Horner. Usiamo stavolta un approccio *modulare*, procedendo a piccoli passi, utilizzando funzioni di cui postuliamo l'effetto, ma che sono ancora da programmare. Contrassegniamo queste funzioni anteponendogli il simbolo `!`.

Osserviamo in primo luogo che, se poniamo $b_{-1} := 0$, si ha $b_0 = b_{-1}\alpha + a_0$, quindi la regola di ricorsione diventa valida anche per $k = 0$. Ponendo $m := n + 1$, dobbiamo ripetere la ricorsione m volte.

	$[a_0 \ldots a_n]\ \alpha$
`0`	$[a_0 \ldots a_n]\ \alpha\ 0$
`2 index`	$[a_0 \ldots a_n]\ \alpha\ 0\ [a_0 \ldots a_n]$
`length`	$[a_0 \ldots a_n]\ \alpha\ 0\ m$
`! passo`	applica la regola di ricorsione
`repeat`	`[]` $\alpha\ f(\alpha)$
`3 1 roll`	$f(\alpha)$ `[]` α
`pop pop`	$f(\alpha)$

Elaboriamo `passo`:

	$[a_k \ldots a_n]\ \alpha\ b$
`1 index`	$[a_k \ldots a_n]\ \alpha\ b\ \alpha$
`mul`	$[a_k \ldots a_n]\ \alpha\ b\alpha$
`3 -1 roll`	$\alpha\ b\alpha\ [a_k \ldots a_n]$
`! spezza`	$\alpha\ b\alpha\ a_k\ [a_{k+1} \ldots a_n]$
`4 1 roll`	$[a_{k+1} \ldots a_n]\ \alpha\ b\alpha\ a_k$
`add`	$[a_{k+1} \ldots a_n]\ \alpha,\ b\alpha + a_k$

`spezza`: $[a_0 \ldots a_n] \longmapsto a_0\ [a_1 \ldots a_n]$

	$[a_0 \ldots a_n]$
`dup`	$[a_0 \ldots a_n]\ [a_0 \ldots a_n]$
`0 get`	$[a_0 \ldots a_n]\ a_0$
`exch`	$a_0\ [a_0 \ldots a_n]$
`! togliprimo`	$a_0\ [a_1 \ldots a_n]$

`togliprimo`: $[a_0 \ldots a_n] \longmapsto [a_1 \ldots a_n]$

	$[a_0 \ldots a_n]$
`1`	$[a_0 \ldots a_n]\ 1$
`1 index`	$[a_0 \ldots a_n]\ 1\ [a_0 \ldots a_n]$
`length`	$[a_0 \ldots a_n]\ 1\ m$
`1 sub`	$[a_0 \ldots a_n]\ 1\ n$
`getinterval`	$[a_1 \ldots a_n]$

Lo schema di Horner corrisponde quindi alla funzione

```
/Alg.horner {0 2 index length
  {1 index mul 3 -1 roll
    dup 0 get exch
      1 1 index length 1 sub getinterval
   4 1 roll add} repeat
  3 1 roll pop pop} def
```

Esempio:

```
[3 15 26 -8 2] 10 Alg.horner 6 string cvs
/normale /Times-Roman findfont 10 scalefont def
normale setfont
10 0 moveto show
```

con lo stesso risultato che avevamo ottenuto prima: 47522

Lo stack dei dizionari

Il PostScript permette una gestione di variabili locali mediante dizionari (*dictionaries*) che possono essere annidati perché organizzati tramite un apposito stack. Con la prima riga in

```
4 dict begin
...
end
```

viene creato un dizionario di almeno 4 voci (il cui numero viene comunque automaticamente aumentato se vengono definite più voci). Tra `begin` e `end` tutte le abbreviazioni si riferiscono a questo dizionario se in esso si trova una tale voce (altrimenti il significato viene cercato nel prossimo dizionario sullo stack dei dizionari); con `end` perdono la loro validità.

Consideriamo ancora la funzione $f(x) = 3x^2+5x+1$. La macroistruzione che abbiamo trovato a pagina 3, benché breve, non è facilmente leggibile senza l'uso di un diagramma di flusso. L'uso di variabili locali mediante lo stack dei dizionari ci permette di ridefinire la funzione in un formato più familiare:

```
/f {1 dict begin /x exch def x x mul 3 mul
  x 5 mul add 1 add} def
```

Esaminiamo anche questa espressione mediante un diagramma di flusso. Usiamo α per il valore di input, per distinguerlo dal nome della variabile.

	α
`1 dict`	α *dizionario*
`begin`	α
`/x`	α, `/x`
`exch`	`/x`, α
`def`	
`x`	α
`x`	α, α
`mul`	α^2
`3`	α^2, 3
`mul`	$3\alpha^2$
`x`	$3\alpha^2$, α
`5`	$3\alpha^2$, α, 5
`mul`	$3\alpha^2$, 5α
`add`	$3\alpha^2 + 5\alpha$
`1`	$3\alpha^2 + 5\alpha$, 1
`add`	$3\alpha^2 + 5\alpha + 1$

L'istruzione `1 dict` crea un dizionario che prima viene posto sullo stack degli operandi; solo con il successivo `begin` il dizionario viene tolto dallo stack degli operandi e posto sullo stack dei dizionari. L'interprete adesso trova `/x` e quindi in questo momento l'ultimo elemento dello stack è `/x`, preceduto da α. Questi due elementi devono essere messi nell'ordine giusto mediante un `exch` per poter applicare il `def` che inserisce la nuova abbreviazione per α nell'ultimo dizionario dello stack degli operandi (cioè nel dizionario appena da noi creato) e toglie gli operandi dallo stack degli operandi. Ci si ricordi che `end` non termina l'espressione tra parentesi graffe ma chiude il `begin`, toglie cioè il dizionario dallo stack dei dizionari.

Naturalmente per calcolare $f(x)$ potremmo anche usare `[3 5 1] x Alg.horner`.

Operatori di confronto e operatori logici

I valori di verità sono `true` e `false`.

a b	`eq`	$(a = b)$
a b	`ne`	$(a \neq b)$
a b	`ge`	$(a \geq b)$
a b	`gt`	$(a > b)$
a b	`le`	$a \leq b$
a b	`lt`	$a < b$
a b	`and`	a e b
a b	`or`	a o b (in senso non esclusivo)
a b	`xor`	esattamente uno tra a e b
a	`not`	negazione logica di a

if ed ifelse

A sia un valore booleano ed `f`, `g` operazioni, rappresentate da espressioni racchiuse tra parentesi graffe.

`A f if`	`f` viene eseguita se A è vera.
`A f g ifelse`	se A è vera, viene eseguita `f`, altrimenti `g`.

Definiamo ad esempio una procedura per il calcolo del *fattoriale* $n! := 1 \cdot 2 \cdot 3 \cdots n$.

```
/Alg.fatt {dup 0 eq {pop 1}
  {dup 1 sub Alg.fatt mul} ifelse} def
```

Perché è necessario il `pop`?

Calcolo di potenze

Per il calcolo di potenze (ad esponenti naturali) lo schema di Horner non comporta vantaggi rispetto all'algoritmo elementare mediante un ciclo.

Esiste però un algoritmo molto veloce (detto spesso *algoritmo del contadino russo*) che formuliamo in maniera ricorsiva per la funzione f definita da $f(x, n) = x^n$:

$$f(x,n) = \begin{cases} 1 & \text{se } n = 0 \\ f(x^2, \frac{n}{2}) & \text{se } n \text{ è pari} \\ xf(x, n-1) & \text{se } n \text{ è dispari} \end{cases}$$

È facile tradurre questo algoritmo in un programma ricorsivo in PostScript. Definiamo prima la funzione `Alg.pari` che restituisce (cioè pone sullo stack) il valore `true` se l'argomento (cioè l'ultimo elemento sullo stack) è pari, e `false` altrimenti:

```
/Alg.pari {2 mod 0 eq} def
```

In `Alg.potenza` usiamo un dizionario per le variabili x ed n:

```
/Alg.potenza {2 dict begin /n exch def /x exch def
  n 0 eq {1}
  {n Alg.pari {x x mul n 2 idiv Alg.potenza}
    {x x n 1 sub Alg.potenza mul} ifelse} ifelse end} def
```

Per calcolare le terze potenze dei numeri naturali tra 0 e 5 possiamo adesso usare le istruzioni

```
10 10 moveto
[0 1 2 3 4 5] {3 Alg.potenza 20 string cvs show
  ( ) show} forall
```

naturalmente dopo aver impostato un insieme di caratteri.

Pensare in PostScript

„PostScript is a mysterious language, powerful and cryptic. It is expressive and complicated and yet surprisingly simple ... PostScript is a full-fledged programming language, and it is possible to accomplish almost any task with it. It includes many high-level language constructs and data structures. The PostScript language is a tool kit filled with hundreds of special-purpose tools ...

... the operand stack is the communication area between procedures; it is the only place where you can pass data between procedures ... In general, the cleanest, fastest and best programs are those that make judicious use of the operand stack ... the operand stack is used by every PostScript operator and procedure as the way to pass data and operands. The more cleanly your program supports this natural interface, the faster and more efficient it will be.“ (Glenn Reid)

G. Reid: Thinking in PostScript. Addison-Wesley 1990.
www.rightbrain.com/pages/books.html.

www.fiveacross.com/company/team.shtml. Glenn Reid è un veterano di Adobe e fondatore e CEO di Five Across.

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 (adesso numerati a partire dal coefficiente della potenza più bassa) sono univocamente determinati. Sia, usando per vettori la notazione di PostScript, $r_2(n) = [a_k \ldots a_0]$ il vettore i cui elementi sono queste cifre. Dalla rappresentazione binaria si deduce la seguente relazione ricorsiva:

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

che può esser facilmente tradotta in PostScript, usando le funzioni `Alg.pari` e `Vett.concat` definite precedentemente:

```
/Alg.rapp2 {1 dict begin /n exch def
  n 1 le {[n]}
  {n Alg.pari {n 2 idiv Alg.rapp2 [0] Vett.concat}
  {n 1 sub 2 idiv Alg.rapp2 [1] Vett.concat} ifelse}
  ifelse end} def
```

Una frequente applicazione dello schema di Horner è il calcolo del valore correspondente a una rappresentazione binaria o esadecimale. Infatti otteniamo $(1, 0, 0, 1, 1, 0, 1, 1, 1)_2$ con

```
[1 0 0 1 1 0 1 1 1] 2 Alg.horner
```

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

```
[10 15 7 3 0 5 14] 16 Alg.horner
```

Creiamo una funzione per stampare una stringa allineata a destra nel punto (h, v), dove h è la coordinata orizzontale che corrisponde alla fine della stringa.

	$a\ h\ v$
`2 index`	$a\ h\ v\ a$
`stringwidth`	$a\ h\ v\ dh\ dv$
`pop`	$a\ h\ v\ dh$
`3 -1 roll`	$a\ v\ dh\ h$
`exch`	$a\ v\ h\ dh$
`sub`	$a\ v$, $h_0 := h - dh$
`exch`	$a\ h_0\ v$
`moveto`	a
`show`	-

In PostScript la funzione è quindi data da

```
/Str.sta.ad {2 index stringwidth pop 3 -1 roll
  exch sub exch moveto show} def
```

Usiamo questa funzione per stampare una tabella che contiene, per i numeri naturali n tra 4 e 12, in ogni colonna n e $n!$, entrambi allineati a destra. Con

```
/s 10 string def /v 144 def
[4 5 6 7 8 9 10 11 12]
{/x exch def x s cvs 30 v Str.sta.ad
x Alg.fatt s cvs 100 v Str.sta.ad
/v v 10 sub def} forall
```

otteniamo

4	24
5	120
6	720
7	5040
8	40320
9	362880
10	3628800
11	39916800
12	479001600

mark

Estremamente potenti nella programmazione con PostScript sono i simboli e operatori che riguardano la *marcatura* dello stack: `mark`, `counttomark`, `cleartomark` e `]`.

Il primo pone una *marca* sullo stack; `counttomark` conta (naturalmente verso sinistra) il numero n degli oggetti sullo stack fino alla marca più vicina e pone n sullo stack; `cleartomark` (il meno usato dei tre) toglie tutti gli elementi dallo stack fino alla più vicina marca che rimane. Una marca è un oggetto come gli altri e più quindi essere tolta con `pop`:

```
10 0 moveto
mark 1 2 3 (A) (B) (C) mark pop (D) (E)
counttomark 2 string cvs show
5 {( ) show show} repeat
```

8 E D C B A

La seconda marca è stata subito ritolta dal `pop` e quindi non ha effetto, semplicemente perché non esiste più. Il `counttomark` conta quindi gli otto elementi `1 2 3 (A) (B) (C) (D) (E)`.

`mark` coincide in verità con il simbolo `[` che conosciamo dalla costruzione di vettori; infatti la seconda parentesi quadra, `]`, è invece un *operatore* il cui effetto è precisamente di creare un vettore e di eliminare la marca all'inizio di questo vettore. Per questa ragione i vettori possono essere anche annidati: `[[1 2 3] 4 [5]]` è un vettore con 3 componenti, di cui 2 sono a loro volta vettori; cfr. pag. 5.

Le espressioni `[1 2 3]` e `mark 1 2 3]` sono equivalenti e per sole ragioni di leggibilità si usa in genere `[` nella definizione di vettori, `mark` nelle altre operazioni di marcatura.

---

for

L'operatore `for` viene usato con la sintassi

```
a p b f for
```

dove `f` è una procedura spesso inclusa tra parentesi graffe, mentre a, p e b sono numeri interi. Si usa un contatore i, all'inizio si pone $i := a$, poi i viene aumentato del valore del passo p (quindi diminuito se p è negativo) fino a raggiungere b. Ogni volta i viene posto sullo stack ed eseguita la `f`. Esempio:

```
10 22 moveto /s 10 string def
1 2 13 {s cvs show ( ) show} for

/Quadrato {dup mul} def
10 10 moveto
13 -2 1 {dup s cvs show (:) show
  Quadrato s cvs show ( ) show} for
```

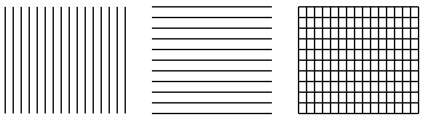
1 3 5 7 9 11 13
13:169 11:121 9:81 7:49 5:25 3:9 1:1

Il `for` di PostScript, essendo più elementare, è anche molto più potente e versatile nella programmazione del `for` di altri linguaggi, come si vede già dalle sorprendenti applicazioni del `for` „vuoto" (`{} for`) che vedremo fra poco. Anche nella grafica il `for` è spesso utilissimo:

Con

```
0.5 setlinewidth
0 3 45 {0 moveto 0 40 rlineto} for stroke
0 4 40 {55 exch moveto 45 0 rlineto} for stroke
110 3 155 {0 moveto 0 40 rlineto} for stroke
0 4 40 {110 exch moveto 45 0 rlineto} for stroke
```

otteniamo serie di linee orizzontali e verticali e, con le ultime due righe, una tavola a quadretti.



Vettori di zeri

Abbiamo già osservato che gli elementi di un vettore creato con `array` sono uguali all'oggetto `null`. Per creare invece un vettore di tutti zeri utilizziamo l'algoritmo

	n
`mark`	n `mark`
`exch`	`mark` n
`{0} repeat`	`mark` $0 \dots 0$
`counttomark`	`mark` $0 \dots 0\ n$
`array`	`mark` $0 \dots 0\ a$
`astore`	`mark` b
`exch pop`	b

Nella terzultima riga a è un vettore di oggetti `null` che con `astore` viene riempito con gli zeri che si trovano sullo stack e diventa b. Nell'ultima riga infine togliamo la marca.

Si vede che, ad esempio con `exch` o `roll`, possiamo anche spostare le marche!

Possiamo così definire la nostra funzione:

```
/Vett.zeri {mark exch {0} repeat
  counttomark array astore exch pop} def
```

Creiamo una funzione che, dato un vettore a di lunghezza n e un numero $m \geq n$, antepone agli elementi di a tanti zeri quanti sono necessari affinché il vettore c che si ottiene abbia lunghezza m.

	$a\ m$
`dup`	$a\ m\ m$
`2 index`	$a\ m\ m\ a$
`length`	$a\ m\ m\ n$
`sub`	$a\ m, i := m - n$
`exch`	$a\ i\ m$
`Vett.zeri`	$a\ i, b := [0 \dots 0]$
`dup`	$a\ i\ b\ b$
`4 2 roll`	$b\ b\ a\ i$
`exch`	$b\ b\ i\ a$
`putinterval`	c

Abbiamo quindi la funzione

```
/Vett.anteponizeri {dup 2 index length sub
  exch Vett.zeri dup 4 2 roll exch putinterval} def
```

Concatenamento efficiente

Utilizzando la tecnica delle marcature possiamo riscrivere `Vett.concat` senza utilizzare `putinterval` e `length`. Invece di `mark` usiamo il simbolo `[`. $a = [a_1 \dots a_n]$ e $b = [b_1 \dots b_m]$ siano i due vettori da concatenare. L'algoritmo è il seguente:

	$a\ b$
`[`	$a\ b$ `[`
`3 -1 roll`	b `[` a
`aload`	b `[` $a_1 \dots a_n\ a$
`pop`	b `[` $a_1 \dots a_n$
`counttomark`	b `[` $a_1 \dots a_n\ n$
`2 add`	b `[` $a_1 \dots a_n, n+2$
`-1 roll`	`[` $a_1 \dots a_n\ b$
`aload`	`[` $a_1 \dots a_n\ b_1 \dots b_m\ b$
`pop`	`[` $a_1 \dots a_n\ b_1 \dots b_m$
`]`	`[` $a_1 \dots a_n\ b_1 \dots b_m$ `]`

Ciò si traduce nella nuova versione di `Vett.concat`:

```
/Vett.concat {[ 3 -1 roll aload pop
  counttomark 2 add -1 roll aload pop ]} def
```

Gli operatori `aload` e `astore` non sono definiti per stringhe, quindi non possiamo direttamente riscrivere `Str.concat`. Possiamo però creare due nostre funzioni `Str.aload` e `Str.astore` a questo scopo.

Per `Str.aload` usiamo il seguente algoritmo per una stringa s, le cui lettere sono $a_1, \dots, a_n$.

	s
`[`	s `[`
`1 index`	s `[` s
`{} forall`	s `[` $a_1 \dots a_n$
`counttomark`	s `[` $a_1 \dots a_n\ n$
`2 add`	s `[` $a_1 \dots a_n, n+2$
`-2 roll`	$a_1 \dots a_n\ s$ `[`
`pop`	$a_1 \dots a_n\ s$

Otteniamo così la funzione

```
% Sembra non funzioni per stringhe con piu' di 798 caratteri.
/Str.aload {[ 1 index {} forall counttomark
  2 add -2 roll pop} def
```

È anche evidente che la stessa funzione applicata a un vettore opera come `aload` e quindi avremmo potuto programmare da soli quella funzione se non esistesse già.

Nell'utilizzo di `Str.aload` per stringhe ci si ricordi che le lettere che compongono una stringa diventano numeri quando vengono estratte.

Nella quarta riga dell'algoritmo abbiamo usato che un'istruzione `a {} forall` pone gli elementi di `a` sullo stack!

Per `Str.astore` abbiamo bisogno di una funzione `Str.davettore` che trasforma un vettore $a = [a_0 \dots a_{n-1}]$ di n bytes in una stringa s e che otteniamo dall'algoritmo

	a
`dup`	$a\ a$
`length dup`	$a\ n\ n$
`string`	$a\ n, s :=$ stringa di n zeri
`exch`	$a\ s\ n$
`1 sub`	$a\ s, m := n - 1$
`0 1`	$a\ s\ m\ 0\ 1$
`3 -1 roll`	$a\ s\ 0\ 1\ m$
`{`	`{` $a\ s, i :=$ contatore del `for`
`1 index`	$a\ s\ i\ s$
`exch`	$a\ s\ s\ i$
`dup`	$a\ s\ s\ i\ i$
`4 index`	$a\ s\ s\ i\ i\ a$
`exch`	$a\ s\ s\ i\ a\ i$
`get`	$a\ s\ s\ i, x := a_i$
`put}`	$a\ s$`}`
`for`	$a\ s$
`exch pop`	s

La parte compresa tra parentesi graffe viene eseguita per ogni $i = 0, \dots, m$. Abbiamo cosi la funzione

```
/Str.davettore {dup length dup string exch 1 sub 0 1
  3 -1 roll {1 index exch dup 4 index exch get put} for
  exch pop} def
```

Da essa otteniamo facilmente un algoritmo per `Str.astore`:

	$a_1 \dots a_n\ s$
`length`	$a_1 \dots a_n\ n$
`[`	$a_1 \dots a_n\ n$ `[`
`exch`	$a_1 \dots a_n$ `[` n
`1 add`	$a_1 \dots a_n$ `[`$, n+1$
`1 roll`	`[` $a_1 \dots a_n$
`]`	$b := [a_1 \dots a_n]$
`Str.davettore`	t

L'operatore `]` nella penultima riga raccoglie gli elementi precedenti (fino alla marca) in un vettore! Otteniamo così la nostra funzione:

```
/Str.astore {length [ exch 1 add 1 roll ] Str.davettore} def
```

Adesso possiamo riscrivere anche `Str.concat`, utilizzando l'algoritmo della nuova versione di `Vett.concat`:

```
/Str.concat {[ 3 -1 roll Str.aload pop
  counttomark 2 add -1 roll Str.aload pop counttomark
  array Str.astore exch pop} def
```

Vettori da stringhe

È semplicissimo anche la trasformazione di una stringa s in un vettore:

	s
`[ exch`	`[` s
`{} forall`	`[` $a_1 \dots a_n$
`]`	`[`$a_1 \dots a_n$`]`

quindi

```
/Vett.dastringa {[ exch {} forall ]} def
```

Per verificare la correttezza provare

```
(Roma) Vett.dastringa Str.davettore show
```

Togliere il primo elemento da un vettore

In modo molto simile possiamo programmare una funzione che toglie il primo elemento da un vettore non vuoto, senza usare `length` e `getinterval` come invece avevamo fatto nella parte `togliprimo` dello schema di Horner a pagina 6:

	$[a_0 \dots a_n]$
`[ exch`	`[` $[a_0 \dots a_n]$
`aload`	`[` $a_0 \dots a_n [a_0 \dots a_n]$
`pop`	`[` $a_0 \dots a_n$
`counttomark`	`[` $a_0 \dots a_n, n+1$
`-1 roll`	`[` $a_1 \dots a_n a_0$
`pop ]`	$[a_1 \dots a_n]$

per cui

```
/Vett.togliprimo {[ exch aload pop counttomark
  -1 roll pop ]} def
```

Sostituendo questa funzione in `Alg.horner` otteniamo una versione più efficiente:

```
/Alg.horner {0 2 index length
  {1 index mul 3 -1 roll
    dup 0 get exch Vett.togliprimo
   4 1 roll add} repeat
  3 1 roll pop pop} def
```

map

Molti linguaggi di programmazione ad alto livello contengono un'operazione `map` con cui da due argomenti, di cui uno è un vettore $a = (a_1, \dots, a_n)$, l'altro una funzione f, si ottiene il vettore $\mathrm{map}(a, f) = (f(a_1), \dots, f(a_n))$.

Quanto siano potenti i meccanismi elementari di PostScript si vede dalla quasi incredibile facilità con cui possiamo realizzare (e generalizzare, se vogliamo) questa operazione avanzata. L'algoritmo è

	$a\ f$
`[`	$a\ f$ `[`
`3 1 roll`	`[` $a\ f$
`forall`	`[` $f(a_1) \dots f(a_n)$
`]`	`[`$f(a_1) \dots f(a_n)$`]`

Perciò la funzione ricercata è semplicemente

```
/Vett.map {[ 3 1 roll forall ]} def
```

Usiamo `Vett.map` per trasformare la rappresentazione binaria di un numero naturale nella stringa corrispondente. Ad esempio $n = 13$ possiede la rappresentazione binaria $(1,1,0,1)_2$ che corrisponde al vettore $[1101]$ da cui vogliamo ottenere la stringa *1101*. Dobbiamo quindi trasformare 0 e 1 nei loro codici ASCII che sono 48 e 49. Da $[1101]$ otteniamo così, usando `Vett.map`, il vettore $[49\ 49\ 48\ 49]$ da cui si trova la stringa cercata con `Str.davettore`. L'algoritmo è perciò

	n
`Alg.rapp2`	a, ad esempio a=`[1 1 0 1 1 0 1]`
`...`	a `{0 eq {48} {49} ifelse}`
`Vett.map`	b, ad esempio b=`[49 49 48 49 49 48 49]`
`Str.davettore`	s

per cui possiamo definire la funzione

```
/Str.dabin {Alg.rapp2 {0 eq {48} {49} ifelse}
  Vett.map Str.davettore} def
```

Nello schema dell'algoritmo i tre puntini nella terza riga indicano che si vedono a destra le istruzioni da inserire.

Possiamo così stampare i numeri $0, 1, 2, 3, 4, 5, 10, 25, 128, 130, 460$ e le loro rappresentazioni binarie. Con

```
/v 120 def /s 30 string def
[0 1 2 3 4 5 10 25 128 130 460]
{dup s cvs 30 v Str.sta.ad
Str.dabin 120 v Str.sta.ad /v v 10 sub def} forall
```

otteniamo

0	0
1	1
2	10
3	11
4	100
5	101
10	1010
25	11001
128	10000000
130	10000010
460	111001100

Perché è necessario il `dup` nella terza riga?

Talvolta, per ragioni tipografiche o per rappresentare punti di un ipercubo, si vorrebbe che le stringhe corrispondenti alla rappresentazione binaria abbiano tutte la stessa lunghezza m, anteponendo zeri nelle posizioni mancanti. Per fare ciò, è sufficiente modificare leggermente l'algoritmo:

	$n\ m$
`exch`	$m\ n$
`Alg.rapp2`	$m\ a$
`exch`	$a\ m$
`Vett.anteponizeri`	b
`...`	b `{0 eq {48} {49} ifelse}`
`Vett.map`	c
`Str.davettore`	s

e quindi definiamo

```
/Str.dabin.anteponizeri {exch Alg.rapp2 exch
  Vett.anteponizeri {0 eq {48} {49} ifelse}
  Vett.map Str.davettore} def
```

Con

```
/v 110 def /s 30 string def
[0 1 2 3 4 5 10 25 128 130 460]
{dup s cvs 30 v Str.sta.ad
9 Str.dabin.anteponizeri 120 v Str.sta.ad
/v v 10 sub def} forall
```

otteniamo adesso

0	000000000
1	000000001
2	000000010
3	000000011
4	000000100
5	000000101
10	000001010
25	000011001
128	010000000
130	010000010
460	111001100

Come in `Vett.anteponizeri` anche in `Str.dabin.anteponizeri` l'argomento m non indica il numero degli zeri da anteporre, ma la lunghezza complessiva del vettore.

L'ipercubo

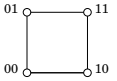
Sia $X = \{1, \ldots, n\}$ con $n \geq 0$.

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

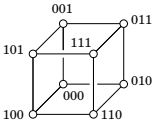
$n = 0$:

$n = 1$:

$n = 2$:



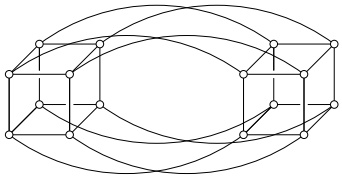
$n = 3$:



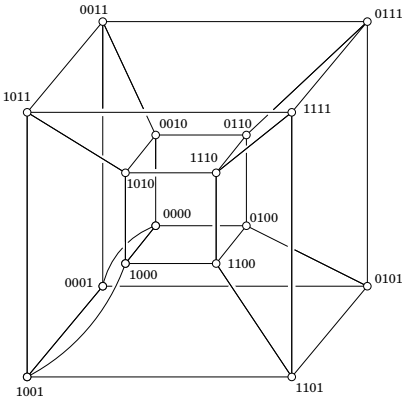
$n = 4$: L'ipercubo a 4 dimensioni si ottiene dal cubo 3-dimensionale attraverso la relazione

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

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 $X = \{1, 2, 3, 4\}$). Con

```
/v 160 def /s 30 string def
0 1 15 {dup s cvs 30 v Str.sta.ad
4 Str.dabin.anteponizeri 80 v Str.sta.ad
/v v 10 sub def} for
```

possiamo elencare gli elementi dell'ipercubo 2^4. Esaminare le differenze con l'ultima istruzione a pagina 10.

Una differenza fondamentale tra [e]

Esiste una differenza fondamentale tra `[` e `]`. Mentre la marca `[` è un semplice oggetto *passivo* che può essere messo sullo stack senza influenzare gli altri elementi, `]` è un *operatore* che, posto sullo stack, induce una specifica operazione.

L'operazione $[a_0 \ldots a_n] \longmapsto a_0\ [a_1 \ldots a_n]$

Conosciamo questa operazione da pagina 6. Se usiamo la funzione `Vett.togliprimo` di pagina 10, otteniamo la procedura

```
/Vett.spezza {dup 0 get exch Vett.togliprimo} def
```

con cui la funzione per lo schema di Horner diventa

```
/Alg.horner {0 2 index length
  {1 index mul 3 -1 roll Vett.spezza
   4 1 roll add} repeat
  3 1 roll pop pop} def
```

Esercizi per gli scritti

1. Completare

	3 5
`dup`	
`mul`	
`exch`	
`1 add`	
	100

2. Completare

	34 60 5
	34 12
`1 index`	
`1`	
	34 1 12 34
`add`	
`exch`	
`sub`	
	79

3. Completare

	4 6 9 3 5
	4 6 9
`2 index`	
`mul`	
`3 -1 roll`	
`sub`	
`add`	
	19

4. Diagramma di flusso (nel formato abbreviato usato negli esercizi precedenti) e macroistruzione per la funzione f definita da

$$f(x) = 2x^4 + 6x^2 - 3x + 10$$

5. Completare

	[2 3 4 6 5]
	2 3 4 6 5
	2 4 6 5 3
`add`	
`dup`	
`add`	
`3 array`	
	2 [4 6 16]
	[4 6 16]
	4 6 16
`add`	
	22 4
	18

6. Visualizzare come stringhe gli elementi del vettore `[8 10 6 205 1 2 7]`.

ALGORITMI E STRUTTURE DI DATI

Conversione ed esecuzione con Ghostscript

La conversione di un file PostScript *alfa.ps* nell'immagine *alfa.png* può anche avvenire usando direttamente *Ghostscript* invece di *convert* (infatti convert stesso utilizza Ghostscript). A questo scopo sotto Linux usiamo il comando

```
gs -q -dBATCH -dNOPAUSE -sDEVICE=png16m
  -sOutputFile=alfa.png alfa.ps
```

scritto tutto su una riga (sotto Windows forse `gswin32c` invece di `gs`). Bisogna subito dire che ciò è pericoloso, soprattutto quando non conosciamo il contenuto del file che intendiamo convertire. Infatti PostScript prevede anche comandi con cui si possono creare, scrivere o distruggere files sul nostro computer e questi comandi, se presenti nel file, vengono eseguiti anche se apparentemente stiamo solo creando un immagine in formato .png. È quindi consigliabile inserire l'opzione `-dSAFER` nell'istruzione; ciò ad esempio avviene nei programmi ausiliari che permettono di vedere i files PostScript mediante un browser Web (quando sono correttamente programmati), mentre sembra che non sia così in alcune versioni di *convert*.

Le opzioni `-dBATCH` e `-dNOPAUSE` sono entrambe necessarie perché l'esecuzione non venga interrotta quando l'interprete incontra `showpage`. `-dBATCH` è equivalente a `-c quit` alla fine del comando.

Quando si usa `gs` direttamente invece che tramite `convert`, bisogna aggiungere `showpage` alla fine del file; ciò vale soltanto quando si desidera creare un immagine, ma non per l'esecuzione sul terminale.

Vengono inoltre ignorati i parametri di `%%BoundingBox` e quindi larghezza e altezza devono essere indicati mediante opzioni della forma

```
-dDEVICEWIDTHPOINTS=800 -dDEVICEHEIGHTPOINTS=300
```

Ciò rende il comando di conversione piuttosto complesso ed è più semplice usare *convert*.

L'uso di `gs` è invece utile quando vogliamo usare il programma PostScript sul terminale. Allora usiamo il comando

```
gs -q -dBATCH -dNOPAUSE -dNODISPLAY alfa.ps
```

Molto comodo è delegare l'esecuzione a un piccolo programma, ad esempio in Python:

```
#! /usr/local/bin/python
import os

file=' alfa.ps'
opzioni=' -q -dBATCH -dNOPAUSE -dNODISPLAY'

os.system('gs'+opzioni+file)
```

Se il file che contiene questo programma si chiama *alfa*, è sufficiente battere *alfa* dalla shell per far eseguire *alfa.ps* tramite Ghostscript.

Output su terminale

Finora per visualizzare stringhe abbiamo usato il comando `show`. Questo in verità non è un comando di output sul terminale, ma un comando con cui una stringa viene *disegnata* nella figura che vogliamo creare. Dobbiamo infatti distinguere l'output grafico che vogliamo ottenere dall'output su terminale per il quali si usano altri comandi: `print` (per stringhe), `=` e `==` (per oggetti qualsiasi), `stack` e `pstack` (per lo stack intero). Più precisamente queste istruzioni hanno i seguenti effetti:

s `print`	Stampa la stringa *s*
`x =`	Toglie l'oggetto `x` dallo stack e lo stampa sul terminale, aggiungendo un carattere di nuova riga. Cosa si vede, dipende dal tipo di oggetto.
`x ==`	Simile, ma spesso più leggibile.
`stack`	Stampa con la stessa modalità di `=` tutto lo stack con un carattere di nuova riga dopo ogni oggetto, ma non toglie elementi dallo stack.
`pstack`	Simile, ma nella modalità di `==`.

In questo numero

Per stampare un carattere di nuova riga sul terminale possiamo ad esempio usare `()=`, oppure `(\n) print`. Si noti che il carattere `\n` viene interpretato come carattere di nuova riga nell'output sul terminale, ma non da `show`.

Soprattutto nei programmi interattivi talvolta dopo l'istruzione di output è necessario il comando `flush` affinché l'operazione di output venga effettuata in tempo reale.

J. Warnock/C. Geschke (ed.): PostScript language reference. Addison-Wesley 1999. Per il comandi di output sul terminale vedere le pagine 74, 87, 513, 594, 633, 635, 692.

Commenti

Se il carattere `%` non si trova all'interno di una stringa, esso indica che il resto della riga in cui si trova, incluso il carattere `%` stesso, viene considerato come commento, cioè ignorato dall'interprete.

L'alfabeto greco

alfa	α	A
beta	β	B
gamma	γ	Γ
delta	δ	Δ
epsilon	ε	E
zeta	ζ	Z
eta	η	H
theta	θ	Θ
iota	ι	I
kappa	$\varkappa$	K
lambda	λ	Λ
mi	μ	M
ni	ν	N
xi	ξ	Ξ
omikron	o	O
pi	π	Π
rho	ϱ	P
sigma	σ,ς	Σ
tau	τ	T
ypsilon	υ	Υ
fi	φ	Φ
chi	χ	X
psi	ψ	Ψ
omega	ω	Ω

Sono 24 lettere. La sigma minuscola ha due forme: alla fine della parola si scrive ς, altrimenti σ. In matematica si usa solo la σ.

Mikros ($\mu\iota\varkappa\rho\acute{o}\varsigma$) significa piccolo, megas ($\mu\acute{\varepsilon}\gamma\alpha\varsigma$) grande, quindi la omikron è la *o* piccola e la omega la *o* grande.

Le lettere greche vengono usate molto spesso nella matematica, ad esempio $\sum_{k=0}^{3} a_k$ è un'abbreviazione per la somma $a_0+a_1+a_2+a_3$ e $\prod_{k=0}^{3} a_k$ per il prodotto $a_0\,a_1\,a_2\,a_3$, mentre A^i_α è un oggetto con due indici, per uno dei quali abbiamo usato una lettera greca.

I numeri binomiali

Situazione 13.1. $n \in \mathbb{N}$, X un insieme finito con n elementi.

Definizione 13.2. Denotiamo con $|X|$ il numero degli elementi di X.
In particolare $|\emptyset| = 0$.

Nota 13.3. Se A e B sono sottoinsiemi di X tali che $X = A \dot\cup B$ (cioè $X = A \cup B$ e $A \cap B = \emptyset$), allora

$$|X| = |A| + |B|.$$

Definizione 13.4. Per $k \in \mathbb{N}$ denotiamo con $\mathcal{P}(X, k)$ l'insieme di quei sottoinsiemi di X che possiedono esattamente k elementi:

$$\mathcal{P}(X, k) = \{A \subset X \mid |A| = k\}$$

Esempio 13.5. Sia $X = \{1, 2, 3, 4, 5\}$. Quindi $|X| = 5$. Quanti sottoinsiemi con esattamente 2 elementi possiede X?
Questi sottoinsiemi sono $\{1, 2\}$, $\{1, 3\}$, $\{1, 4\}$, $\{1, 5\}$, $\{2, 3\}$, $\{2, 4\}$, $\{2, 5\}$, $\{3, 4\}$, $\{3, 5\}$, $\{4, 5\}$. X possiede perciò 10 sottoinsiemi con 2 elementi, in altre parole $|\mathcal{P}(X, 2)| = 10$.

Definizione 13.6. Per $k \in \mathbb{N}$ sia $\binom{n}{k}$ il numero dei sottoinsiemi di X che possiedono esattamente k elementi. Quindi

$$\binom{n}{k} = |\mathcal{P}(X, k)|$$

I numeri della forma $\binom{n}{k}$ si chiamano *coefficienti binomiali* o *numeri binomiali*. La definizione naturalmente non dipende da X, ma solo da n.

Osservazione 13.7.
$\binom{n}{0} = 1$ *perché* $\emptyset$ *è l'unico sottoinsieme di* X *con* 0 *elementi.*
$\binom{n}{n} = 1$ *perché* X *è l'unico sottoinsieme con* n *elementi.*
$\binom{n}{1} = n$ *perché i sottoinsiemi con un elemento corrispondono univocamente agli elementi di* X*, e di questi ce ne sono proprio* n*.*

Proposizione 13.8. $\binom{n}{n-k} = \binom{n}{k}$ *per* $0 \le k \le n$.

Dimostrazione. Ogni sottoinsieme $A \subset X$ determina univocamente il suo complemento $X \setminus A$ e viceversa. Se A ha k elementi, $X \setminus A$ ne ha $n - k$. Abbiamo quindi una biiezione tra $\mathcal{P}(X, k)$ e $\mathcal{P}(X, n - k)$.

Corollario 13.9. $\binom{n}{n-1} = n$.

Osservazione 13.10. $\binom{n}{k} = 0$ *per* $k > n$.

Dimostrazione. X non può possedere sottoinsiemi con più di n elementi, perché si ha sempre $|A| \le |X|$ per $A \subset X$.

Nota 13.11. e sia un elemento esterno che non appartiene ad X. Allora $X \cup \{e\}$ possiede $n + 1$ elementi. Sia $k \in \mathbb{N}$ con $1 \le k \le n$.

Siccome ogni sottoinsieme di X è anche un sottoinsieme di $X \cup \{e\}$, i sottoinsiemi di $X \cup \{e\}$ con k elementi sono o sottoinsiemi con k elementi di X, oppure consistono di e e di altri $k - 1$ elementi appartenenti ad X.
Vediamo così che $\mathcal{P}(X \cup \{e\}, k)$ è uguale a

$$\mathcal{P}(X, k) \dot\cup \{A \dot\cup \{e\} \mid A \in \mathcal{P}(X, k - 1)\}$$

per cui

$$\binom{n+1}{k} = \binom{n}{k} + \binom{n}{k-1}$$

per $1 \le k \le n$.

Questa formula vale però anche per $k > n$:

Infatti per $k = n+1$ abbiamo a sinistra $\binom{n+1}{n+1} = 1$ e anche a destra $\binom{n}{n+1} + \binom{n}{n} = 0 + 1 = 1$; per $k \ge n + 2$ abbiamo 0 sia a destra che a sinistra.

Teorema 13.12.

$$\boxed{\binom{n+1}{k} = \binom{n}{k} + \binom{n}{k-1}}$$

per ogni $n \ge 0$ *ed ogni* $k \ge 1$.

Dimostrazione. Nota 13.11.

Osservazione 13.13. *Il teorema 13.12 è equivalente al triangolo di Pascal:*

```
                    1
                  1   1
                1   2   1
              1   3   3   1
            1   4   6   4   1
          1   5   10  10  5   1
        1   6   15  20  15  6   1
```

Teorema 13.14 (teorema binomiale).

$$(a + b)^n = \sum_{k=0}^{n} \binom{n}{k} a^{n-k} b^k$$

per ogni $a, b \in \mathbb{R}$. Questa formula è più in generale valida in ogni anello commutativo con unità.
Dimostrazione. Non richiesta, ma facile. Corsi di Geometria o Analisi.

Definizione 13.15. Poniamo

$$n! := 1 \cdot 2 \cdots (n-1)n \quad \text{per } n \ge 1$$
$$0! := 1$$

$n!$ si pronuncia *n fattoriali*; si parla anche del *fattoriale di* n.
Dalla definizione segue direttamente l'equazione funzionale

$$(n + 1)! = n!(n + 1)$$

Teorema 13.16. $\binom{n}{k} = \dfrac{n!}{k!(n-k)!}$ *per* $0 \le k \le n$.

Dimostrazione. Facile per induzione su n, fissato k. Non richiesta.

Nota 13.17. La formula del teorema 13.15 è molto importante nella teoria, ma nei conti a mano può essere utilizzata solo per numeri molto piccoli. Infatti per calcolare $\binom{200}{4}$ con il teorema 13.15 dovremmo calcolare 200! e 196!, due numeri giganteschi, e formare $\frac{200!}{4!196!}$. In pratica si usa invece la regola

$$\binom{n}{k} = \frac{n(n-1)\cdots(n-k+1)}{1 \cdot 2 \cdots k}$$

dove sopra stanno tanti numeri quanti sotto, cioe k.
Ad esempio

$$\binom{12}{3} = \frac{12 \cdot 11 \cdot 10}{1 \cdot 2 \cdot 3}$$
$$\binom{20}{5} = \frac{20 \cdot 19 \cdot 18 \cdot 17 \cdot 16}{1 \cdot 2 \cdot 3 \cdot 4 \cdot 5}$$
$$\binom{200}{4} = \frac{200 \cdot 199 \cdot 198 \cdot 197}{1 \cdot 2 \cdot 3 \cdot 4}$$

Non è difficile giustificare questa regola (esercizio 7).

Definizione 13.18. I numeri

$$\binom{n}{k}^{+} := \frac{n(n+1)\cdots(n+k-1)}{1 \cdot 2 \cdots k}$$

si chiamano *binomiali superiori*. Anche qui sia sopra che sotto stanno k numeri. Infatti

$$\boxed{\binom{n}{k}^{+} = \binom{n+k-1}{k}}$$

Basta leggere il numeratore all'indietro nella definizione. Nonostante ciò, i binomiali superiori hanno interessanti interpretazioni combinatoriche.

Uso del logaritmo

Per calcolare fattoriali e binomiali molto grandi al calcolatore conviene utilizzare il logaritmo. Siccome

$$0! = 1 \quad \text{e} \quad n! = (n-1)!n \quad \text{per} \quad n \geq 1$$

abbiamo

$$\log 0! = 0 \quad \text{e}$$
$$\log n! = \log(n-1)! + \log n \quad \text{per} \quad n \geq 1$$

da cui si deduce facilmente un algoritmo.

Quando si usa il logaritmo, si può anche usare il teorema 20.15 per calcolare i numeri binomiali:

$$\log \binom{n}{k} = \log n! - \log k! - \log(n-k)!$$

Quando questi numeri, ad esempio $n!$ per $0 \leq n \leq 1000$, servono spesso, conviene creare una tabella.

La formula di Stirling

Teorema 14.1. *Il valore di $n!$ è sempre compreso tra*

$$n^n e^{-n} \sqrt{2\pi n}\ e^{\frac{1}{12n} - \frac{1}{120n^2}}$$

e

$$n^n e^{-n} \sqrt{2\pi n}\ e^{\frac{1}{12n} + \frac{1}{120n^2}}$$

Dimostrazione. Non richiesta. Richiede gli strumenti potenti dell'analisi complessa.

Calcolo di fattoriali e prodotti

A pagina 7 abbiamo dato un algoritmo procedurale e ricorsivo per il calcolo del fattoriale che usa un'istruzione `ifelse`. Questa impostazione dell'algoritmo non è però del tutto naturale in PostScript, dove tutte le operazioni avvengono sullo stack (cfr. il brano di Glenn Reid che abbiamo citato su quella stessa pagina), una struttura unidimensionale (una specie di scacchiera unidimensionale) che non si presta bene a diramazioni.

Cerchiamo a questo punto di formulare alcuni principi fondamentali per una buona programmazione in PostScript.

(1) Le sequenze di istruzioni nelle procedure di PostScript sono difficilmente comprensibili e non possono essere memorizzate (e quindi non verranno nemmeno chieste all'esame).

(2) Bisogna invece analizzare i cambiamenti sullo stack durante l'esecuzione di una procedura.

(3) Soltanto quando questa successione di cambiamenti è stata definita, vengono scritte le istruzioni elementari che connettono i singoli stati dello stack. In questa fase possiamo usare la tecnica della tabella con la riga verticale che già conosciamo. Questa fase la chiamiamo *analisi operazionale*.

(4) Siccome in PostScript le parentesi tonde non svolgono un compito sintattico essenziale (vengono usate solo per definire le stringhe), le useremo per denotare un particolare stato dello stack. (a b c) significa che gli ultimi tre elementi dello stack sono a, b e c (come sempre lo stack cresce verso destra e quindi c è l'ultimo elemento). Si possono separare componenti con virgole facoltative; stringhe vengono denotate tra virgolette come in C.
Siccome si tratta soltanto di una metanotazione che adottiamo nell'analisi, possiamo usare indici inferiori e superiori, radici, scrivere il prodotto di a e b nella forma ab, ecc. Un'assegnazione della forma $x = a + b$ significa, come d'uso in altri linguaggi, che x assume il valore $a + b$ e può essere usato con questo valore nel seguito.

(5) Talvolta all'interno di uno stato dello stack e separato dalla parte precedente con un doppio punto (:) aggiungiamo il nome di una funzione o un commento che spiega in che modo questo stato viene raggiunto. Così ad esempio (a b : f) indica che lo stato (a b) è stato raggiunto usando la funzione f.

(6) La descrizione dei cambiamenti dello stack così ottenuta prende il nome di *programma analitico*.

(7) È molto utile aggiungere il programma analitico alle funzioni della libreria che creiamo in forma di commenti. Allora cercheremo di esprimere i simboli eventualmente usati (punto (4)) in modo comprensibile mediante caratteri ASCII, talvolta usano uno pseudo-Latex.

(8) Ai fini della trascrizione del programma analitico che avviene nell'analisi operazionale introdurremo alcune abbreviazione per le operazioni fondamentali elementari sullo stack, come `S` (scambio) per `exch`, `RC3` (rotazione contraria) per `3 -1 roll`, `R4` (rotazione) per `4 1 roll`, `U` (ultimo) per `dup`, `T` (togli) per `pop`, ecc. Nomi brevi simili per gli operatori di base sono spesso già definiti in Forth, mentre mancano in PostScript.

Lo sviluppo di un programma in PostScript avviene quindi in 3 fasi:

programma analitico
analisi operazionale
programma eseguibile

Applichiamo queste idee generali adesso ad alcuni compiti di programmazione in parte già visti.

Dati $a, b \in \mathbb{Z}$ con $a \leq b$ vogliamo calcolare il prodotto $a(a+1)\cdots b$ dei numeri interi compresi tra a e b.

Programma analitico (che, come sempre, comincia con lo stato iniziale dello stack che costituisce l'input della funzione):

(a b) ($p = 1$, a 1 b) {(p i) (pi)} `for` (p)

Si noti che la parte `{ ... } for` non è considerata uno stato dello stack e deve essere inserita nella forma indicata. Lo stesso vale per altre istruzioni di controllo, ad esempio `if`, `ifelse`, `forall`.

Abbiamo introdotto la variabile p per il valore dei prodotti intermedi, mentre i è la variabile contatrice del `for` che automaticamente viene posta (aumentando ogni volta il suo valore di 1) sullo stack all'inizio di ogni passaggio del ciclo. pi è il prodotto di p ed i.

Analisi operazionale: Qui dobbiamo saper indicare come si passa da uno stato dello stack a quello successivo. Si noti che nel ciclo il primo stato (p i) è lo stato in cui si trova lo stack all'inizio di ogni passaggio del ciclo.

Cerchiamo adesso di ricostruire il programma analitico, aggiungendo stati intermedi, quando un passaggio non è immediato, come ad esempio in (a b) (1 a 1 b). Naturalmente si procede riga per riga nello schema che segue, scrivendo sempre prima la parte a destra, ciò lo stato dello stack che si vuole raggiungere. Utilizziamo anche le abbreviazioni che abbiamo annunciato e che verranno elencate in seguito.

	a b
`1 1`	a b 1 1
`R4`	1 a b 1
`S`	$p = 1$, a 1 b
`{`	{ p i
`mul}`	pi}
`for`	p

Il programma eseguibile è perciò

```
/Alg.prod {1 1 R4 S {mul} for} def
```

Da `Alg.prod` otteniamo facilmente un programma per il fattoriale.

Programma analitico: (n) (1 n) ($n!$: `Alg.prod`)

Analisi operazionale:

	n
`1 S`	1 n
`Alg.prod`	$n!$

cosicché il programma eseguibile diventa

```
/Alg.fatt {1 S Alg.prod} def
```

Calcolo dei numeri binomiali

Per calcolare i numeri binomiali usiamo la regola della nota 13.19 che possiamo scrivere nella forma

$$\binom{n}{k} = \frac{\mathrm{prod}(n-k+1, n)}{k!}$$

dove $\mathrm{prod}(a, b)$ corrisponde al nostro `a b Alg.prod`.

Programma analitico:

$(n\ k)\,(n\ k,\ f = k!)\,(f,\ n-k+1,\ n)$
$(f,\ \mathrm{prod}(n-k+1, n))\,(\binom{n}{k} : \texttt{idiv})$

Analisi operazionale:

	$n\ k$
`D Alg.fatt`	$n\ k,\ f = k!$
`R3`	$f\ n\ k$
`PU S`	$f\ n\ n\ k$
`sub 1 add`	$f\ n,\ n-k+1$
`S`	$f,\ n-k+1,\ n$
`Alg.prod`	$f,\ \mathrm{prod}(n-k+1, n)$
`S idiv`	$\binom{n}{k}$

usando l'abbreviazione `PU` (penultimo) per `1 index`.

Si osservi che a destra non abbiamo scritto direttamente gli stati del programma analitico, ma siamo proceduti a passi più piccoli che devono però comprendere tutti gli stati del programma analitico.

Programma eseguibile:

```
/Alg.bin {U Alg.fatt R3 PU S
  sub 1 add S Alg.prod S idiv} def
```

Per provarlo usiamo

```
0 1 11 {11 exch Alg.bin =} for
```

ottenendo l'output corretto

```
1
11
55
165
330
462
462
330
165
55
11
1
```

Come già osservato, è molto utile aggiungere il programma analitico all'eseguibile, per poter riconoscere più tardi l'algoritmo. Le ultime tre funzioni potrebbero quindi comparire nella nostra libreria nella forma seguente:

```
/Alg.prod % (a b) (p=1 a 1 b) {(p i) (pi)} for (p)
  {1 1 R4 S {mul} for} def

/Alg.fatt % (n) (1 n) (n! : Alg.prod)
  {1 S Alg.prod} def

/Alg.bin % (n k) (n k f=k!) (f n-k+1 n)
  % (f prod(n-k+1,n)) (bin(n,k) : idiv)
  {U Alg.fatt R3 PU S
  sub 1 add S Alg.prod S idiv} def
```

Alcuni programmi rivisti

Vogliamo adesso riesaminare alcuni dei programmi scritti nel primo numero per vedere se li possiamo semplificare applicando le tecniche di analisi che abbiamo imparato.

Per il calcolo della potenza usiamo lo stesso algoritmo come a pagina 7, ma le nuove tecniche di analisi ci permettono di scrivere il programma senza l'uso del dizionario.

Programma analitico:

$(x\ n)\,(x\ n,\ n = 0?)$ {$(x\ n)\,(1)$}
{$(x\ n)\,(x\ n,\ n$ pari?) {$(x\ n)\,(x = x^2,\ n = n/2)\,(x^n)$}
{$(x\ n)\,(x\ x,\ n-1)\,(x^n)$} `ifelse`} `ifelse`

Analisi operazionale:

	$x\ n$
`U 0 eq`	$x\ n\ n =?$
`{`	{ $x\ n$
`T T 1}`	1 }
`{`	{ $x\ n$
`U Alg.pari`	$x\ n$, n pari?
`{`	{ $x\ n$
`S U mul`	$n\ x^2$
`S 2 idiv`	$x^2\ n/2$
`Alg.potenza`	x^n }
`{`	{ $x\ n$
`PU R3`	$x\ x\ n$
`1 sub`	$x\ x,\ n-1$
`Alg.potenza`	$x\ x^{n-1}$
`mul}`	x^n }
`ifelse}`	x^n
`ifelse`	x^n

La nuova funzione per la potenza è quindi

```
/Alg.potenza {U 0 eq {T T 1}
  {U Alg.pari {S U mul S 2 idiv Alg.potenza}
  {PU R3 1 sub Alg.potenza mul} ifelse} ifelse} def
```

Anche l'analisi operazionale della funzione `Vett.anteponizeri` poteva essere preceduta da un programma analitico:

$(a\ m)\,(a\ m,\ n = |a|)\,(a\ m,\ i = m-n)$
$(a\ i,\ b = [0 \ldots 0])\,(b\ b\ i\ a)\,(c : \texttt{putinterval})$

Definiamo infine un altro algoritmo particolarmente semplice per lo schema di Horner:

$(a\ x)\,(b = 0,\ x\ a)$
{ $(b\ x\ a_i)\,(x\ a_i,\ b = bx)\,(b = b + a_i,\ x)$ } `forall`
$(b\ x)\,(b)$

Dopo un'appropriata analisi operazionale (esercizio 11) si ottiene l'eseguibile

```
/Alg.horner {0 R3 S {RC3 TU mul add S} forall T} def
```

con l'abbreviazione `TU` (terzultimo) per `2 index`.

Somma e prodotto degli elementi di un vettore

Molto facili sono le funzioni per la somma e il prodotto di tutti gli elementi di un vettore. Per la somma possiamo usare il programma analitico

$(a)\,(s = 0,\ a)$ { $(s\ a_i)\,(s + a_i)$ } `forall` (s)

a cui corrispondono l'analisi operazionale

	a
`0 S`	$s = 0,\ a$
`{`	{ $s\ a_i$
`add`	$s + a_i$ }
`forall`	s

e l'eseguibile

```
/Vett.somma {0 S {add} forall} def
```

Similmente per il prodotto abbiamo

```
/Vett.prod {1 S {mul} forall} def
```

Per il prodotto il valore iniziale è 1 e naturalmente bisogna usare `mul` al posto di `add`.

Il teorema di convoluzione per i numeri binomiali

Consideriamo due polinomi

$$f = a_0 + a_1 x + \ldots + a_n x^n$$

$$g = b_0 + b_1 x + \ldots + a_m x^m$$

Il loro prodotto fg lo si ottiene svolgendo formalmente il prodotto secondo le regole comuni dell'aritmetica e raccogliendo i termini che corrispondono alla stessa potenza di x. Ad esempio

$$\begin{aligned}(a_0 + a_1 x + a_2 x^2)(b_0 + b_1 x + b_2 x^2) = \\ a_0 b_0 + (a_0 b_1 + a_1 b_0)x + (a_0 b_2 + a_1 b_1 + a_2 b_0)x^2 \\ +(a_1 b_2 + a_2 b_1)x^3 + a_2 b_2 x^4\end{aligned}$$

Si vede che nel caso generale si ha

$$fg = c_0 + c_1 x + \ldots + c_{n+m} x^{n+m} \quad \text{con} \quad c_k = \sum_{i=0}^{k} a_i b_{k-i}.$$

Questa formula è spesso detta formula di *convoluzione*, perché è un analogo discreto della convoluzione dell'analisi armonica (o analisi di Fourier) in cui la convoluzione di due *funzioni* f e g è definita da

$$(f * g)(s) := \int f(t) g(s-t)\, dt$$

È chiaro anche che coefficienti uguali a zero non modificano le somme con cui sono espressi i c_k e quindi la regola non cambia se $a_n = 0$ oppure $b_m = 0$.

Teorema 16.1. *Per ogni* $k = 0, \ldots, n+m$ *vale*

$$\sum_{i=0}^{k} \binom{n}{i} \binom{m}{k-i} = \binom{n+m}{k}$$

Dimostrazione. Applichiamo la regola per il prodotto ai polinomi

$$f = (1+x)^n \quad \text{e} \quad g = (1+x)^m$$

Per il teorema binomiale (teorema 13.13) abbiamo

$$f = \sum_{i=0}^{n} \binom{n}{i} x^i \quad e \quad g = \sum_{j=0}^{m} \binom{m}{j} x^j$$

e quindi $\quad fg = \sum_{k=0}^{n+m} c_k x^k \quad$ con $\quad c_k = \sum_{i=0}^{k} \binom{n}{i} \binom{m}{k-i}$.

D'altra parte

$$fg = (1+x)^n (1+x)^m = (1+x)^{n+m} = \sum_{k=0}^{n+m} \binom{n+m}{k} x^k$$

Siccome due polinomi sono uguali se e solo se i coefficienti delle singole potenze di x sono uguali, otteniamo l'enunciato.

Questo risultato prende il nome di teorema di convoluzione per i numeri binomiali; talvolta è anche detto formula di convoluzione di Vandermonde (pubblicata da questo autore nel 1772), ma era noto già nel 1303 ai matematici cinesi (Needham, citato in Knuth, pag. 58). Una dimostrazione combinatoria si trova a pagina 11 del libro di Cerasoli/Eugeni/Protasi.

M. Cerasoli/F. Eugeni/M. Protasi: Elementi di matematica discreta. Zanichelli 1988. Un ottimo testo, di facile lettura, molto completo e moderno nella presentazione. Marco Protasi è morto prematuramente nel 1998 a 48 anni.

D. Knuth: The art of computer programming I. Addison-Wesley 1973. Questo famoso classico dell'informatica combina un'esposizione dei principi elementari della programmazione con le basi computazionali della matematica discreta.

J. Needham: Science and civilisation in China. 17 volumi. Cambridge UP 1954. Joseph Needham (1900-1995) è considerato il più grande storico inglese della Cina. La sua opera è continuata nel Needham Research Institute (*www.nri.org.uk*) a Cambridge in Inghilterra.

Il calcolo combinatorio ha una lunga tradizione in Italia. Tra i suoi esponenti di spicco e famosi nel mondo indichiamo Gian-Carlo Rota (1932-1999, funzioni generatrici e geometrie combinatorie astratte), Giuseppe Tallini (1930-1995, geometrie finite e teoria dei codici), Adriano Barlotti (geometrie finite), Aldo De Luca (linguaggi formali e combinatoria delle parole).

Abbreviazioni per operatori elementari

Useremo spesso le seguenti abbreviazioni, di cui diamo prima le corrispondenze analitiche e le interpretazioni verbali:

`U`	(a) (a a)	ultimo
`PU`	(a b) (a b a)	penultimo
`TU`	(a b c) (a b c a)	terzultimo
`QU`	(a b c d) (a b c d a)	quartultimo
`T`	(a) ()	togli
`R3`	(a b c) (c a b)	rotazione `3 1 roll`
`R4`	(a b c d) (d a b c)	rotazione `4 1 roll`
`R42`	(a b c d) (c d a b)	rotazione `4 2 roll`
`R5`	(a b c d e) (e a b c d)	rotazione `5 1 roll`
`RC3`	(a b c) (b c a)	rotazione contraria `3 -1 roll`
`RC4`	(a b c d) (b c d a)	rotazione contraria `4 -1 roll`
`RC5`	(a b c d e) (b c d e a)	rotazione contraria `5 -1 roll`
`S`	(a b) (b a)	scambio
`SPT`	(a b c) (b a c)	scambio penultimo terzultimo
`SUT`	(a b c) (c b a)	scambio ultimo terzultimo

Le operazioni `S`, `SUT`, `SPT`, `R3` e `RC3` formano con l'identità l'insieme delle permutazioni di tre elementi.

Gli operatori elencati sono così programmati:

```
/U {dup} def

/PU {1 index} def

/TU {2 index} def

/QU {3 index} def

/T {pop} def

/R3 {3 1 roll} def

/R4 {4 1 roll} def

/R42 {4 2 roll} def

/R5 {5 1 roll} def

/RC3 {3 -1 roll} def

/RC4 {4 -1 roll} def

/RC5 {5 -1 roll} def

/S {exch} def

/SPT {RC3 S} def

/SUT {R3 S} def
```

Esercizi per gli scritti

7. Giustificare la regola della nota 13.16.

8. Con `print` ottenere l'output su terminale dei binomiali $\binom{11}{k}$ tutti su una riga, separati da uno spazio.

9. Programma analitico, analisi operazionale e nuovo programma eseguibile per `Alg.rapp2`.

10. Programma analitico su cui si basa l'analisi operazionale per `Vett.concat` a pagina 9.

11. Analisi operazionale per `Alg.horner` a pagina 15.

12. Visualizzare sul terminale le radici quadrate dei numeri naturali da 0 a 12. Usare prima `forall`, poi `for`.

13. Introducendo un dizionario e usando `repeat` visualizzare ancora sul terminale le radici quadrate dei numeri naturali da 0 a 12.

14. Programmare la funzione di due variabili f definita da $f(x,y) = xy + \log\sqrt{x^2+y^2}$.

ALGORITMI E STRUTTURE DI DATI

Corso di laurea in matematica | Anno accademico 2005/06 | Numero 3

Come si crea una libreria

Per poter conservare le funzioni che abbiamo preparato creiamo una raccolta (*libreria*) con la tecnica seguente. Il file *Alfa.ps* contiene soltanto l'informazione sulla *BoundingBox* (necessaria per *convert*, ma non per l'uso diretto di Ghostscript, come sappiamo), l'impostazione dell'insieme di caratteri e qualche altra istruzione preliminare come parte più o meno fissa, a cui aggiungiamo di volta in volta i comandi che vogliamo sperimentare. La parte fissa potrebbe essere questa:

```
%%BoundingBox: 0 0 800 400
(Alfa.h) run

2 2 scale
10 Font.times /s 60 string def 10 180 moveto
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Il comando `10 Font.times` imposta l'insieme di caratteri e verrà discusso più avanti. Con `2 2 scale` il disegno viene ingrandito del fattore 2; ciò ci permette di usare per la visualizzazione sullo schermo all'incirca la stessa dimensione (qui 10) per i caratteri che useremo in un documento da stampare. Viene inoltre creata una stringa `s` per l'uso con il comando `cvs`, mentre il `moveto` permette di poter eseguire direttamente il comando `show`.

Ai fini della libreria che vogliamo creare è essenziale solo l'istruzione `(Alfa.h) run`. L'argomento di `run` è il nome di un file il cui contenuto viene eseguito come se questo contenuto fosse scritto in quel punto del nostro file sorgente. Se ad esempio il file *beta* contiene solo il comando `4 add` e se in *Alfa.ps* aggiungiamo

```
7 (beta) run =
3 5 {(beta) run} repeat =
```

l'output sarà

```
11
23
```

`run` corrisponde quindi abbastanza precisamente al `# include` del C. Con `(Alfa.h) run` lo usiamo per poter scrivere le inclusioni dei files di libreria in un file separato *Alfa.h* che modifichiamo soltanto quando aggiungiamo o togliamo un file nella nostra raccolta. *Alfa.h* contiene infatti i seguenti comandi:

```
/Concat {dup length 2 index length dup 5 1 roll
  add string dup dup 6 -1 roll 5 -1 roll putinterval
  3 1 roll exch 0 exch putinterval} def

[(Alg) (File) (Font) (Stack) (Str) (Vett)]
  {(Moduli/) exch Concat run} forall
```

Siccome all'inizio la libreria non è ancora caricata e quindi non possiamo utilizzare la funzione `Str.concat` della libreria, la riscriviamo nella forma vista a pagina 5. Dobbiamo poi creare una cartella *Moduli* che conterrà i files della libreria. Il nome relativo della libreria per i vettori è allora (sotto Linux) *Moduli/Vett* e con questo nome viene eseguito il `run`. Sotto Windows bisogna forse usare *Moduli*\ o *Moduli*\\ invece di *Moduli*/.

Il comando `run` può essere usato anche nelle istruzioni `\special` del Latex. Si potrebbe usare anche la procedura `filenameforall` che verrà trattata più avanti. Non avremo più bisogno di `Concat` allora.

PostScript è stato concepito come linguaggio per la grafica stampata e presenta quindi limitazioni piuttosto restrittive nelle applicazioni al calcolo scientifico. Il comando `1 1 50000 {} for` è ancora corretto, ma `1 1 60000 {} for` supera i limiti; un vettore non può avere più di 65535 elementi; lo stack dei dizionari è piuttosto limitato; ricorsioni possono diventare molto lente perché le operazioni sullo stack devono avvenire in un certo senso a corsia unica. Calcoli numerici di piccole dimensioni compaiono però spesso anche nella grafica su carta ed è molto utile che sia possibile saperli trattare direttamente in PostScript. Per il calcolo scientifico con Forth si può consultare il sito Internet indicato a destra.

In questo numero

Font e Font.times

Il comando `Font.times` si trova nel file *Font* della libreria e, per il momento, contiene le seguenti istruzioni:

```
% Font

% n nomefont Font -.
/Font {findfont S scalefont setfont} def

% n Font.dingbats -.
/Font.dingbats {/ZapfDingbats Font} def

% n Font.times -.
/Font.times {/Times-Roman Font} def
```

ZapfDingbats è un unsieme di caratteri decorativi creato da Hermann Zapf, un famoso disegnatore di caratteri, nato nel 1918 e sposato con la tipografa e legatrice Gudrun von Hesse (nata 1918). Fu uno dei primi a comprendere l'importanza del computer per la tipografia, ma in Germania venne praticamente deriso e costretto a realizzare i suoi progetti in America.

Con

```
30 0 moveto 15 Font.dingbats
[90 91 92 93 94 95 96] Str.davettore show
```

otteniamo

Il carattere con codice ASCII 32 è lo spazio anche in questo insieme di caratteri, perciò con

```
30 0 moveto 20 Font.dingbats
[96 32 46 32 34 32 87 32 109 32 114 32 219 32 100]
Str.davettore show
```

otteniamo

de.wikipedia.org/wiki/Hermann_Zapf.
www.linotype.com/freedownload/howtouse/ZapfBiography.pdf. Autobiografia di Hermann Zapf.
www.hermannzapf.de/.

---

www.taygeta.com/fsl/sciforth.html. Una raccolta di funzioni per il calcolo numerico con Forth.

Minimi e massimi

Per calcolare il minimo di due numeri a e b usiamo il seguente algoritmo:

	$a\ b$
`2 copy`	$a\ b\ a\ b$
`gt`	$a\ b\ (a > b)$
`{`	$\{\ a\ b$
`TPU}`	$m = b\}$
`{`	$\{\ a\ b$
`T}`	$m = a\ \}$
`ifelse`	m

Definiamo perciò

```
/Alg.min {2 copy gt {TPU} {T} ifelse} def
```

Sostituendo `gt` con `lt` otteniamo il massimo:

```
/Alg.max {2 copy lt {TPU} {T} ifelse} def
```

È facile adesso trovare il minimo di un vettore numerico. Estraiamo prima il primo elemento x del vettore, poi, tramite un `forall`, sostituiamo in ogni passo x con $\min(x, a_i)$.

	a
`U 0 get`	$a\ a_0$
`S`	$a_0\ a$
`{`	$\{\ x = a_0, a_i$
`Alg.min}`	$x = \min(x, a_i)\ \}$
`forall`	x

Il programma eseguibile diventa quindi

```
/Alg.vettmin {U 0 get S {Alg.min} forall} def
```

e similmente per il massimo:

```
/Alg.vettmax {U 0 get S {Alg.max} forall} def
```

I numeri di Fibonacci

La successione dei numeri di Fibonacci è definita dalla ricorrenza

$$F_0 = F_1 = 1,\ F_n = F_{n-1} + F_{n-2} \quad \text{per } n \geq 2.$$

È facilissimo calcolarli con PostScript:
Dal programma analitico

$(n)\ (a = 1, b = 1,\ 2\ 1\ n)$
$\{(a\ b\ i)\ (a\ b : i \text{ non serve})$
$(b\ a\ b)\ (a = b, b = a + b)\}$ `for` $(a\ b)\ (b)$

otteniamo il programma eseguibile

```
/Alg.fib {1 1 RC3 2 1 RC3 {T U R3 add} for TPU} def
```

Eseguire da soli l'analisi operazionale (esercizio 19). Esaminare attentamente il programma per convincersi che la funzione calcola correttamente F_n anche per $n = 0, 1$. Prove:

```
0 15 Alg.intervallo {Alg.fib} map ==
% output: [1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987]

40 Alg.fib pstack
% output: 165580141
```

Eliminazione e inserimento all'interno dello stack

Abbiamo già introdotto le funzioni `T`, `TPU` e `TTU` per togliere gli ultimi tre elementi dello stack. Per togliere l'n-esimo elemento, cominciando sempre a contare da zero e verso sinistra, usiamo la funzione `Tn` definita dal seguente semplice algoritmo:

	$a_n\ a_{n-1} \ldots a_1\ a_0\ n$
`1 add`	$a_n\ a_{n-1} \ldots a_1\ a_0, n+1$
`-1 roll`	$a_{n-1} \ldots a_1\ a_0\ a_n$
`T`	$a_{n-1} \ldots a_1\ a_0$

per cui

```
/Tn {1 add -1 roll T} def
```

Ancora più semplice è l'inserimento di un elemento x nella posizione n dello stack. Ciò non è altro che l'effetto dell'operazione `n+1 1 roll`:

	$a_{n-1} \ldots a_1\ a_0\ x\ n$
`1 add`	$a_{n-1} \ldots a_1\ a_0\ x, n+1$
`1 roll`	$x\ a_{n-1} \ldots a_1\ a_0$

per cui definiamo

```
/INS {1 add 1 roll} def
```

Inversione

Con un `for` su `1 roll` otteniamo una funzione per l'inversione degli ultimi n elementi dello stack. Consideriamo ad esempio la situazione per $n = 5$:

	$a_4\ a_3\ a_2\ a_1\ a_0$
`5 1 roll`	$a_0\ a_4\ a_3\ a_2\ a_1$
`4 1 roll`	$a_0\ a_1\ a_4\ a_3\ a_2$
`3 1 roll`	$a_0\ a_1\ a_2\ a_4\ a_3$
`2 1 roll`	$a_0\ a_1\ a_2\ a_3\ a_4$

Vediamo che il `for` deve essere percorso in modo decrescente da n a 2, cosicché la funzione desiderata diventa

```
/INV {-1 2 {1 roll} for} def
```

Per invertire gli elementi di un vettore a possiamo adesso utilizzare la seguente procedura:

	a
`[`	$a\ [$
`S`	$[\ a$
`aload`	$[\ a_1 \ldots a_n\ a$
`pop`	$[\ a_1 \ldots a_n$
`counttomark`	$[\ a_1 \ldots a_n\ n$
`INV`	$[\ a_n \ldots a_1$
`]`	$[a_1 \ldots a_n]$

L'eseguibile è

```
/Vett.inv {[S aload pop counttomark INV ]} def
```

Vett.anteponi

Definiamo una funzione che aggiunge un elemento x all'inizio di un vettore. Potremmo trasformare x in un vettore e usare `Vett.concat`, ma un algoritmo apposito è più semplice e più veloce.

	$x\ a$
`[`	$x\ a\ [$
`R3`	$[\ x\ a$
`aload T`	$[\ x\ a_1 \ldots a_n$
`]`	$[x\ a_1 \ldots a_n]$

Definiamo così

```
/Vett.anteponi {[ R3 aload T ]} def
```

Prova:

```
0 [1 2 3 4 5] Vett.anteponi ==
% output: [0 1 2 3 4 5]
```

Abbiamo usato == perché = non riconosce vettori e stamperebbe solo `--nostringval--`.

Anche qui, ai fini dell'esame, memorizzare solo un programma analitico che corrisponde all'algoritmo:

$(x\ a)\ ([\ x\ a)\ ([\ x\ a_1 \ldots a_n\ )\ ([x\ a_1 \ldots a_n])$

La funzione C generalizza index

Ciò significa che, se la parte finale dello stack è ad esempio $x_4\ x_3\ x_2\ x_1\ x_0$, con `[3 1 4 1 1 0] C` vengono aggiunti allo stack gli elementi $x_3\ x_1\ x_4\ x_1\ x_1\ x_0$:

`[3 1 4 1 1 0] C`	$x_4\ x_3\ x_2\ x_1\ x_0$
	$x_4\ x_3\ x_2\ x_1\ x_0\ \ x_3\ x_1\ x_4\ x_1\ x_1\ x_0$

Osserviamo prima che non possiamo definire

```
/C {{index} forall} def % (*)
```

perché ogni volta, essendo stato aggiunto un elemento, bisogna aumentare l'indice che si vuole usare. Con la definizione (*) avremmo infatti

`[1 4 2] C`	$x_4\ x_3\ x_2\ x_1\ x_0$
	$x_4\ x_3\ x_2\ x_1\ x_0\ x_1\ x_3\ x_0$

Sono stati quindi aggiunti gli elementi $x_1\ x_3\ x_0$ e non $x_1\ x_4\ x_2$. Vediamo che dobbiamo aumentare gli indici: il primo indice di 0, il secondo di 1, il terzo di 2, ecc. e quindi usare `[1 5 4]`. Infatti

`[1 5 4] {index} forall`	$x_4\ x_3\ x_2\ x_1\ x_0$
	$x_4\ x_3\ x_2\ x_1\ x_0\ x_1\ x_4\ x_2$

è il risultato corretto. Si noti che, come d'uso in PostScript, anche noi numeriamo gli elementi dello stack da destra verso sinistra, iniziando da 0.

Abbiamo così bisogno di una funzione `Cprep` che trasforma un vettore $[i_1 \dots i_n]$ in $[i_1, i_2+1, \dots, i_n+n-1]$.

Possiamo eseguire direttamente l'analisi operazionale, in cui $i = [i_1 \dots i_n]$ denota un vettore di indici.

	i
`[ S`	$[\ i$
`aload length`	$[\ i_1 \dots i_n\ n$
`1 sub`	$[\ i_1 \dots i_n, m = n-1$
`-1 0`	$[\ i_1 \dots i_n\ m\ {-1}\ 0$
`{`	$\{\ [\ i_1 \dots i_{n-1}\ i_n\ k$
`add`	$[\ i_1 \dots i_{n-1}, i'_n = i_n + k$
`counttomark`	$[\ i_1 \dots i_{n-1}\ i'_n\ n$
`1 roll}`	$[\ i'_n\ i_1 \dots i_{n-1}\ \}$
`for`	$[\ i'_1 \dots i'_n$
`]`	$[i'_1 \dots i'_n]$

Il programma eseguibile è quindi

```
/Cprep {[ S aload length 1 sub -1 0
  {add counttomark 1 roll} for ]} def
```

Prova per `Cprep`:

```
[0 0 0 5 2] Cprep {s cvs print ( ) print} forall ()=
% output: 0 1 2 8 6
```

Adesso possiamo definire

```
/C {Cprep {index} forall} def
```

Prova:

```
5 4 3 2 1 0 [2 2 5 1] C
9 -1 0 {index s cvs print ( ) print} for ()=
% output: 5 4 3 2 1 0 2 2 5 1
```

Vett.filtro

La funzione che creiamo adesso è una delle più versatili della nostra raccolta. Si tratta di una funzione *filtro* che, dati un vettore a una funzione a valori booleani f, crea un vettore b che contiene (nello stesso ordine in cui sono elencati in a) tutti gli elementi di a per cui f assume il valore vero.

Si noti che in PostScript, a differenza da molti altri linguaggi, non si possono usare numeri, ad esempio 0 e 1, come valori booleani (valori di verità), ma solo `true` e `false` (cfr. pagina 7), spesso indirettamente ad esempio come risultati di operazioni logiche.

La funzione f deve essere inoltre una funzione di un argomento che restituisce un solo valore (e toglie quindi l'operando dallo stack e lo sostituisce con un altro oggetto). Sono perciò errate le istruzioni

```
[0 2 8 4 3] {true} Vett.filtro

[0 2 8 4 3] {T 1} Vett.filtro
```

la prima, perché la funzione usata lascia il suo argomento sullo stack (ed è quindi, nella filosofia di PostScript, una funzione a due valori), la seconda perché 1 non è un valore booleano. È invece corretta l'istruzione

```
[0 2 8 4 3] {T true} Vett.filtro
```

banale, perché sostituisce il vettore dato con una copia identica.

Il programma per la funzione è sorprendentemente semplice, perché possiamo usare di nuovo la tecnica della marcatura. Nel ciclo `forall` dell'analisi operazionale che segue, $t_1 \dots t_m$ sono gli elementi trovati nei precedenti passaggi del ciclo.

	$a\ f$
`[`	$a\ f\ [$
`SUT`	$[\ f\ a$
`{`	$\{\ [\ t_1 \dots t_m\ f\ a_i$
`U TU`	$[\ t_1 \dots t_m\ f\ a_i\ a_i\ f$
`exec`	$[\ t_1 \dots t_m\ f\ a_i\ f(a_i)$
`{`	$\{[\ t_1 \dots t_m\ f\ a_i$
`S}`	$[\ t_1 \dots t_m\ a_i\ f\ \}$
`{`	$\{\ [\ t_1 \dots t_m\ f\ a_i$
`T}`	$[\ t_1 \dots t_m\ f\ \}$
`ifelse}`	$[\ t_1 \dots f\ \}$
`forall`	$[\ t_1 \dots f$
`T`	$[\ t_1 \dots$
`]`	$[t_1 \dots]$

da cui

```
/Vett.filtro {[ SUT {U TU exec {S} {T} ifelse} forall T ]} def
```

Questa funzione ha moltissime applicazioni.

Filtri per stringhe

Possiamo applicare `Vett.filtri` anche a stringhe. Ciò ci permetterà ad esempio di ordinare alfabeticamente un elenco di nomi, come vedremo fra poco.

Cerchiamo prima tutti quei nomi da un vettore di stringhe non vuote la cui prima lettera è *F*. Se per trovare la prima lettera usiamo `get`, dobbiamo ricordarci che i caratteri estratti vengono rappresentati dal loro codice ASCII (pagina 5) che nel caso di *F* è 70.

Ad esempio:

```
[(Ferrara) (Pisa) (Padova) (Firenze) (Roma)]
  {0 get 70 eq} Vett.filtro ==
% output: [(Ferrara) (Firenze)]
```

Si potrebbe anche determinare il numero ASCII automaticamente:

```
/AsciiF (F) 0 get def
[(Ferrara) (Pisa) (Padova) (Firenze) (Roma)]
  {0 get AsciiF eq} Vett.filtro ==
% output: [(Ferrara) (Firenze)]
```

Siccome `eq` può essere usato anche per confrontare stringhe, si può anche evitare il codice ASCII mediante l'utilizzo di `getinterval`:

```
[(Ferrara) (Pisa) (Padova) (Firenze) (Roma)]
  {0 1 getinterval (F) eq} Vett.filtro ==
% output: [(Ferrara) (Firenze)]
```

Anche in questo caso però il vettore non deve contenere stringhe vuote.

Con la stessa tecnica possiamo trovare tutte le stringhe che iniziano con *Pa*:

```
[(Palermo) (Pisa) (Padova) (Parma) (Roma)]
  {0 2 getinterval (Pa) eq} Vett.filtro ==
% output: [(Palermo) (Padova) (Parma)]
```

Altre applicazioni della funzione filtro

Diamo alcuni esempi elementari per l'applicazione di `Vett.filtro`.

Dato un vettore di numeri, troviamo tutti gli elementi positivi del vettore.

```
[2 1 -3 2.55 6 -4 3.1 -5 -2 0] {0 gt} Vett.filtro ==
% output: [2 1 2.55 6 3.1]
```

Troviamo tutti i numeri pari di un vettore di numeri interi:

```
[1 5 2 0 0 1 4 19 21 12 6 9] {Alg.pari} Vett.filtro ==
% output: [2 0 0 4 12 6]
```

Troviamo tutti gli elementi di un vettore di numeri naturali che danno resto 1 nella divisione per 4:

```
[3 0 5 6 7 13 8 17 25 8] {4 mod 1 eq} Vett.filtro ==
% output: [5 13 17 25]
```

Qui bisogna stare attenti che, come in altri linguaggi di programmazione, ad esempio il C, la funzione `mod` di PostScript dà un risultato corretto solo per numeri naturali.

Da un vettore di coppie (x, y) di numeri reali troviamo quelle per cui $x^2 + y^2 < 1$:

```
[[1 0.3] [0.5 0.4] [0.1 0] [-0.04 0.8] [0.7 0.9]]
  {aload T 2 copy SPT mul R3 mul add 1 lt} Vett.filtro ==
% output: [[0.5 0.4] [0.1 0] [-0.04 0.8]]
```

Controllare con un'analisi operazionale l'algoritmo. In questo caso naturalmente conviene definire con

```
/q {U mul} def
```

una funzione per il quadrato e usare

```
[[1 0.3] [0.5 0.4] [0.1 0] [-0.04 0.8] [0.7 0.9]]
  {aload T q S q add 1 lt} Vett.filtro ==
```

Controllare anche questo algoritmo.

Cerchiamo gli elementi x di un vettore numerico che soddisfano la disuguaglianza $4 < x < 10$:

```
[8 1 3 4 5 6 15 25 0 4]
{U 10 lt S 4 gt and} Vett.filtro ==
% output: [8 5 6]
```

Controllare l'algoritmo.

Intervalli di numeri interi

In molti compiti di programmazione, ma anche nella matematica pura, sono importanti intervalli finiti di numeri interi, cioè successioni finite della forma

$$\{a, a+1, \ldots, b-1, b\} = \{n \in \mathbb{Z} \mid a \le n \le b\}$$

È molto facile crearli in PostScript. Osserviamo prima che l'istruzione

```
[ a 1 b {} for]
```

rappresenta già un tale intervallo, come possiamo verificare con

```
[ 2 1 9 {} for ] pstack
% output: [2 3 4 5 6 7 8 9]
```

Dobbiamo soltanto conoscere a e b. Nella funzione che vogliamo creare questi valori devono essere gli ultimi due elementi dello stack, cosicché dopo la semplice analisi operazionale

	$a\ b$
`[ 1`	$a\ b$ `[ 1`
`R42`	`[` $1\ a\ b$
`SPT`	`[` $a\ 1\ b$
`{}`	`[` $a\ 1\ b\ \{\}$
`for`	`[` $a, a+1, \ldots, b$
`]`	$[a, a+1, \ldots, b]$

otteniamo la funzione

```
/Alg.intervallo {[ 1 R42 SPT {} for ]} def
```

Prova:

```
5 18 Alg.intervallo pstack
% output: [5 6 7 8 9 10 11 12 13 14 15 16 17 18]
```

Il crivello di Eratostene

Definizione 20.1. Siano $a, b \in \mathbb{Z}$.

Diciamo che a divide b e scriviamo $a|b$, se esiste $k \in \mathbb{Z}$ tale che $b = ka$.

Quindi $a|b$ se e solo se b è un multiplo di a, cioè se e solo se $b \in \mathbb{Z}a$. È chiaro che allora anche $\mathbb{Z}b \subset \mathbb{Z}a$ e viceversa, quindi $a|b$ se e solo se $\mathbb{Z}b \subset \mathbb{Z}a$.

Questa riduzione della relazione di divisibilità a una relazione insiemistica è molto importante e conduce alla *teoria degli ideali*, un ramo centrale dell'algebra commutativa.

Definizione 20.2. Un numero naturale p si chiama *primo*, se $p \ge 2$ e se per $d \in \mathbb{N}$ con $d|p$ si ha $d \in \{1, p\}$.

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*.

L'idea è questa: All'inizio calcoliamo $r = \sqrt{n}$ e creiamo il vettore $a = (2, 3, 4, \ldots, n)$. Poi eseguiamo un ciclo (riconoscibile dal comando `loop` che segue le istruzioni): in ogni passo del ciclo togliamo il primo elemento p, necessariamente primo, da a e lo aggiungiamo ai primi $p_1, \ldots, p_i$ già trovati. Se $p > r$, ci fermiamo (per l'oss. 21.1), dopo aver aggiunto anche i rimanenti elementi di a. Altrimenti, utilizzando `Vett.filtro`, eliminiamo tutti i multipli di p da a.

Nel programma analitico abbiamo bisogno di un dizionario a una voce in cui segniamo ogni volta il valore di p. Usiamo la tecnica della marcatura.

(n) (n `1 dict begin`) ([$n, r = \sqrt{n}$) ([$r\ a = [2\ 3\ 4 \ldots n]$)
{ ([$p_1 \ldots p_i\ r\ a$: dopo l'i-esimo passo)
([$p_1 \ldots p_i\ r\ p\ b$, $(p > r)$)
{ ([$p_1 \ldots p_i\ r\ p\ b$) ([$p_1 \ldots p_i\ r\ b$) `exit` } { ([$p_1 \ldots p_i\ r\ p\ b$)
([$p_1 \ldots p_{i+1}\ r\ b$ `p QU def {p mod 0 ne} Vett.filtro`)
([$p_1 \ldots p_{i+1}\ r$, $b = b'$) `} ifelse}`
`loop` ([$p_1 \ldots p_{i+1}\ r\ b$)
([$p_1 \ldots p_{i+1}\ b$)
([$p_1 \ldots p_{i+1} b_1 \ldots b_m$] `end`)

Dall'analisi operazionale (esercizio 21) otteniamo il programma eseguibile:

```
/Alg.era {1 dict begin [ S U sqrt S 2 S Alg.intervallo
  {Vett.spezza PU QU gt {SPT exit}
  {SPT /p QU def {p mod 0 ne} Vett.filtro} ifelse} loop
  TPU aload T ] end} def
```

Per calcolare i numeri primi minori di 100 usiamo

```
100 Alg.era ==
% output, che riportiamo su due righe:
% [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]
```

L'esecuzione è sufficientemente veloce anche per numeri più grandi. Il programma funziona ancora per $n = 40000$, vengono invece superati i limiti dello stack per $n = 50000$.

La funzione $\pi(n)$

Osservazione 21.1. *$n \in \mathbb{N} + 2$ sia un numero naturale ≥ 2 fissato. m sia un altro numero naturale con $2 \leq m \leq n$ che non sia primo. Allora esiste $a \in \mathbb{N}$ con $2 \leq a \leq \sqrt{n}$ tale che $a|m$.*

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

Dimostrazione. Per ipotesi esistono $a, b \in \mathbb{N}$ tali che $ab = n$ con $a, b \geq 2$. Se si avesse $a > \sqrt{n}$ e $b > \sqrt{n}$, allora $ab > n$, una contraddizione.

Lemma 21.2. *Sia $n \in \mathbb{N} + 2$. Allora il più piccolo divisore maggiore di 1 di n è primo. In particolare vediamo che esiste sempre un primo che divide n.*

Dimostrazione. È intuitivamente evidente e può essere dimostrato per induzione che ogni sottoinsieme non vuoto di $\mathbb{N}$ possiede un elemento più piccolo. Siccome n stesso è un divisore maggiore di 1 di n, vediamo che esiste veramente il più piccolo divisore maggiore di 1 di n; lo chiamiamo p.

Se p non fosse primo, avrebbe a sua volta un divisore q maggiore di 1 e diverso da p e quindi più piccolo di p. q sarebbe anche un divisore di n, in contraddizione alla minimalità di p.

Osservazione 21.3. *Siano $a, b, d \in \mathbb{Z}$ tali che $d|a$ e $d|b$. Allora $d|a + b$ e $d|a - b$.*

Dimostrazione. Esercizio 22.

Teorema 21.4. *Esiste un numero infinito di numeri primi.*

Dimostrazione. Questa dimostrazione risale ad Euclide!

Assumiamo che $p_1, \ldots, p_k$ siano tutti i numeri primi esistenti. Sia $a := p_1 \cdots p_k + 1$. Allora $a > 1$ e dal lemma 21.2 sappiamo che esiste un primo p che divide a. Per ipotesi allora p deve essere uno dei p_j. Per l'osservazione 21.3 ciò implica $p_j|1$, una contraddizione.

Definizione 21.5. Per $n \in \mathbb{N}$ sia $\pi(n)$ il numero dei primi $p \leq n$.

Teorema 21.6. $\lim_{n \longrightarrow \infty} \dfrac{\pi(n)}{n/\log n} = 1.$

Dimostrazione. Questo enunciato, il *teorema dei numeri primi*, ha richiesto più di cento anni per la sua dimostrazione, dopo che era stato congetturato dal giovane Gauß.

Nota 21.7. Il teorema dei numeri primi significa che il rapporto $\dfrac{\pi(n)}{n}$ tra i numeri primi $\leq n$ e tutti i numeri naturali $\leq n$ è asintoticamente all'incirca uguale a $\dfrac{1}{\log n}$. Il logaritmo è quello in base e, che si distingue dal logaritmo in base 10 per il fattore $\log 10 \sim 2.3$.

Usiamo PostScript per sperimentare queste stime:

```
[100 1000 10000 100000 1000000 1000000000] {ln} Vett.map ==
% output arrotondato:
% [4.6 6.9 9.2 11.5 13.8 20.7]
```

Ciò significa (se assumiamo una validità della stima anche per n relativamente piccolo) che all'incirca ogni quarto o ogni quinto numero tra i primi cento è primo, ogni settimo tra i primi mille, uno su nove tra i primi diecimila, uno su 11.5 tra i primi centomila, uno su 14 nel primo milione, uno su 21 nel primo miliardo.

I numeri primi diventano quindi sempre meno densi, ma la densità decresce molto lentamente, come il reciproco del logaritmo appunto.

Per 100, 1000 e 10000 possiamo usare `Alg.era` per confrontare il valore vero di $\dfrac{n}{\pi(n)}$ con quello stimato:

```
100 Alg.era length 100 S div ==
% output arrotondato: 4.0 invece di 4.6

1000 Alg.era length 1000 S div ==
% output arrotondato: 5.95 invece di 6.9

10000 Alg.era length 10000 S div ==
% output arrotondato: 8.1 invece di 9.2
```

La corrispondenza non è perfetta, migliora però quando n diventa molto grande. Se vogliamo solo sapere quanti sono i numeri primi, usiamo

```
100 Alg.era length ==
% output: 25

1000 Alg.era length ==
% output: 168

10000 Alg.era length ==
% output: 1229
```

H. Cohen: A course in computational algebraic number theory. Springer 1993.

M. Drmota/R. Tichy: Sequences, discrepancies and applications. Springer 1997. La teoria dei numeri, la regina della matematica pura, ha trovato negli ultimi vent'anni applicazioni: la teoria dei numeri primi nella crittografia, la teoria dell'uniforme distribuzione, soprattutto in spazi ad alta dimensione, nella generazione di numeri casuali e quindi nella simulazione di fenomeni ad esempio della fisica delle particelle o della fluidodinamica, ma anche della matematica finanziaria.

L. Hua: Introduction to number theory. Springer 1982. Una delle migliori introduzioni alla teoria dei numeri.

H. Niederreiter: Random number generation and quasi-Monte Carlo methods. SIAM 1992.

H. Niederreiter/P. Shiue (ed.): Monte Carlo and quasi-Monte Carlo methods in scientific computing. Springer 1995.

H. Riesel: Prime numbers and computer methods for factorization. Birkhäuser 1985.

www.finanz.jku.at/. L'istituto di matematica finanziaria dell'università di Linz è guidato da Gerhard Larcher, un noto teorico dei numeri.

Esercizi per gli scritti

15. Scrivere due funzioni `TPU` e `TTU` che tolgono il penultimo e il terzultimo elemento dallo stack.

16. Eseguire l'analisi operazionale del programma analitico per la funzione `Vett.filtro` a pagina 19.

17. Dati i nomi *prog.ps*, *prog.png*, *lettera.ps*, *lettera.txt*, *imm1.gif*, *imm2.gif*, *imm2.ps*, trovare tutti quelli che terminano in *.ps*.

18. Scrivere una funzione `Intervallo` con un terzo argomento d, che genera successioni della forma $(a, a + d, a + 2d, \ldots, b)$ (b può essere raggiunto, ma non deve essere superato).

19. Analisi operazionale per `Alg.fib` (numeri di Fibonacci).

20. Scrivere funzioni `And`, `Or` e `Not` che hanno 0 e 1 come argomenti e come risultati. Provarle con

```
0 0 And == 0 1 And == 1 0 And == 1 1 And ==

0 0 Or == 0 1 Or == 1 0 Or == 1 1 Or ==

0 Not == 1 Not ==
```

21. Analisi operazionale per `Alg.era` (crivello di Eratostene).

22. Siano $a, b, d \in \mathbb{Z}$ tali che $d|a$ e $d|b$. Allora $d|a + b$ e $d|a - b$.

23. Trasformare

```
(((0 or 1) or (0 and not 1)) and (1 or not (1 and 0)))
    or ((0 and 1) and 1)
```

in formato RPN e calcolare il valore con pedine da scacchi.

24. Usare carte da gioco per simulare gli stati dello stack nell'esecuzione di

```
4 2 add 3 sub 2 add 2 mul 4 3 add sub
```

Usare il fante per la sottrazione, la dama per l'addizione, il re per la moltiplicazione.

Primo scritto venerdì 28 ottobre, ore 16.

ALGORITMI E STRUTTURE DI DATI

Corso di laurea in matematica | Anno accademico 2005/06 | Numero 4

Il re dei matematici

Carl Friedrich Gauß (1777-1855) è considerato il re dei matematici. La lettera ß alla fine del nome è una s tedesca antica; il nome (talvolta scritto Gauss) si pronuncia gaos, simile a caos, ma con la g invece della c e con la o molto breve e legata alla a in modo che le due vocali formino un dittongo. Nessun altro matematico ha creato tanti concetti profondi ancora oggi importanti nelle discipline matematiche più avanzate (teoria dei numeri, geometria differenziale e geodesia matematica, teoria degli errori e statistica, analisi complessa). Il ritratto lo mostra a ventisei anni.

È stato forse il primo a concepire le geometrie non euclidee, ha dato una semplice interpretazione dei numeri complessi come punti del piano reale con l'addizione vettoriale e la moltiplicazione

$$(a,b)\cdot(c,d)=(ac-bd,bc+ad)$$

e ha dimostrato il teorema fondamentale dell'algebra (che afferma che ogni polinomio con coefficienti complessi possiede, nell'ambito dei numeri complessi, una radice), ha introdotto la distribuzione gaussiana del calcolo delle probabilità, ha conseguito importanti scoperte nella teoria dell'elettromagnetismo; è stato direttore dell'osservatorio astronomico di Gottinga.

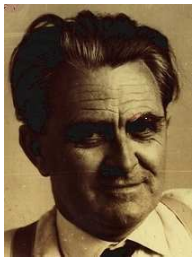
L'algoritmo di eliminazione era noto nel 1759 a Lagrange (1736-1813) e già 2000 anni fa in Cina; Gauß lo ha usato nel suo lavoro sui moti celesti del 1809, in cui descrisse il metodo dei minimi quadrati, una tecnica di approssimazione ancora oggi universalmente utilizzata.

Le basi di Gröbner

Se si prova ad imitare l'algoritmo di eliminazione nella soluzione di sistemi polinomiali di grado maggiore di uno, ad esempio di

$$3x^2+5xy+6x-7y=4$$
$$8x^2+y^2+x+10y=3$$

si incontrano molte difficoltà (provare). Il problema è stato risolto solo nel 1965 con l'introduzione delle *basi di Gröbner* (Wolfgang Gröbner, 1899-1980, era un matematico austriaco) e dell'*algoritmo di Buchberger* (Bruno Buchberger, nato 1942), molto più profondo e complicato dell'algoritmo di eliminazione nel caso lineare. Sistemi di equazioni polinomiali appaiono in molti campi della matematica con numerose applicazioni in ingegneria e statistica. Per questa ragione la geometria algebrica computazionale (compresa la geometria algebrica reale, importantissima e molto difficile) è oggi un campo molto attivo della matematica, interagendo con la teoria dell'ottimizzazione, la teoria dei poliedri convessi, la crittografia, le equazioni differenziali parziali, la fisica teorica e, se ci si fida, la matematica finanziaria.



Wolfgang Gröbner

Bruno Buchberger nel 1987 ha fondato il *RISC* (*Research Institute for Symbolic Computation*), che ha sede nel castello di Hagenberg a 25 km da Linz e di cui è stato direttore fino al 2003.

Il RISC è un istituto dell'università di Linz e ospita circa 70 collaboratori, tra cui molti studenti. Le attività, iniziate con la geometria algebrica algoritmica nell'intento di sfruttare le possibilità offerte dall'algoritmo di Buchberger, sono molto varie, ma hanno tutte in qualche modo da fare con la risoluzione di equazioni e disequazioni, talvolta in senso molto lato confinando con la logica computazionale, la dimostrazione automatica di teoremi, l'inteligenza artificiale, la robotica e la chimica industriale.

Il castello di Hagenberg

www.risc.uni-linz.ac.at/.

In questo numero

Equazioni lineari in una incognita

Siano dati numeri reali a e b. Cercare di risolvere l'equazione $ax=b$ nell'incognita x significa cercare tutti i numeri reali x per i quali $ax=b$. Per $a\neq 0$ la soluzione è unica e data da $x=\dfrac{b}{a}$.

Dimostrazione: È chiaro che le seguenti equazioni sono equivalenti, cioè se x soddisfa una di esse, allora le soddisfa tutte:

$$ax=b$$
$$\frac{ax}{a}=\frac{b}{a}$$
$$x=\frac{b}{a}$$

È evidente che nel nostro ragionamento solo le proprietà algebriche formali dei numeri reali sono state usate e che rimane quindi valido, così come le considerazioni successive, se lavoriamo con numeri razionali o numeri complessi o altri insiemi di numeri con quelle corrispondenti proprietà. Si vede comunque anche che abbiamo avuto bisogno di poter dividere per un numero $\neq 0$, e quindi il risultato non è vero nell'ambito dei numeri naturali o interi.

Un insieme, su cui sono date due operazioni di addizione e di moltiplicazione che soddisfano le familiari regole di calcolo e in cui si può sempre dividere per un elemento $\neq 0$, in matematica si chiama un *campo*. Quindi l'algoritmo di eliminazione di Gauß rimane valido per sistemi di equazioni lineari i cui coefficienti appartengono a un campo qualsiasi.

Elenchiamo le regole che devono valere in un campo; verranno stabilite e studiate più dettagliatamente negli altri corsi.

$$a+(b+c)=(a+b)+c$$
$$a+b=b+a$$
$$a+0=a$$
$$a+(-a)=0$$
$$a(bc)=(ab)c$$
$$ab=ba$$
$$a1=a$$
$$\frac{1}{a}a=a \text{ per } a\neq 0$$
$$a(b+c)=ab+ac$$

Sistemi astratti

Definizione 23.1. Siano dati un insieme X ed una funzione $f : X \longrightarrow \mathbb{R}$. Denotiamo con $(f = 0)$ l'insieme degli zeri di f:

$$(f = 0) := \{x \in X \mid f(x) = 0\}.$$

Date m funzioni $f_1, \ldots, f_m : X \longrightarrow \mathbb{R}$, denotiamo inoltre con $(f_1 = 0, \ldots, f_m = 0)$ l'insieme $(f_1 = 0) \cap \cdots \cap (f_m = 0)$ degli zeri comuni di queste funzioni. Questa notazione è molto usata in calcolo delle probabilità.

Soprattutto quando una funzione f è data da una formula, la scriviamo talvolta nella forma $f = \bigcirc_x f(x)$.

Osservazione 23.2. *X sia un insieme. Con $\mathbb{R}^X$ si denota l'insieme di tutte le funzioni a valori reali definite su X. Possiamo formare somme di funzioni, moltiplicare funzioni con un numero reale o con un'altra funzione, e costruire combinazioni lineari di funzioni in questo spazio nel modo seguente.*

Siano $f, g, f_1, \ldots, f_m \in \mathbb{R}^X$ ed $\alpha, \alpha_1, \ldots, \alpha_m \in \mathbb{R}$. Allora:

$$f + g := \bigcirc_x f(x) + g(x)$$

$$\alpha f := \bigcirc_x \alpha f(x)$$

$$\alpha_1 f_1 + \cdots + \alpha_m f := \bigcirc_x \alpha_1 f_1(x) + \cdots + \alpha_m f_m(x)$$

Teorema 23.3. *Siano dati un insieme X ed m funzioni $f_1, \ldots, f_m : X \longrightarrow \mathbb{R}$. Siano $\alpha_1, \ldots, \alpha_m$ numeri reali con $\alpha_1 \neq 0$. Allora gli insiemi*

$$(f_1 = 0, \ldots, f_m = 0)$$

e

$$(\alpha_1 f_1 + \alpha_2 f_2 + \cdots + \alpha_m f_m = 0, f_2 = 0, \ldots, f_m = 0)$$

coincidono.

Dimostrazione. Per dimostrare l'uguaglianza tra i due insiemi, dobbiamo dimostrare che ogni elemento del primo insieme è anche elemento del secondo, e che ogni elemento del secondo insieme è elemento del primo.

Sia quindi x un elemento fissato di X.

(1) Sia $f_1(x) = 0, \ldots, f_m(x) = 0$. È chiaro che allora anche

$$\begin{aligned} &\alpha_1 f_1(x) + \alpha_2 f_2(x) + \cdots + \alpha_m f_m(x) = 0 \\ &f_2(x) = 0 \\ &\cdots \\ &f_m(x) = 0. \end{aligned}$$

(2) Sia viceversa

$$\begin{aligned} &\alpha_1 f_1(x) + \alpha_2 f_2(x) + \cdots + \alpha_m f_m(x) = 0 \\ &f_2(x) = 0 \\ &\cdots \\ &f_m(x) = 0. \end{aligned}$$

Dobbiamo dimostrare che $f_1(x) = 0$. Ma se sostituiamo $f_2(x) = 0, \ldots, f_m(x) = 0$ nella prima equazione, vediamo che

$$\alpha_1 f_1(x) = 0.$$

Qui possiamo adesso applicare la nostra ipotesi che $\alpha_1 \neq 0$. Essa implica $f_1(x) = 0$.

Attenzione: Il ragionamento non vale più, se non sappiamo che $\alpha_1 \neq 0$!

Nota 23.4. Nel seguito avremo $X = \mathbb{R}^2$ oppure, più in generale, $X = \mathbb{R}^n$. Nel primo caso scriveremo gli elementi di X nella forma (x, y), cosicché x denoterà la prima coordinata di un elemento e non l'elemento stesso.

Gli elementi di $\mathbb{R}^3$ saranno scritti nella forma (x, y, z), gli elementi di $\mathbb{R}^n$ nella forma $x = (x_1, \ldots, x_n)$. Questo passaggio da una notazione all'altra è frequente e diventerà presto familiare.

Due equazioni lineari in due incognite

Siano dati numeri reali $a_1, b_1, c_1, a_2, b_2, c_2$. Risolvere il sistema lineare

$$\begin{aligned} a_1 x + b_1 y &= c_1 \\ a_2 x + b_2 y &= c_2 \end{aligned}$$

significa trovare tutte le coppie (x, y) di numeri reali che soddisfano entrambe le equazioni. Per poter applicare il teorema 23.3 introduciamo le funzioni $f_1, f_2 : \mathbb{R}^2 \longrightarrow \mathbb{R}$ definite da

$$\begin{aligned} f_1(x, y) &:= a_1 x + b_1 y - c_1 \\ f_2(x, y) &= a_2 x + b_2 y - c_2 \end{aligned}$$

cosicché l'insieme delle soluzioni cercate coincide con $(f_1 = 0, f_2 = 0)$. Lasciamo come esercizio il caso molto facile che $a_1 = 0$.

Assumiamo quindi che $a_1 \neq 0$ e definiamo la funzione

$$f_3 := a_1 f_2 - a_2 f_1$$

Per il teorema 23.3 abbiamo

$$(f_1 = 0, f_2 = 0) = (f_1 = 0, f_3 = 0)$$

perché è f_2 che sicuramente appare con un coefficiente $\neq 0$ in f_3. Scritta per esteso l'equazione $f_3 = 0$ diventa

$$\underline{a_1 a_2 x} + a_1 b_2 y - a_1 c_2 - \underline{a_2 a_1 x} - a_2 b_1 y + a_2 c_1 = 0$$

cioè

$$(a_1 b_2 - a_2 b_1) y = a_1 c_2 - a_2 c_1,$$

e quindi le soluzioni del sistema originale coincidono con le soluzioni del sistema

$$\begin{aligned} a_1 x + b_1 y &= c_1 \\ (a_1 b_2 - a_2 b_1) y &= a_1 c_2 - a_2 c_1 \end{aligned}$$

Il numero $a_1 b_2 - a_2 b_1$ si chiama il *determinante* del sistema; lasciamo ancora come esercizio il caso che il determinante si annulli; se è invece $\neq 0$, allora la seconda equazione significa che

$$y = \frac{a_1 c_2 - a_2 c_1}{a_1 b_2 - a_2 b_1}.$$

Se per numeri reali a, b, c, d poniamo $\begin{vmatrix} a & b \\ c & d \end{vmatrix} := ad - bc$, possiamo scrivere

$$y = \frac{\begin{vmatrix} a_1 & c_1 \\ a_2 & c_2 \end{vmatrix}}{\begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix}}$$

Vediamo che anche il numeratore ha la forma di un determinante; infatti si ottiene dal denominatore sostituendo per la seconda colonna la colonna che costituisce il lato destro del sistema.

A questo punto possiamo calcolare anche x. Ricordando che $a_1 \neq 0$, otteniamo

$$x = \frac{c_1 - b_1 y}{a_1} = \frac{c_1 - b_1 \dfrac{a_1 c_2 - a_2 c_1}{a_1 b_2 - a_2 b_1}}{a_1} =$$

$$= \frac{a_1 b_2 c_1 - \underline{a_2 b_1 c_1} - b_1 a_1 c_2 + \underline{b_1 a_2 c_1}}{a_1 (a_1 b_2 - a_2 b_1)} =$$

$$= \frac{a_1 b_2 c_1 - b_1 a_1 c_2}{a_1 (a_1 b_2 - a_2 b_1)} = \frac{b_2 c_1 - b_1 c_2}{a_1 b_2 - a_2 b_1} = \frac{\begin{vmatrix} c_1 & b_1 \\ c_2 & b_2 \end{vmatrix}}{\begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix}}$$

Quindi nel caso che il determinante del sistema sia $\neq 0$, il sistema possiede un'unica soluzione data da

$$x = \frac{\begin{vmatrix} c_1 & b_1 \\ c_2 & b_2 \end{vmatrix}}{\begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix}} \qquad y = \frac{\begin{vmatrix} a_1 & c_1 \\ a_2 & c_2 \end{vmatrix}}{\begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix}}$$

Si osservi che il numeratore di x si ottiene anch'esso dal determinante del sistema, sostituendo stavolta la prima colonna con il lato destro del sistema. Questo risultato è molto importante per l'algebra lineare e può essere generalizzato a più dimensioni; è noto come *regola di Cramer*.

Esempi

Risolviamo con la regola di Cramer il sistema

$$\begin{aligned} 3x - 2y &= 8 \\ x + 6y &= 5 \end{aligned}$$

Il determinante del sistema è $\begin{vmatrix} 3 & -2 \\ 1 & 6 \end{vmatrix} = 18+2 = 20$, quindi diverso da 0, per cui

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

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

Esercizio. Risolvere da soli

$$\begin{aligned} 4x + 3y &= 10 \\ 2x + 9y &= 7 \end{aligned}$$

Esercizio. Perché non si può applicare la regola di Cramer al sistema

$$\begin{aligned} x + 3y &= 1 \\ 2x + 6y &= 4 \end{aligned}$$

Eppure non è difficile trovare "tutte" le soluzioni. Perché ho messo "tutte" tra virgolette? E perché è anche (quasi) facile trovare tutte le soluzioni di

$$\begin{aligned} x + 3y &= 2 \\ 2x + 6y &= 4 \end{aligned}$$

La forma generale della regola di Cramer

Sia dato un sistema di n equazioni lineari in n incognite (quindi il numero delle equazioni è uguale al numero delle incognite):

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= c_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= c_2 \\ &\cdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= c_n \end{aligned}$$

Anche in questo caso più generale si può definire il *determinante* del sistema, un numero che viene denotato con

$$D := \begin{vmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ & \cdots & & \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{vmatrix}$$

e si dimostrerà nel corso di Geometria I che questo determinante è $\neq 0$ se e solo se il sistema possiede un'unica soluzione che in tal caso è data da

$$x_1 = \frac{\begin{vmatrix} c_1 & a_{12} & \dots & a_{1n} \\ c_2 & a_{22} & \dots & a_{2n} \\ & \cdots & & \\ c_n & a_{n2} & \dots & a_{nn} \end{vmatrix}}{D}$$

$$x_2 = \frac{\begin{vmatrix} a_{11} & c_1 & \dots & a_{1n} \\ a_{21} & c_2 & \dots & a_{2n} \\ & \cdots & & \\ a_{n1} & c_n & \dots & a_{nn} \end{vmatrix}}{D}$$

$$\cdots$$

$$x_n = \frac{\begin{vmatrix} a_{11} & a_{12} & \dots & c_1 \\ a_{21} & a_{22} & \dots & c_2 \\ & \cdots & & \\ a_{n1} & a_{n2} & \dots & c_n \end{vmatrix}}{D}$$

x_i è quindi un quoziente il cui numeratore si ottiene dal determinante del sistema, sostituendo la i-esima colonna con il lato destro del sistema.

Determinanti

Conosciamo già i determinanti 2×2: $\begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix} = a_1 b_2 - a_2 b_1$.

Definizione 24.1. Per induzione definiamo i determinanti di ordine superiore:

$$\begin{vmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{vmatrix} := a_1 \begin{vmatrix} b_2 & c_2 \\ b_3 & c_3 \end{vmatrix} - a_2 \begin{vmatrix} b_1 & c_1 \\ b_3 & c_3 \end{vmatrix} + a_3 \begin{vmatrix} b_1 & c_1 \\ b_2 & c_2 \end{vmatrix}$$

$$\begin{vmatrix} a_1 & b_1 & c_1 & d_1 \\ a_2 & b_2 & c_2 & d_2 \\ a_3 & b_3 & c_3 & d_3 \\ a_4 & b_4 & c_4 & d_4 \end{vmatrix} := a_1 \begin{vmatrix} b_2 & c_2 & d_2 \\ b_3 & c_3 & d_3 \\ b_4 & c_4 & d_4 \end{vmatrix} - a_2 \begin{vmatrix} b_1 & c_1 & d_1 \\ b_3 & c_3 & d_3 \\ b_4 & c_4 & d_4 \end{vmatrix} + a_3 \begin{vmatrix} b_1 & c_1 & d_1 \\ b_2 & c_2 & d_2 \\ b_4 & c_4 & d_4 \end{vmatrix} - a_4 \begin{vmatrix} b_1 & c_1 & d_1 \\ b_2 & c_2 & d_2 \\ b_3 & c_3 & d_3 \end{vmatrix}$$

e così via. Si noti l'alternanza dei segni. I determinanti hanno molte proprietà importanti che verranno studiate nel corso di Geometria. Qui ci limiteremo a determinanti 2×2 e 3×3, per i quali dimostriamo alcune semplici regole, valide anche per determinanti $n \times n$, se riformulate in modo naturale.

Lemma 24.2. *Se in un determinante 2×2 scambiamo tra di loro due righe o due colonne, il determinante si moltiplica con -1.*

Dimostrazione. Immediata.

Lemma 24.3. *Un determinante 3×3 può essere calcolato anche secondo la regola*

$$\begin{vmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{vmatrix} := a_1 \begin{vmatrix} b_2 & c_2 \\ b_3 & c_3 \end{vmatrix} - b_1 \begin{vmatrix} a_2 & c_2 \\ a_3 & c_3 \end{vmatrix} + c_1 \begin{vmatrix} a_2 & b_2 \\ a_3 & b_3 \end{vmatrix}$$

Dimostrazione. Le due espansioni si distinguono in

$$-a_2 \begin{vmatrix} b_1 & c_1 \\ b_3 & c_3 \end{vmatrix} + a_3 \begin{vmatrix} b_1 & c_1 \\ b_2 & c_2 \end{vmatrix} = -a_2 b_1 c_3 + a_2 b_3 c_1 + a_3 b_1 c_2 - a_3 b_2 c_1$$

e

$$-b_1 \begin{vmatrix} a_2 & c_2 \\ a_3 & c_3 \end{vmatrix} + c_1 \begin{vmatrix} a_2 & b_2 \\ a_3 & b_3 \end{vmatrix} = -b_1 a_2 c_3 + b_1 a_3 c_2 + c_1 a_2 b_3 - c_1 a_3 b_2$$

che però, come vediamo, danno lo stesso risultato.

Lemma 24.4. *Se si scambiano due righe o due colonne in una matrice 3×3, il determinante si moltiplica per -1.*

Dimostrazione. Ciò, per il lemma 24.2, è evidente per lo scambio della seconda e della terza colonna e, per il lemma 24.3, anche per lo scambio della seconda e della terza riga. Se invece scambiamo la prima e la seconda colonna, otteniamo il determinante

$$\begin{vmatrix} b_1 & a_1 & c_1 \\ b_2 & a_2 & c_2 \\ b_3 & a_3 & c_3 \end{vmatrix} := b_1 \begin{vmatrix} a_2 & c_2 \\ a_3 & c_3 \end{vmatrix} - a_1 \begin{vmatrix} b_2 & c_2 \\ b_3 & c_3 \end{vmatrix} + c_1 \begin{vmatrix} b_2 & a_2 \\ b_3 & a_3 \end{vmatrix}$$

uguale, come si vede subito, al determinante originale moltiplicato per -1. Gli altri casi seguono adesso applicando le regole già dimostrate.

Lemma 24.5. *Se in un determinante appaiono due righe o due colonne uguali, allora il determinante è uguale a 0.*

Dimostrazione. Ciò per un determinante 2×2 è ovvio, e se ad esempio sono uguali le ultime due colonne, l'enunciato segue (usando il caso 2×2) dalla formula di espansione anche per i determinanti 3×3, e poi dal caso 3×3 anche per i determinanti 4×4 ecc.

Esempio. Verificare con calcoli a mano che

$$\begin{vmatrix} a_1 & a_1 & c_1 \\ a_2 & a_2 & c_2 \\ a_3 & a_3 & c_3 \end{vmatrix} = 0$$

e che

$$\begin{vmatrix} a_1 & b_1 + a_1 & c_1 \\ a_2 & b_2 + a_2 & c_2 \\ a_3 & b_3 + a_3 & c_3 \end{vmatrix} = \begin{vmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{vmatrix}$$

L'ultima uguaglianza è un caso speciale di un'altra proprietà fondamentale dei determinanti.

L'algoritmo di eliminazione di Gauß

La teoria dei determinanti e la regola di Cramer hanno una grandissima importanza teorica, ma non possono essere utilizzate se non per sistemi in due o al massimo tre incognite. Inoltre la regola di Cramer si applica solo al caso di un sistema quadratico. Esiste invece un metodo molto efficiente (anche nel calcolo a mano) per la risoluzione di sistemi di equazioni lineari, che viene detto *algoritmo di eliminazione di Gauß* e che consiste nella sistematica applicazione del teorema 23.3.

Esempio 25.1. Consideriamo il sistema

$$\begin{aligned} 3x + 2y - z &= 10 \dots f_1 \\ 4x - 9y + 2z &= 6 \quad \dots f_2 \\ x + y - 14z &= 2 \quad \dots f_3 \end{aligned}$$

In analogia a quanto abbiamo fatto a pagina 23 per il sistema 2×2 le funzioni f_1, f_2 ed f_3 sono definite da

$$\begin{aligned} f_1(x,y) &:= 3x + 2y - z - 10 \\ f_2(x,y) &:= 4x - 9y + 2z - 6 \\ f_3(x,y) &:= x + y - 14z - 2 \end{aligned}$$

Le indichiamo alla destra delle equazioni corrispondenti. Con la notazione che abbiamo introdotto nell'osservazione 23.2 poniamo

$$f_4 := 4f_1 - 3f_2$$

Per il teorema 23.3 allora

$$(f_1 = 0, f_2 = 0, f_3 = 0) = (f_4 = 0, f_2 = 0, f_3 = 0)$$

perché il coefficiente con cui f_1 appare in f_4 è diverso da 0. Esplicitamente $f_4 = 0$ equivale a

$$4(3x + 2y - z) - 3(4x - 9y + 2z) = 4 \cdot 10 - 3 \cdot 6$$

cioè a

$$35y - 10z = 22.$$

Se chiamiamo due sistemi *equivalenti* quando hanno le stesse soluzioni, possiamo dire che il sistema originale è equivalente al sistema

$$\begin{aligned} 4x - 9y + 2z &= 6 \quad \dots f_2 \\ x + y - 14z &= 2 \quad \dots f_3 \\ 35y - 10z &= 22 \dots f_4 \end{aligned}$$

Nell'ultima equazione la variabile x in f_4 è sparita, è stata *eliminata*. Ripetiamo questa operazione sostituendo la funzione f_2 con

$$f_5 := f_2 - 4f_3$$

Ciò è possibile perché in f_5 la f_2 appare con un coefficiente $\neq 0$. Esplicitamente $f_5 = 0$ significa

$$4x - 9y + 2z - 4(x + y - 14z) = 6 - 4 \cdot 2$$

cioè

$$-13y + 58z = -2.$$

Perciò il sistema originale ha le stesse soluzioni come il sistema

$$\begin{aligned} x + y - 14z &= 2 \quad \dots f_3 \\ 35y - 10z &= 22 \quad \dots f_4 \\ -13y + 58z &= -2 \dots f_5 \end{aligned}$$

Adesso formiamo $f_6 := 13f_4 + 35f_5$ che può sostituire sia la f_4 che la f_5. Possiamo togliere la f_5. $f_6 = 0$ è equivalente a

$$13(35y - 10z) + 35(-13y + 58z) = 13 \cdot 22 + 35 \cdot (-2),$$

cioè a

$$1900z = 216.$$

Otteniamo così il sistema

$$\begin{aligned} x + y - 14z &= 2 \quad \dots f_3 \\ 35y - 10z &= 22 \quad \dots f_4 \\ 1900z &= 216 \dots f_6 \end{aligned}$$

che è ancora equivalente a quello originale. Ma adesso vediamo che nell'ultima equazione è stata eliminata anche la y ed è rimasta solo la z che possiamo così calcolare direttamente:

$$z = \frac{216}{1900} = 0.11368,$$

poi, usando $f_4 = 0$, otteniamo

$$y = \frac{22 + 10z}{35} = 0.66105,$$

e infine dal $f_3 = 0$

$$x = -y + 14z + 2 = 2.930526.$$

Nella pratica si userà uno schema in cui vengono scritti, nell'ordine indicato dall'ordine delle variabili, solo i coefficienti. Nell'esempio appena trattato i conti verrebbero disposti nel modo seguente:

3	2	−1	10	f_1	✓
4	−9	2	6	f_2	✓
1	1	−14	2	f_3	
0	35	−10	22	$f_4 = 4f_1^* - 3f_2$	
0	−13	58	−2	$f_5 = f_2^* - 4f_3$	✓
0	0	1900	216	$f_6 = 13f_4 + 35f_5^*$	

L'asterisco indica ogni volta l'equazione cancellata in quel punto; l'uncino a destra di un'equazione significa che questa equazione è stata cancellata. Nei conti a mano spesso si preferirà forse cancellare la riga con un tratto orizzontale piuttosto di usare l'uncino.

Come si vede, nell'algoritmo cerchiamo prima di ottenere un sistema equivalente all'originale in cui tutti i coefficienti tranne al massimo uno nella prima colonna sono $= 0$, poi, usando le equazioni rimaste, applichiamo lo stesso procedimento alla seconda colonna (non modificando più però quella riga a cui corrisponde quell'eventuale coefficiente $\neq 0$ nella prima colonna), ecc. È chiaro che il procedimento termina sempre: alle m equazioni iniziali si aggiungono prima $m - 1$, poi $m - 2$, poi $m - 3$, ecc.

L'insieme delle soluzioni rimane sempre lo stesso; le equazioni cancellate naturalmente sono superflue e non vengono più usate. Quindi, se il sistema non ha soluzioni o più di una soluzione, riusciamo a scoprire anche questo.

Esempio 25.2. Consideriamo il sistema

$$\begin{aligned} 2x_1 - 5x_2 + 3x_3 - x_4 &= 1 \\ x_1 + 4x_2 - x_3 + 2x_4 &= 3 \\ 3x_1 - x_2 + 2x_3 + x_4 &= 7 \end{aligned}$$

Applichiamo il nostro schema:

2	−5	3	−1	1	f_1	✓
1	4	−1	2	3	f_2	
3	−1	2	1	7	f_3	✓
0	−13	5	−5	−5	$f_4 = f_1^* - 2f_2$	✓
0	−13	5	−5	−2	$f_5 = f_3^* - 3f_2$	
0	0	0	0	−3	$f_6 = f_4^* - f_5$	

Il sistema dato è quindi equivalente al sistema

$$\begin{aligned} x_1 + 4x_2 - x_3 + 2x_4 &= 3 \\ -13x_2 + 5x_3 - 5x_4 &= -2 \\ 0 &= -3 \end{aligned}$$

In particolare siamo arrivati alla contraddizione $0 = -3$, quindi il sistema non ha soluzione.

Sistemi con più di una soluzione

Consideriamo il sistema

$$\begin{aligned} 4x - y + 3z &= 5 \\ x + 2y - 10z &= 4 \\ 6x + 3y - 17z &= 13 \end{aligned}$$

Usiamo di nuovo il nostro schema di calcolo:

4	−1	3	5	f_1	✓
1	2	−10	4	f_2	
6	3	−17	13	f_3	✓
0	−9	43	−11	$f_4 = f_1^* - 4f_2$	✓
0	9	−43	11	$f_5 = 6f_2 - f_3^*$	
0	0	0	0	$f_6 = f_4^* + f_5$	

Stavolta non abbiamo una contraddizione, ma un'ultima equazione $0 = 0$ superflua, quindi siamo rimasti con due equazioni per tre incognite:

$$\begin{aligned} x + 2y - 10z &= 4 \\ 9y - 43z &= 11 \end{aligned}$$

Per ogni valore t di z possiamo risolvere

$$y = \frac{11 + 43t}{9}$$

$$x = -2y + 10t + 4 = \frac{-22 - 86t}{9} + 10t + 4 =$$

$$= \frac{-22 - 86t + 90t + 36}{9} = \frac{14}{9} + \frac{4}{9}t$$

e vediamo che l'insieme delle soluzioni è una retta nello spazio $\mathbb{R}^3$ con la rappresentazione parametrica

$$\begin{aligned} x(t) &= \frac{14}{9} + \frac{4}{9}t \\ y(t) &= \frac{11}{9} + \frac{43}{9}t \\ z(t) &= t \end{aligned}$$

Per ogni numero reale t si ottiene un punto $(x(t), y(t), z(t)) \in \mathbb{R}^3$ che è una soluzione del nostro sistema, e viceversa ogni soluzione è di questa forma.

L'insieme delle soluzioni di un sistema lineare

Negli esempi visti finora abbiamo trovato sistemi che non avevano soluzioni, oppure un'unica soluzione (descriventi cioè un unico punto nello spazio), oppure, nell'ultimo esempio, una retta di soluzioni.

Ciò vale per ogni sistema di equazioni lineari: l'insieme delle soluzioni è sempre o vuoto (nessuna soluzione), oppure un solo punto, oppure una retta, oppure un piano, oppure uno spazio affine tridimensionale ecc., e viceversa ogni insieme di questa forma può essere descritto da un sistema di equazioni lineari. La dimostrazione di questo teorema e la definizione precisa del concetto di spazio affine verranno date nel corso di Geometria I.

Nonostante l'efficienza dell'algoritmo di eliminazione che permette la risoluzione abbastanza agevole di sistemi lineari non troppo grandi (con un po' di pazienza si possono risolvere anche sistemi 10×10 a mano) la pratica è più complicata. Nelle applicazioni reali si affrontano sistemi con decine di migliaia di equazioni e variabili e non solo il tempo di calcolo, ma anche l'accumularsi di errori di arrotondamento nei calcoli approssimati che il software normalmente utilizza possono creare grandi problemi.

Piccoli errori, spesso inevitabili, nei dati in entrata (ad esempio nei coefficienti a_{ij} e b_i del nostro sistema) possono provocare in taluni casi, che bisogna riconoscere e controllare, grandi cambiamenti nelle soluzioni. Così il sistema

$$\begin{aligned} 4x + 1.99y &= 2.01 \\ 2x + y &= 1 \end{aligned}$$

possiede un'unica soluzione $x = 1, y = -1$, ma se lo cambiamo di poco,

$$\begin{aligned} 4x + 2y &= 2 \\ 2x + y &= 1 \end{aligned}$$

il determinante si annulla e l'insieme delle soluzioni è dato da $y = 1 - 2x$, e quindi le soluzioni non sono più univocamente determinate e possono essere arbitrariamente distanti dalla soluzione $(1, -1)$ del primo sistema.

Esercizi per gli scritti

25. Sia $f := \bigcirc_x 3x^4 - 2x^2 + 10x - 6 : \mathbb{R} \longrightarrow \mathbb{R}$. Calcolare $f(2)$.

26. Siano $f := \bigcirc_x 3x^2 + 4$ e $g := \bigcirc_x 6x - 1$ considerate come funzioni $\mathbb{R} \longrightarrow \mathbb{R}$. Calcolare $f \circ g$ e $g \circ f$.

27. Siano $f := \bigcirc_{(x,y)} x + y^3 : \mathbb{R}^2 \longrightarrow \mathbb{R}$ e $g := \bigcirc_x x^2 : \mathbb{R} \longrightarrow \mathbb{R}$. Calcolare $g \circ f$.

28. Risolvere con la regola di Cramer il sistema

$$\begin{aligned} 3x - 2y &= 9 \\ 2x + 7y &= 5 \end{aligned}$$

29. Calcolare il determinante

$$\begin{vmatrix} 3 & 8 & 2 \\ 6 & 2 & 1 \\ 4 & 2 & 1 \end{vmatrix}$$

30. Calcolare il determinante

$$\begin{vmatrix} 2 & 8 & 12 \\ 5 & 1 & 11 \\ 4 & 3 & 11 \end{vmatrix}$$

Risolvere i sistemi con l'algoritmo di Gauß usando lo schema.

31.
$$\begin{aligned} 2x_1 - 4x_2 + x_3 - x_4 &= 8 \\ x_1 + 5x_2 - x_3 + 2x_4 &= 0 \\ 2x_1 - x_2 + 4x_3 + x_4 &= 6 \\ 4x_1 + x_2 - x_3 + 3x_4 &= 10 \end{aligned}$$

32.
$$\begin{aligned} x_1 + 2x_2 + 3x_3 + 2x_4 &= 8 \\ 4x_1 + x_2 + 2x_3 + x_4 &= 5 \\ 4x_1 + 2x_2 + 3x_3 + 4x_4 &= 0 \\ x_1 - 2x_2 - 2x_3 - 2x_4 &= 4 \end{aligned}$$

33.
$$\begin{aligned} 8x + 7y + 15z &= 10 \\ 2x + 3y + 3z &= 4 \\ 3x + 2y + 6z &= 3 \end{aligned}$$

34.
$$\begin{aligned} 2x_1 + x_2 + x_3 + x_4 + 2x_5 &= 1 \\ 2x_1 + x_2 + x_3 + x_4 + x_5 &= 2 \\ x_1 - x_2 + 2x_3 + x_4 + 2x_5 &= 9 \\ x_1 + 2x_2 + 2x_3 + x_4 + 3x_5 &= 4 \\ 3x_1 + x_2 + x_4 + 2x_5 &= -2 \end{aligned}$$

35.
$$\begin{aligned} 4x_1 + 8x_2 + 7x_3 + x_4 + 2x_5 &= 0 \\ 3x_1 + 5x_2 + 2x_3 + 3x_4 + 4x_5 &= 1 \\ 2x_2 + x_3 + x_4 + 2x_5 &= 4 \\ 2x_1 + 3x_2 + x_3 + 6x_4 + 5x_5 &= -1 \\ 4x_1 + 2x_2 + x_3 + 7x_4 + 2x_5 &= 2 \end{aligned}$$

36.
$$\begin{aligned} x_1 + 5x_2 + x_3 + 4x_4 &= 1 \\ 2x_1 + x_2 + 6x_3 + 2x_4 &= 0 \\ 4x_1 + 2x_2 + x_3 + 7x_4 &= 2 \\ 2x_1 + x_2 + 3x_3 + 4x_4 &= -1 \end{aligned}$$

37.
$$\begin{aligned} 6x_1 + 2x_2 + 3x_3 + 5x_4 + 2x_5 + x_6 &= 1 \\ 3x_1 + x_2 + x_3 + 10x_4 + x_5 + 2x_6 &= 0 \\ 3x_1 + 2x_2 + x_3 + 2x_4 + x_5 + 2x_6 &= 0 \\ 2x_1 + x_3 + 5x_4 + x_5 + 4x_6 &= 0 \\ 3x_1 + x_2 + 2x_5 + x_6 &= 0 \\ 6x_1 + x_3 + 5x_4 + x_5 + 4x_6 &= 0 \end{aligned}$$

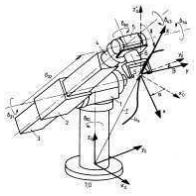
ALGORITMI E STRUTTURE DI DATI

Corso di laurea in matematica | Anno accademico 2005/06 | Numero 5

Trigonometria oggi

Dai piani di studio, soprattutto nell'università, la trigonometria è sparita da molto tempo. Ma questa disciplina, una delle più antiche della matematica, è ancora oggi una delle più importanti.

Mentre almeno gli elementi della trigonometria piana vengono insegnati nelle scuole, la trigonometria sferica è ormai conosciuta pochissimo anche tra i matematici di professione. Eppure le applicazioni sono tantissime: nautica, cartografia, geodesia e geoinformatica, astronomia, cristallografia, classificazione dei movimenti nello spazio, cinematica e quindi robotica e costruzione di macchine, grafica al calcolatore.



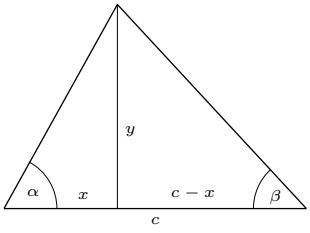
www.igm.rwth-aachen.de/deutsch/lehre-lehrveranstaltungen/guk/index.php.
L'immagine rappresenta un robot con i suoi movimenti.

In questo numero

Un problema di geodesia

Sia dato, come nella figura, un triangolo con base di lunghezza nota c e in cui anche gli angoli α e β siano noti e tali che $0 < \alpha, \beta < 90°$.

Vogliamo calcolare x ed y.



Per le nostre ipotesi $\tan\alpha$ e $\tan\beta$ sono numeri ben definiti e > 0 (cfr. pag. 29). Inoltre abbiamo

$$\tan\alpha = \frac{y}{x}$$
$$\tan\beta = \frac{y}{c-x}$$

Queste equazioni possono essere riscritte come sistema lineare di due equazioni in due incognite:

$$x\tan\alpha - y = 0$$
$$x\tan\beta + y = c\tan\beta$$

Il determinante $\begin{vmatrix} \tan\alpha & -1 \\ \tan\beta & 1 \end{vmatrix}$ di questo sistema è uguale a

$$\tan\alpha + \tan\beta$$

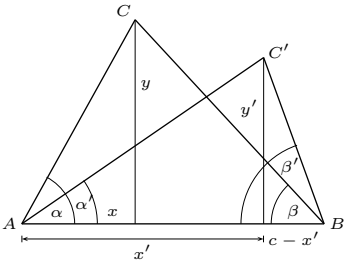
e quindi > 0. Possiamo perciò applicare la regola di Cramer e otteniamo

$$x = \frac{\begin{vmatrix} 0 & -1 \\ c\tan\beta & 1 \end{vmatrix}}{\tan\alpha + \tan\beta} = \frac{c\tan\beta}{\tan\alpha + \tan\beta}$$

mentre per y possiamo, se calcoliamo prima x, usare direttamente la relazione $y = x\tan\alpha$.

Esercizio: Prendendo il centimetro come unità di misura e con l'uso di un goniometro verificare le formule con le distanze nella figura.

Con questo metodo possiamo adesso risolvere un compito elementare ma frequente di geodesia illustrato dalla figura seguente.



Assumiamo di conoscere la distanza tra i punti A e B e, mediante un teodolite, di essere in grado di misurare gli angoli α, β, α' e β'. Vorremmo conoscere la distanza tra i punti C e C', ai quali però non possiamo accedere direttamente, ad esempio perché da essi ci separa un fiume che non riusciamo ad attraversare o perché si trovano in mezzo a una palude. Se le distanze sono molto grandi (maggiore di 50 km), dovremo appellarci alla trigonometria sferica, per distanze sufficientemente piccole invece possiamo utilizzare la tecnica vista sopra che ci permette di calcolare x, y, x' e y', da cui la distanza tra C e C' si ottiene come

$$|C - C'| = \sqrt{(x-x')^2 + (y-y')^2}$$

Calcoliamo l'errore $d - c$ che si commette approssimando la distanza d sulla sfera terrestre tra due punti mediante la lunghezza c del segmento di retta che si ottiene utilizzando la trigonometria piana:

c	$d - c$
50 km	0.13 m
100 km	1 m
500 km	128 m
1000 km	1029 m

Grafica al calcolatore e geometria

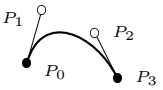
La geometria viene utilizzata in molti campi della tecnologia moderna: nella tomografia computerizzata, nella pianificazione di edifici, nella creazione di animazioni per film e pubblicità, nell'analisi dei movimenti di robot e satelliti.

La grafica al calcolatore e le discipline affini come la geometria computazionale e l'elaborazione delle immagini si basano sulla matematica. È importante separare gli algoritmi dalla loro realizzazione mediante un linguaggio di programmazione. È importante separare la rappresentazione matematica delle figure nello spazio dalle immagini che creiamo sullo schermo di un calcolatore.

Il matematico è molto avvantaggiato in questo. Già semplici nozioni di trigonometria e di geometria (traslazioni, rotazioni, riflessioni, coordinate baricentriche, i vari tipi di proiezioni) e algebra lineare possono rendere facili o immediate costruzioni e formule di trasformazione (e quindi gli algoritmi che da esse derivano) che senza questi strumenti matematici risulterebbero difficoltose o non verrebbero nemmeno scoperte.

La geometria proiettiva, apparentemente una vecchia teoria astratta e filosofica, diventa di sorpresa una tecnica molto utile per trasformare compiti di proiezione in semplici calcoli con coordinate omogenee.

I concetti dell'analisi e della geometria differenziale portano all'introduzione e allo studio delle curve e superficie di Bézier, largamente utilizzate nei programmi di disegno al calcolatore (CAD, computer aided design).



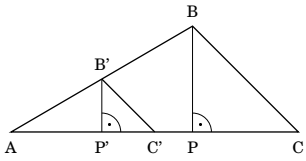
Molte figure possono essere descritte mediante equazioni algebriche; per questa ragione la geometria algebrica assume notevole importanza nella grafica al calcolatore moderna. Curve e superficie possono essere date in forme parametrica oppure mediante un sistema di equazioni; le basi di Gröbner forniscono uno strumento per passare da una rappresentazione all'altra.

La topologia generale, una disciplina tra la geometria, l'analisi e l'algebra, è la base della morfologia matematica, mentre la topologia algebrica e la geometria algebrica reale possiedono applicazioni naturali in robotica.

H. Pottmann/J. Wallner: Computational line geometry. Springer 1999.

W. Böhm/H. Prautzsch: Geometric concepts for geometric design. Peters 1994.

Il triangolo



In questa figura i segmenti BC e $B'C'$ sono paralleli. Nella geometria elementare si dimostra che le *proporzioni* del triangolo più piccolo $AB'C'$ sono uguali alle proporzioni del triangolo grande ABC. Cio significa che, se $\overline{AB}$ denota la lunghezza del segmento AB, allora

$$\frac{\overline{AB'}}{\overline{AB}} = \frac{\overline{AC'}}{\overline{AC}} = \frac{\overline{B'C'}}{\overline{BC}}$$

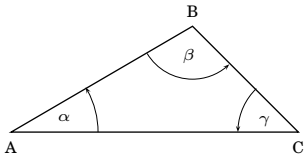
Se il valore comune di queste tre frazioni viene denotato con λ, abbiamo quindi

$$\begin{array}{l}\overline{AB'} = \lambda \cdot \overline{AB} \\ \overline{AC'} = \lambda \cdot \overline{AC} \\ \overline{B'C'} = \lambda \cdot \overline{BC}\end{array}$$

Una relazione analoga vale anche per le altezze:

$$\overline{B'P'} = \lambda \cdot \overline{BP}$$

Dati tre punti A, B, C denotiamo con $\sphericalangle(AC, AB)$ l'angolo α tra i segmenti AC e AB:



Evidentemente $0 < \alpha < 180°$.

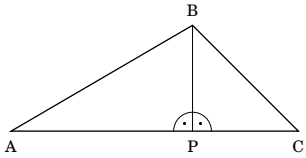
Con β e γ indichiamo gli altri due angoli come nella figura; spesso serve solo la grandezza assoluta degli angoli, allora si lasciano via le punte di freccia.

Nella prima figura il triangolo piccolo e il triangolo grande hanno gli stessi angoli, cioè

$$\begin{array}{l}\sphericalangle(AC, AB) = \sphericalangle(AC', AB') \\ \sphericalangle(BA, BC) = \sphericalangle(B'A, B'C') \\ \sphericalangle(CB, CA) = \sphericalangle(C'B', C'A)\end{array}$$

Si può dimostrare ed è chiaro intuitivamente che, dati due triangoli con gli stessi angoli, essi possono essere sovrapposti in maniera tale che si ottenga una figura simile alla nostra.

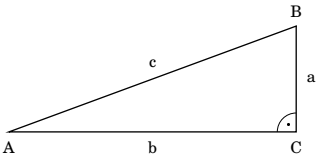
Ogni triangolo può essere considerato (talvolta anche in più modi - quando?) come unione di due triangoli rettangoli.



Le formule per i triangoli rettangoli sono particolarmente semplici; conviene quindi studiare separatamente i triangoli APB e PBC.

Il triangolo rettangolo

Il triangolo ABC sia rettangolo, ad esempio $\sphericalangle(CA, CB) = 90°$.



Il lato più lungo è quello opposto all'angolo retto, cioè AB, e si chiama *ipotenusa*, i due altri lati sono più brevi e sono detti *cateti*.

La somma dei tre angoli α, β, γ di un triangolo è sempre uguale a $180°$:

$$\alpha + \beta + \gamma = 180°.$$

Ciò implica che un triangolo può avere al massimo un angolo retto (se ce ne fossero due, il terzo dovrebbe essere zero e non avremmo più un triangolo).

Teorema di Pitagora: Dato un triangolo rettangolo e posto $a := \overline{BC}$, $b := \overline{AC}$ e $c := \overline{AB}$ come nella figura, si ha

$$a^2 + b^2 = c^2.$$

Dimostrazione: Pag. 29.

Il teorema di Pitagora implica che l'ipotenusa è veramente più lunga di ciascuno dei due cateti (perché $a, b > 0$). La relazione $c^2 = a^2 + b^2$ può essere anche usata per il calcolo di uno dei lati di un triangolo rettangolo dagli altri due:

$$\begin{array}{l}c = \sqrt{a^2 + b^2} \\ a = \sqrt{c^2 - b^2} \\ b = \sqrt{c^2 - a^2}\end{array}$$

Triple pitagoree

Una tripla pitagorea è una tripla (a, b, c) di numeri naturali positivi tali che $a^2 + b^2 = c^2$. La tripla pitagorea si chiama *primitiva*, se a, b e c sono relativamente primi tra di loro. Diamo una tavola delle prime triple pitagoree primitive in ordine crescente di c.

3	4	5
5	12	13
8	15	17
7	24	25
20	21	29
12	35	37
9	40	41
28	45	53
11	60	61
33	56	65
16	63	65
48	55	73

Gli arabi possedevano già nel 972 tavole simili a questa.

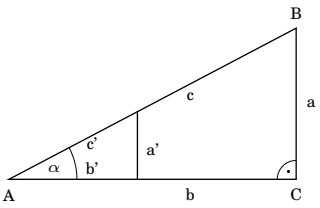
Per $n > 2$ non esistono invece soluzioni dell'equazione

$$a^n + b^n = c^n$$

con a, b, c interi > 0. La dimostrazione di questo teorema (detto *ultimo teorema di Fermat*) è stata molto difficile; per circa tre secoli i matematici l'avevano cercata invano e solo intorno al 1995 Andrew Wiles ci è riuscito, utilizzando strumenti molto avanzati della geometria algebrica.

Le funzioni trigonometriche

Consideriamo la seguente figura,



in cui a, b, c sono come prima i lati del triangolo rettangolo più grande e a', b' e c' sono i lati del triangolo più piccolo, che è ancora rettangolo. Le proporzioni nella figura dipendono solo dall'angolo α, si ha cioè

$$\frac{c'}{c} = \frac{b'}{b} = \frac{a'}{a},$$

e da ciò anche

$$\begin{array}{l}\dfrac{a'}{c'} = \dfrac{a}{c} \\ \dfrac{b'}{a'} = \dfrac{b}{a} \\ \dfrac{c'}{b'} = \dfrac{c}{b}\end{array}$$

Questi rapporti sono perciò funzioni dell'angolo α che vengono dette funzioni trigonometriche e denotate come segue:

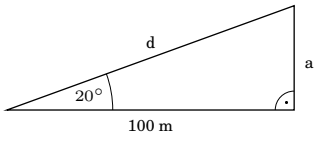
$$\begin{array}{l}\sin\alpha := \dfrac{a}{c} \ldots \text{seno di } \alpha \\ \cos\alpha := \dfrac{b}{c} \ldots \text{coseno di } \alpha \\ \tan\alpha := \dfrac{a}{b} \ldots \text{tangente di } \alpha \\ \cot\alpha := \dfrac{b}{a} \ldots \text{cotangente di } \alpha\end{array}$$

Dalle definizioni seguono le relazioni

$$\begin{array}{l}a = c\sin\alpha = b\tan\alpha \\ b = c\cos\alpha = a\cot\alpha \\ c = \dfrac{a}{\sin\alpha} = \dfrac{b}{\cos\alpha}\end{array}$$

Esercizio. Calcolare $\sin 45°$, $\cos 45°$, $\tan 45°$, $\cot 45°$.

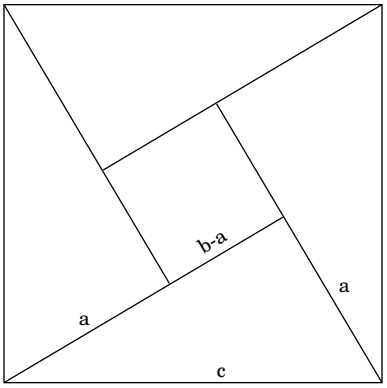
Esercizio. I valori delle funzioni trigonometriche si trovano in tabelle oppure possono essere calcolati con la calcolatrice tascabile oppure con una semplice istruzione in quasi tutti i linguaggi di programmazione. Ricavare in uno di questi modi i necessari valori per calcolare la distanza d e l'altezza a nella seguente figura:



Pierre de Fermat (circa 1607-1665) sostenne di conoscere una dimostrazione del teorema che poi portò il suo nome, ma non è mai stata trovata e si dubita molto che sia esistita.

La dimostrazione indiana

In una fonte indiana del dodicesimo secolo si trova il seguente disegno, con una sola parola in sanscrito: *guarda!*



Da esso si deduce immediatamente il teorema di Pitagora:

Il nostro triangolo rettangolo abbia i lati a, b, c con $a < b < c$. Allora l'area del quadrato grande è uguale a quella del quadrato piccolo più quattro volte l'area del triangolo, quindi

$$c^2 = (b-a)^2 + 4\frac{ab}{2},$$

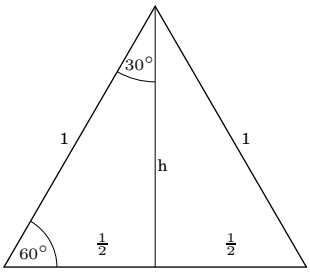
cioè

$$c^2 = b^2 - 2ab + a^2 + 2ab = b^2 + a^2.$$

Esercizio: Disegnare la figura nel caso che $a = b$ e convincersi che la dimostrazione rimane ancora valida.

Il triangolo isolatero

Consideriamo adesso un triangolo isolatero di lato 1. In esso anche gli angoli devono essere tutti uguali, quindi, dovendo essere la somma degli angoli $180°$, ogni angolo è uguale a $60°$.



Dalla figura otteniamo

$$h = \sqrt{1 - \tfrac{1}{4}} = \frac{\sqrt{3}}{2}$$

$$\sin 60° = \cos 30° = \frac{\sqrt{3}}{2}$$

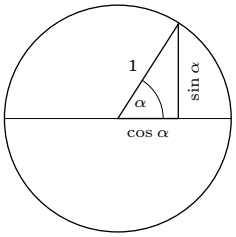
$$\sin 30° = \cos 60° = \frac{1}{2}$$

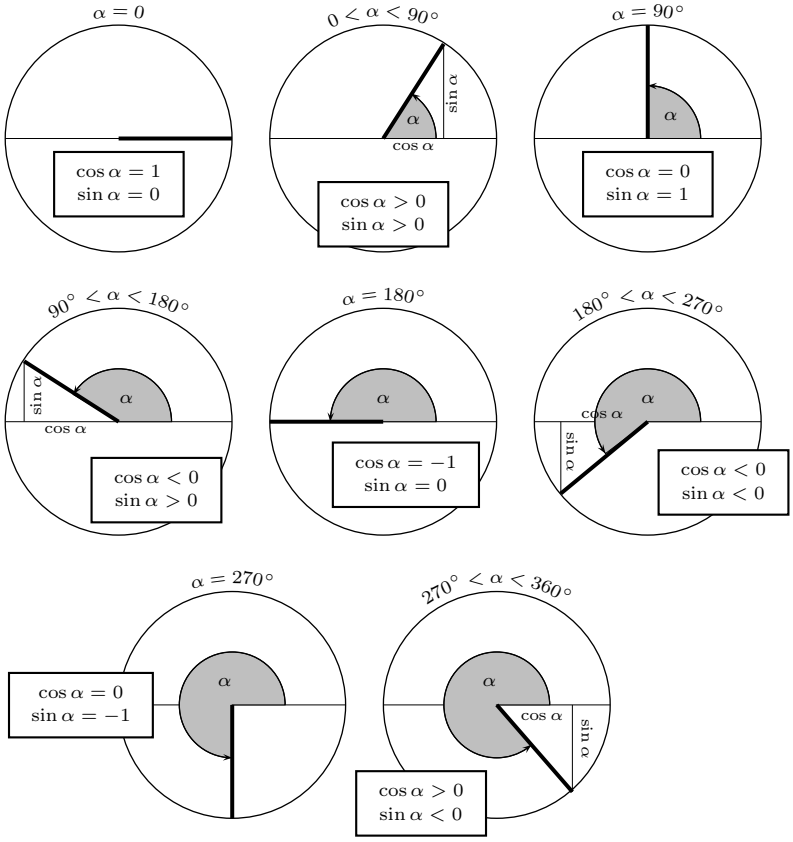
$$\tan 60° = 2h = \sqrt{3}$$

$$\tan 30° = \frac{1}{2h} = \frac{\sqrt{3}}{3}$$

Angoli sul cerchio

Siccome le lunghezze assolute non sono importanti, possiamo assumere che l'ipotenusa del triangolo rettangolo considerato sia di lunghezza 1 e studiare le funzioni trigonometriche sulla circonferenza di raggio 1.
Questo ci permette inoltre di estendere la definizione delle funzioni trigonometriche a valori arbitrari di α, non necessariamente sottoposti, come finora, alla condizione $0 < \alpha < 90°$. Definiamo prima $\sin\alpha$ e $\cos\alpha$ per ogni α con $0 \le \alpha \le 360°$ come nelle seguenti figure:





Definiamo poi ogni volta

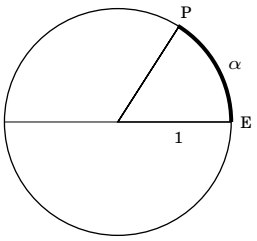
$$\tan\alpha := \frac{\sin\alpha}{\cos\alpha} \qquad \cot\alpha := \frac{\cos\alpha}{\sin\alpha}$$

quando $\cos\alpha \neq 0$ risp. $\sin\alpha \neq 0$. Si vede subito che questa definizione coincide con quella data a pag. 28, quando $0 < \alpha < 90°$.

Quindi $\tan\alpha = \dfrac{1}{\cot\alpha}$ quando entrambi i valori sono definiti.

Se α è infine un numero reale qualsiasi (non necessariamente compreso tra 0 e $360°$), esiste sempre un numero intero n tale che $\alpha = n \cdot 360° + \alpha_0$ con $0 \le \alpha_0 < 360°$ e possiamo definire $\cos\alpha := \cos\alpha_0$, $\sin\alpha := \sin\alpha_0$, $\tan\alpha := \tan\alpha_0$, $\cot\alpha := \cot\alpha_0$.

In matematica si identifica l'angolo con la lunghezza dell'arco descritto sulla circonferenza tra i punti E e P della figura a lato, aggiungendo però multipli del perimetro della circonferenza se l'angolo è immaginato ottenuto dopo essere girato più volte attorno al centro. Se il centro del cerchio è l'origine $(0,0)$ del piano, possiamo assumere che $E = (1,0)$. Siccome il perimetro della circonferenza di raggio 1 è 2π, si ha $360° = 2\pi$.

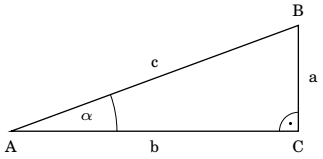


È chiaro che un angolo di $g°$ è uguale a $\dfrac{g}{360}2\pi$,
in altre parole $g° = \dfrac{2\pi g}{360}$, e viceversa $\alpha = \alpha\dfrac{360°}{2\pi}$ per ogni $\alpha \in \mathbb{R}$.

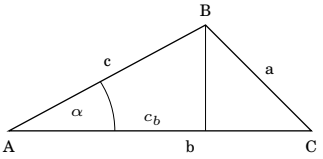
Infatti $1 = \dfrac{360°}{2\pi} \sim 57.29577951°$.

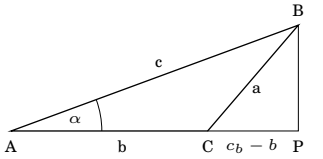
Il teorema del coseno

Dato un triangolo con i vertici A, B, C poniamo ancora $a := \overline{BC}$, $b := \overline{AC}$ e $c := \overline{AB}$. Denotiamo inoltre con c_b la lunghezza della proiezione di AB su AC misurando a partire da A. In modo analogo sono definite le grandezze c_a, b_a ecc. Se l'angolo α è ottuso, c_b sarà negativo. Sono possibili quattro situazioni:

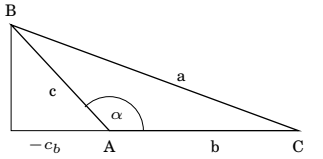


In questo caso $c_b = b$.





Si osservi che qui c_b è la lunghezza di tutto il segmento AP.



Teorema: In tutti i casi, quindi in ogni triangolo, vale la relazione

$$a^2 = b^2 + c^2 - 2bc_b.$$

Per simmetria vale anche

$$c^2 = a^2 + b^2 - 2ab_a.$$

Dimostrazione: Quando $c_b = b$, la formula diventa $a^2 = c^2 - b^2$ e segue direttamente dal teorema di Pitagora.

Nei rimanenti tre casi calcoliamo l'altezza del triangolo con il teorema di Pitagora in due modi.
Nella seconda figura abbiamo

$$c^2 - c_b^2 = a^2 - (b - c_b)^2,$$

cioè

$$c^2 - \underline{c_b^2} = a^2 - b^2 + 2bc_b - \underline{c_b^2},$$

per cui

$$c^2 = a^2 - b^2 + 2bc_b.$$

Similmente nella terza figura

$$c^2 - c_b^2 = a^2 - (c_b - b)^2,$$

la stessa equazione di prima.
Nella quarta figura infine abbiamo

$$c^2 - (-c_b)^2 = a^2 - (b - c_b)^2,$$

che è ancora la stessa equazione.

Teorema di Pitagora inverso: Un triangolo è rettangolo con l'ipotenusa c se e solo se

$$c^2 = a^2 + b^2.$$

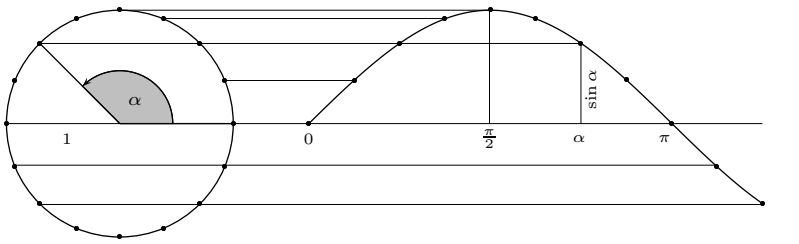
Dimostrazione: Dalla figura in alto a destra a pag. 28 si vede che il triangolo è rettangolo con ipotenusa c se e solo se $b_a = 0$ (oppure, equivalentemente, $a_b = 0$). L'enunciato segue dal teorema precedente.

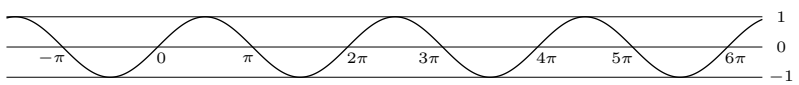
Teorema del coseno:

$$a^2 = b^2 + c^2 - 2bc\cos\alpha$$

Dimostrazione: $c_b = c\cos\alpha$ in tutti e quattro i casi del precedente teorema (cfr. le definizioni degli angoli sul cerchio a pag. 29).

Il grafico della funzione seno





Come si vede dalla figura e come sarà dimostrato rigorosamente nel corso di Analisi, la funzione seno è iniettiva sull'intervallo chiuso $[-\frac{\pi}{2}, \frac{\pi}{2}]$ e assume su questo intervallo tutti i valori possibili per il seno, cioè tutti i valori tra -1 e 1. Possiamo quindi definire una funzione biiettiva $\bigcirc_{\alpha} \sin\alpha : [-\frac{\pi}{2}, \frac{\pi}{2}] \longrightarrow [-1, 1]$. L'inversa di questa funzione viene denotata con $\arcsin$. In modo analogo si definiscono l'inversa $\arccos$ della funzione biiettiva $\bigcirc_{\alpha} \cos\alpha : [0, \pi] \longrightarrow [-1, 1]$ e l'inversa $\arctan$ della funzione biiettiva $\bigcirc_{\alpha} \tan\alpha : (-\frac{\pi}{2}, \frac{\pi}{2}) \longrightarrow (-\infty, \infty)$.

La periodicità di seno e coseno

Dalle definizioni date a pag. 29 segue che

$$\cos(\alpha + 360^\circ) = \cos\alpha$$
$$\sin(\alpha + 360^\circ) = \sin\alpha$$

per ogni numero reale α. Invece di 360° possiamo anche scrivere 2π, quindi

$$\cos(\alpha + 2\pi) = \cos\alpha$$
$$\sin(\alpha + 2\pi) = \sin\alpha$$

per ogni numero reale α. Le funzioni $\sin$ e $\cos$ sono quindi funzioni periodiche con periodo 2π.

Facendo percorrere α l'asse reale e riportando $\sin\alpha$ come ordinata, otteniamo il grafico della funzione seno rappresentato in basso a sinistra.

Altre proprietà di seno e coseno

$$\cos(-\alpha) = \cos\alpha$$
$$\sin(-\alpha) = -\sin\alpha$$

per ogni numero reale α, come si vede dai disegni a pagina 29. Il coseno è quindi una funzione *pari*, il seno una funzione *dispari*.

Teorema di addizione:

$$\sin(\alpha + \beta) = \sin\alpha \cdot \cos\beta + \sin\beta \cdot \cos\alpha$$
$$\cos(\alpha + \beta) = \cos\alpha \cdot \cos\beta - \sin\alpha \cdot \sin\beta$$

Dimostrazione: Corsi di Analisi.

Esercizio: $\cos\alpha = \sin(\alpha + \frac{\pi}{2})$.

Verificare l'enunciato prima nelle illustrazioni a pag. 29 e utilizzare poi il teorema di addizione per la dimostrazione.

Esercizio: Calcolare $\sin 2\alpha$ e $\cos 2\alpha$.

Teorema: $\sin^2\alpha + \cos^2\alpha = 1$.

Ciò segue direttamente dalle definizioni geometriche. Mentre queste proprietà algebriche delle funzioni trigonometriche rimangono valide anche per un argomento α complesso, ciò non è più vero per le disuguaglianze $|\sin\alpha| \leq 1$ e $|\cos\alpha| \leq 1$. Infatti, se dall'analisi complessa anticipiamo le formule

$$\cos z = \frac{e^{iz} + e^{-iz}}{2}$$
$$\sin z = \frac{e^{iz} - e^{-iz}}{2i}$$

valide per ogni numero complesso z, vediamo che ad esempio $\cos ai = \frac{e^{-a} + e^{a}}{2}$, quindi per a reale e tendente ad infinito (in questo caso e^{-a} tende a 0) $\cos ai$ si comporta come $\frac{e^a}{2}$ e tende quindi fortemente ad infinito.

arcsin, arccos e arctan

Queste funzioni, definite a sinistra, sono determinate dalle seguenti relazioni:

$\arcsin x = \alpha \iff \sin\alpha = x$

per $-1 \leq x \leq 1$ e $-\frac{\pi}{2} \leq \alpha \leq \frac{\pi}{2}$

$\arccos x = \alpha \iff \cos\alpha = x$

per $-1 \leq x \leq 1$ e $0 \leq \alpha \leq \pi$

$\arctan x = \alpha \iff \tan\alpha = x$

per $-\infty < x < \infty$ e $-\frac{\pi}{2} \leq \alpha \leq \frac{\pi}{2}$

ALGORITMI E STRUTTURE DI DATI

Riduzione di un vettore

Talvolta, ad esempio nelle operazioni insiemistiche, vogliamo ridurre un vettore a a un vettore senza elementi multipli, creare cioè un vettore che contiene ogni elemento di a una ed una sola volta.
La tecnica che usiamo è molto simile a quella vista nel crivello di Eratostene. Anche stavolta usiamo la funzione `Vett.filtro`.

	a
`1 dict begin`	a
`[ S`	$[\, a$
`{`	$\{\, [\, x_1 \dots x_i\ b$
`U length`	$[\, x_1 \dots x_i\ b$, $n = \lvert b \rvert$
`0 eq`	$[\, x_1 \dots x_i\ b\ (n = 0)$
`{exit}`	$[\, x_1 \dots x_i\ \ b = [\,]$
`{Vett.spezza`	$[\, x_1 \dots x_i\ x\ b$
`/x TU def`	$[\, x_1 \dots x_i\ x\ b$
`{x ne} Vett.filtro}`	$[\, x_1 \dots x_i\ x\ b$
`ifelse}`	$[\, x_1 \dots x_i\ x\ b\ \}$
`loop`	$[\, x_1 \dots x_i\ x\ b\ \}$
`T ] end`	$[x_1 \dots x_r]$

Possiamo perciò definire

```
/Vett.riduci {1 dict begin [ S {U length 0 eq {exit}
  {Vett.spezza /x TU def {x ne} Vett.filtro} ifelse} loop
  T ] end} def
```

Come `Vett.filtro`, la funzione è corretta anche per il vettore vuoto:

```
[1 2 3 1 2 5 0 1 0 4 4 4 0 1 7 3 2 1] Vett.riduci ==
% output: [1 2 3 5 0 4 7]

[] Vett.riduci ==
% output: []
```

Se volessimo usare questa funzione per creare una libreria di funzioni insiemistiche, definendo ad esempio

```
/Unione {Vett.concat Vett.riduci} def
```

ci troveremmo di fronte alla difficoltà di dover sostituire la funzione `ne` in `Vett.filtro` con una funzione analoga in cui la uguaglianza (o disuguaglianza) tra insiemi è definita in modo appropriato anche per insiemi. Con la nostra funzione abbiamo ad esempio

```
[[1 2] [2 1]] Vett.riduci ==
% output: [[1 2] [2 1]]
```

mentre naturalmente vorremmo considerare i vettori [1 2] e [2 1] uguali come insiemi. Una buona implementazione delle operazioni insiemistiche fa parte delle ultimi versioni del linguaggio Python che impareremo al secondo anno.

J. Schwartz/R. Dewar/E. Dubinsky/E. Schonberg: Programming with sets. An introduction to SETL. Springer 1986. SETL è un linguaggio per la programmazione insiemistica, purtroppo non molto diffuso e insufficientemente supportato, che ha influenzato anche Python.

cvx e cvlit

L'operatore `cvx` (*convert to executable*) trasforma un oggetto in un oggetto eseguibile, mentre l'operatore `cvlit` ha l'effetto opposto, come si vede dai tre esempi che seguono; esaminarli bene.

```
{1 add 2 mul} ==
% output: {1 add 2 mul}

{1 add 2 mul} cvlit ==
% output: [1 add 2 mul]
```

In questo numero


```
{1 add 2 mul} cvlit cvx ==
% output: {1 add 2 mul}
```

Si noti che, mentre `{ ... }` è un oggetto di tipo array, la parentesi `{` non è una marca, non viene posta sullo stack (per questa ragione nei nostri schemi non viene ripetuta nelle descrizioni dello stack), e non può ad esempio essere tolta con `T` o scambiata con `S`, come avviene invece per `[`. Leggere la voce *Procedures* alle pagine 32-33 del manuale di Adobe.

L'abbreviazione + per Vett.concat

Usiamo da ora in avanti l'abbreviazione `+` al posto di `Vett.concat`:

```
/+ {[ 3 -1 roll aload pop
  counttomark 2 add -1 roll aload pop ]} def
```

Vett.minug e Vett.magg

Definiamo due funzioni `Vett.minug` e `Vett.magg`: per un vettore (di numeri o di stringhe) a e un numero risp. una stringa x con `a x Vett.minug` otteniamo il vettore b di tutti gli elementi di a che sono $\leq x$, mentre `a x Vett.magg` ci fornisce il vettore di tutti gli elementi di a maggiori di x. Usiamo gli operatori `cvx` e `[S]`; quest'ultimo trasforma x in $[x]$ (esercizio 43).

L'analisi operazionale per `Vett.minug` è

	$a\ x$
`[S]`	$a\ [x]$
`{le}`	$a\ [x]$ `{le}`
`+`	$a\ [x$ `le]`
`cvx`	$a\ \{x$ `le}`
`Vett.filtro`	b

L'eseguibile è perciò

```
/Vett.minug {[S] {le} + cvx Vett.filtro} def
```

Sostituendo `le` con `gt` otteniamo

```
/Vett.magg {[S] {gt} + cvx Vett.filtro} def
```

Prova:

```
[3 5 8 1 4 3 2 0 9] 4 Vett.minug ==
% output: [3 1 4 3 2 0]

[3 5 8 1 4 3 2 0 9] 4 Vett.magg ==
% output: [5 8 9]
```

Quicksort

Una delle operazioni più frequenti e più importanti nell'informatica è l'ordinamento. Il più popolare e in un certo senso più efficiente algoritmo di ordinamento è il *Quicksort*. Assumiamo che dobbiamo ordinare in ordine crescente un vettore a di numeri. Togliamo il primo elemento a_0 da a e formiamo il vettore b di tutti gli elementi rimasti di a che sono $\leq a_0$. Poi formiamo il vettore c di tutti gli elementi di a che sono maggiori di a_0. Se adesso ordiniamo (ricorsivamente, cioè con lo stesso algoritmo che stiamo descrivendo) b e c, ottenimao due vettori b^* e c^*; tutti gli elementi di b^* sono $\leq a_0$, tutti gli elementi di c^* sono maggiori di a_0. Anteponendo a_0 a c^* otteniamo un vettore c^{**} che è ancora ordinato e i cui elementi sono tutti $\geq a_0$. Se concateniamo b^* e c^{**} otteniamo l'ordinamento desiderato a^* di a.

	a
`U length`	$a, n = \|a\|$
`0 eq`	$a\ (n = 0)$
`{}`	$\{\ a^* = a\ \}$
`{`	$\{\ a$
`Vett.spezza`	$a_0\ a$
`PU`	$a_0\ a\ a_0$
`2 copy`	$a_0\ a\ a_0\ a\ a_0$
`Vett.minug`	$a_0\ a\ a_0\ b$
`Vett.ordina`	$a_0\ a\ a_0\ b^*$
`R4`	$b^*\ a_0\ a\ a_0$
`Vett.magg`	$b^*\ a_0\ c$
`Vett.ordina`	$b^*\ a_0\ c^*$
`Vett.anteponi`	$b^*\ c^{**}$
`+}`	$a^*\}$
`ifelse`	a^*

L'eseguibile è

```
/Vett.ordina {U length 0 eq {} {Vett.spezza PU 2 copy
  Vett.minug Vett.ordina R4 Vett.magg Vett.ordina
  Vett.anteponi +} ifelse} def
```

Prova:

```
[0 1 2 5 3 8 0 1 2 3 9 4 5] Vett.ordina ==
% output: [0 0 1 1 2 2 3 3 4 5 5 8 9]
```

La funzione può essere anche utilizzata per ordinare alfabeticamente un vettore di stringhe:

```
[(Roma) (Pisa) (Padova) (Trento) (Adria)] Vett.ordina ==
% output: [(Adria) (Padova) (Pisa) (Roma) (Trento)]
```

Come tutte le funzioni ricorsive, anche la nostra diventa lenta per vettori molto lunghi. Vettori con 500 elementi vengono però ancora ordinati entro meno di un secondo, mentre un vettore con 5000 elementi provoca un errore per superamento dei limiti dello stack di esecuzione. A parte ciò, vediamo che anche operazioni complicate possono essere affrontate in PostScript in modo trasparente e strutturato.

Programmazione funzionale con + e cvx

Siccome un'espressione `{ ... }` dal punto di vista delle operazioni formali (ma non dal punto di vista semantico) si comporta come un vettore, possiamo usare `+` (il vecchio `Vett.concat`, cfr. pag. 31) per concatenare due tali espressioni:

```
{5 add} {6 mul} + pstack
% output: [5 add 6 mul]
```

Se aggiungiamo `cvx` dopo `+`, otteniamo di nuovo un'espressione eseguibile:

```
{5 add} {6 mul} + cvx pstack
% output: {5 add 6 mul}
```

Ciò implica che possiamo in modo semplicissimo definire la *composizione* (da sinistra verso destra) di due procedure in PostScript:

```
/Fun.comp {+ cvx} def
```

Facciamo due prove per la funzione $\bigcirc\limits_x (x+1)^2$:

```
3 {1 add} {U mul} Fun.comp exec ==
% output: 16

[1 2 3 5] {1 add} {U mul} Fun.comp Vett.map ==
% output: [4 9 16 36]
```

Si può anche scrivere direttamente

```
[1 2 3 5] {1 add} {U mul} + cvx Vett.map ==
% output: [4 9 16 36]
```

ma la nuova definizione ci porta più lontano. `Fun.comp` infatti è essa stessa una funzione, che solo per questa considerazione denotiamo con φ, e matematicamente siamo nella situazione

$$\varphi = \bigcirc_{(f,g)} \bigcirc_x g(f(x)) : Y^X \times Z^Y \longrightarrow Z^X$$

Possiamo quindi ad esempio provare a comporre φ a destra con l'evaluazione $\varepsilon := \bigcirc\limits_h h(3) : Z^X \longrightarrow T$ (i nomi degli insiemi non hanno importanza) e vedere se questa composizione viene calcolata correttamente:

```
/Epsilon {3 S exec} def

{1 add} {U mul} {Fun.comp} {Epsilon} Fun.comp exec ==
% output: 16
```

Perché questo risultato è corretto?

Abbiamo quindi scoperto che PostScript può simulare i linguaggi funzionali!

Creazione di coppie da due vettori

Molto importante è anche la seguente funzione che trasforma due vettori $a = [a_1 \dots a_n]$ e $b = [b_1 \dots b_n]$ (della stessa lunghezza) nel vettore di coppie $[[a_1\ b_1] \dots [a_n\ b_n]]$. Questa funzione ha moltissime applicazioni.

Con un'accurata analisi risulta più semplice di quanto possa sembrare. Usiamo anche qui le tecniche delle marcature e delle rotazioni iterate.

Con le abbreviazioni

```
/CARICA {aload pop} def

/COPPIA {2 array astore} def

/CTM {counttomark} def
```

possiamo effettuare l'analisi operazionale

	$a\ b$
`[`	$a\ b\ [$
`R3`	$[\ a\ b$
`CARICA`	$[\ a\ b_1 \dots b_n$
`CTM`	$[\ a\ b_1 \dots b_n, n+1$
`-1 roll`	$[\ b_1 \dots b_n\ a$
`CARICA`	$[\ b_1 \dots b_n\ a_1 \dots a_n$
`CTM`	$[\ b_1 \dots b_n\ a_1 \dots a_n\ 2n$
`2 idiv`	$[\ b_1 \dots b_n\ a_1 \dots a_n\ n$
`-1 1`	$[\ b_1 \dots b_n\ a_1 \dots a_n\ n\ {-1}\ 1$
`{`	$\{\ [\ [a_{i+1}\ b_{i+1}] \dots [a_n\ b_n]\ b_1 \dots b_i\ a_1 \dots a_i\ i$
`1 add`	$[\ [a_{i+1}\ b_{i+1}] \dots [a_n\ b_n]\ b_1 \dots b_i\ a_1 \dots a_i, i+1$
`-1 roll`	$[\ [a_{i+1}\ b_{i+1}] \dots [a_n\ b_n]\ b_1 \dots b_{i-1}\ a_1 \dots a_{i-1}\ a_i\ b_i$
`COPPIA`	$[\ [a_{i+1}\ b_{i+1}] \dots [a_n\ b_n]\ b_1 \dots b_{i-1}\ a_1 \dots a_{i-1}\ [a_i\ b_i]$
`CTM 1 roll}`	$[\ [a_i\ b_i] \dots [a_n\ b_n]\ b_1 \dots b_{i-1}\ a_1 \dots a_{i-1}\ \}$
`for`	$[\ [a_1\ b_1] \dots [a_n\ b_n]$
`]`	$[[a_1\ b_1] \dots [a_n\ b_n]]$

da cui otteniamo l'eseguibile

```
/Vett.coppie {[ R3 CARICA CTM -1 roll CARICA
  CTM 2 idiv -1 1
  {1 add -1 roll COPPIA CTM 1 roll} for ]} def
```

Per rendere la funzione più veloce, potremmo sostituire le abbreviazioni con le operazioni originali di PostScript (esercizio 44).

Addizione di due vettori

Da due vettori $a = [a_1 \dots a_n]$ e $b = [b_1 \dots b_n]$ vogliamo ottenere il vettore somma $[a_1 + b_1 \dots a_n + b_n]$.

Usando `Vett.coppie`, ciò è facilissimo. Otteniamo prima il vettore $[[a_1\ \ b_1] \dots [a_n\ \ b_n]]$, su ogni componente $[x\ \ y]$ del quale dobbiamo prima eseguire `CARICA` per porre x ed y sullo stack, poi `add`. Possiamo perciò definire

```
/Vett.add {Vett.coppie {aload pop add} Vett.map} def
```

Per la sottrazione dobbiamo soltanto sostituire `add` con `sub`:

```
/Vett.sub {Vett.coppie {aload pop sub} Vett.map} def
```

In modo simile potremmo definire `Vett.mul` oppure anche `Vett.and` e `Vett.or` per vettori booleani.

Addizione di funzioni

Vogliamo definire una funzione `Fun.add` che, date due funzioni a valori numerici f e g, crea la funzione $f + g := \bigcirc_x f(x) + g(x)$. Matematicamente la funzione cercata è quindi $\bigcirc_{(f,g)} \bigcirc_x f(x) + g(x)$.

In PostScript la troviamo utilizzando una tecnica che, in parte, abbiamo già usato nella definizione di `Vett.minug`, `Vett.magg` e `Fun.comp`.

(1) Esaminiamo prima il risultato che vogliamo ottenere, inserendo le funzioni come faremmo se f e g ci fossero note. In tal caso infatti potremmo effettuare l'analisi operazionale

	x
U	$x\ x$
f	$x\ f(x)$
S	$f(x)\ x$
g	$f(x)\ g(x)$
add	$f(x) + g(x)$

L'espressione cercata è quindi

```
{U f S g add}                (*)
```

(2) A questo punto dobbiamo però utilizzare f e g come argomenti della nostra funzione `Fun.add` e cercare di creare l'espressione (*) utilizzando `Vett.concat` (cioè `+`) e tenendo conto del fatto che f e g sono date come espressioni procedurali della forma $f = \{f_1 \dots f_r\}$, $g = \{g_1 \dots g_s\}$.

	$\{f_1 \dots f_r\}\ \{g_1 \dots g_s\}$
{U}	$\{f_1 \dots f_r\}\ \{g_1 \dots g_s\}$ {U}
RC3	$\{g_1 \dots g_s\}$ {U} $\{f_1 \dots f_r\}$
+	$\{g_1 \dots g_s\}$ [U $f_1 \dots f_r$]
{S}	$\{g_1 \dots g_s\}$ [U $f_1 \dots f_r$] {S}
+	$\{g_1 \dots g_s\}$ [U $f_1 \dots f_r$ S]
S	[U $f_1 \dots f_r$ S] $\{g_1 \dots g_s\}$
+	[U $f_1 \dots f_r$ S $g_1 \dots g_s$]
{add}	[U $f_1 \dots f_r$ S $g_1 \dots g_s$] {add}
+	[U $f_1 \dots f_r$ S $g_1 \dots g_s$ add]
cvx	{U $f_1 \dots f_r$ S $g_1 \dots g_s$ add}

Si noti che il risultato di `+` è sempre un vettore delimitato da parentesi quadre. Questa tecnica è molto importante! Otteniamo così la funzione cercata:

```
/Fun.add {{U} RC3 + {S} + S + {add} + cvx} def
```

Prove:

```
6 {3 mul} {1 add U mul} Fun.add exec ==
% output: 67
```

Il procedimento è in realtà molto semplice; l'ultima analisi operazionale è soltanto apparentemente complicata perché abbiamo scritto per esteso i componenti interni di f e g.

Esercizi per gli scritti

38. Con le limitazioni riguardanti l'uso di `eq` e `ne` per uguaglianza e disuguaglianza discusse a pagina 31 definiamo alcune operazioni insiemistiche. In questo e nei prossimi esercizi stabilire per ogni funzione la natura e l'ordine degli argomenti, eseguire l'analisi operazionale, descrivere l'effetto della funzione (intuibile facilmente dal nome) e fare una prova usando `==`.

```
/Elemento {false R3 {PU eq {S T true S} if} forall T} def
```

Provare per $3 \in \{5, 3, 0, 1, 7\}$, $4 \in \{5, 3, 0, 1, 7\}$.

39. Esaminare

```
/Inter {1 dict begin /b S def
  {b Elemento} Vett.filtro Vett.riduci end} def
```

Provare con $\{1, 5, 3, 7\} \cap \{3, 0, 5, 8\}$.

40. Esaminare

```
/Diff {1 dict begin /b S def
  {b Elemento not} Vett.filtro Vett.riduci end} def
```

Provare con $\{3, 4, 0, 5, 7, 6\} \setminus \{5, 6, 3, 9\}$.

41. Esaminare

```
/Esiste {false R3 S {PU exec {S T true S} if} forall T} def
```

Provare la funzione per determinare se gli insiemi $\{1, 3, 5, 6, 9\}$ e $\{1, 3, 5, 9\}$ contengono un numero pari.

42. Esaminare

```
/Pertutti {{not} + cvx Esiste not} def
```

Provare la funzione per determinare se $\{1, 3, 5, 6, 9\}$ e $\{1, 3, 5, 9\}$ consistono solo di numeri dispari.

43. `[S]` trasforma x in $[x]$.

44. Sostituire in `Vett.coppie` (pag. 32) le abbreviazioni con le operazioni originali di PostScript.

45. Creare una funzione `fattmap` che, dato un numero naturale n come input, restituisce (cioè mette sullo stack) il vettore $[0!\ 1!\ 2!\ \dots n!]$ dei fattoriali $k!$ per $0 \le k \le n$, utilizzando `Vett.map`.

46. Creare una funzione `fattsenzamap` che opera nello stesso modo come `fattmap`, ma invece di utilizzare `Vett.map` impiega un algoritmo simile a quello che abbiamo usato per calcolare un singolo fattoriale (soltanto che stavolta i risultati intermedi devono rimanere sullo stack e alla fine essere raccolti in un vettore).

47. Creare una funzione `rec` che restituisce il reciproco $\frac{1}{x}$ di un numero reale $x \neq 0$.

48. Come si ottiene adesso, utilizzando `fattsenzamap` (o `fattmap`) il vettore dei reciproci dei fattoriali fino ad n?

49. Il numero e, base del logaritmo naturale, è definito mediante una somma infinita:

$$e = \sum_{n=0}^{\infty} \frac{1}{n!} = 2.718281828\dots$$

cioè come il limite per $n \longrightarrow \infty$ delle somme finite $\sum_{k=0}^{n} \frac{1}{k!}$. Creare una funzione `en` che calcola, per n dato, una tale somma parziale. Fare una prova con

```
20 en ==
% output: 2.71828198
```

Le prime 6 cifre decimali sono corrette.

ALGORITMI E STRUTTURE DI DATI

Corso di laurea in matematica | Anno accademico 2005/06 | Numero 7

Tracciati

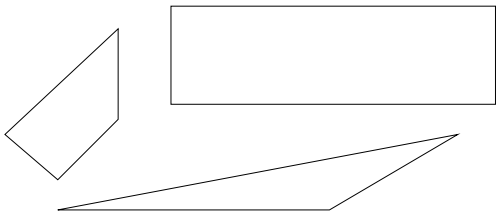
Abbiamo incontrato le seguenti istruzioni alle pagine 2, 4, 8.

`x y moveto`	Imposta la posizione della matita (detta *punto corrente, current point*) ad (x, y).
`x y lineto`	Aggiunge al tracciato la congiunzione della posizione attuale con (x, y).
`dx dy rlineto`	Aggiunge al tracciato la congiunzione della posizione attuale (x, y) con $(x + dx, y + dy)$.
`k setlinewidth`	Imposta lo spessore della matita.
`stroke`	Disegna il tracciato.
`fill`	Riempie l'interno del tracciato utilizzando algoritmi descritti nel manuale di Adobe.
`setrgbcolor`	Imposta il colore di disegno usando una scala RGB continua a valori nell'intervallo $[0, 1]$.

Un *tracciato* (in inglese *path*) è una successione di istruzioni grafiche che descrivono una figura. Un nuovo tracciato vuoto viene creato con `newpath`; un tracciato può essere disegnato con `stroke` e riempito con `fill`. Un tracciato consiste di *sottotracciati* connessi (*subpaths*, a pagina 190 del manuale di Adobe). `closepath` unisce il punto corrente al punto iniziale del sottotracciato attuale. Con

```
2.83 2.83 scale 0.1 setlinewidth newpath
17 0 moveto 53 0 lineto 70 10 lineto closepath
32 14 moveto 75 14 lineto 75 27 lineto
  32 27 lineto closepath
17 4 moveto 25 12 lineto 25 24 lineto
  10 10 lineto closepath stroke
```

otteniamo ad esempio



Stati grafici

I dati che compongono uno stato grafico sono elencati alle pagine 179-180 del manuale di Adobe. In particolare fanno parte dello stato grafico la posizione della matita (il punto corrente), il tracciato corrente, il colore di disegno, l'insieme dei caratteri, lo spessore della matita e il sistema di coordinate. Uno stato grafico può essere salvato con `gsave` e ripristinato con `grestore`. Noi useremo per questi comandi le abbreviazioni

```
/$ {gsave} def       /* {grestore} def
```

Assumiamo ad esempio che vogliamo colorare l'ultima figura di giallo, però con il bordo in nero. Definiamo in primo luogo alcuni colori:

```
/giallo {1 1 0 setrgbcolor} def
/nero {0 0 0 setrgbcolor} def
/rosso {1 0 0 setrgbcolor} def
/verde {0 1 0 setrgbcolor} def
```

Invece di ripetere due volte tutte le istruzioni che costituiscono il tracciato come in

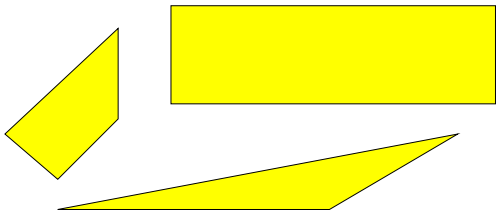
```
2.83 2.83 scale 0.1 setlinewidth newpath
17 0 moveto 53 0 lineto 70 10 lineto closepath
32 14 moveto 75 14 lineto 75 27 lineto
  32 27 lineto closepath
17 4 moveto 25 12 lineto 25 24 lineto
  10 10 lineto closepath giallo fill
```

In questo numero

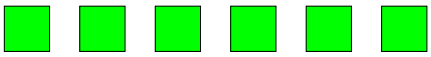

```
17 0 moveto 53 0 lineto 70 10 lineto closepath
32 14 moveto 75 14 lineto 75 27 lineto
  32 27 lineto closepath
17 4 moveto 25 12 lineto 25 24 lineto
  10 10 lineto closepath nero stroke
```

sfruttiamo il fatto che anche il tracciato fa parte dello stato grafico e viene salvato da `gsave` e ripristinato con `grestore`:

```
2.83 2.83 scale 0.1 setlinewidth newpath
17 0 moveto 53 0 lineto 70 10 lineto closepath
32 14 moveto 75 14 lineto 75 27 lineto
  32 27 lineto closepath
17 4 moveto 25 12 lineto 25 24 lineto
  10 10 lineto closepath
$ giallo fill * $ nero stroke *
```



Questa tecnica è molto importante; l'utilizzo coerente di `$` e `*` è utilissimo soprattutto in disegni con elementi ripetuti o comunque composti: si creano funzioni per questi elementi tipicamenti posizionati nell'origine; successivamente essi vengono ridisegnati in altre posizioni. Distingueremo anche tra funzioni che creano tracciati e funzioni che disegnano questi tracciati. Un primo esempio:



Questa figura è stata ottenuta con il programma

```
/impostagrafica {2.83 2.83 scale 0.1 setlinewidth} def

/quadratino {newpath -3 -3 moveto 3 -3 lineto
  3 3 lineto -3 3 lineto closepath} def

/disegno {10 10 60 {$ 2 translate quadratino
  $ verde fill * $ nero stroke * *} for} def

impostagrafica disegno
```

L'istruzione `impostagrafica` deve essere usata soltanto all'inizio del programma.

Useremo da ora in avanti l'abbreviazione `---` per `stroke`.
Inseriamo perciò nella nostra libreria la seguente definizione:

```
/--- {stroke} def
```

Archi

Un arco circolare con centro (x, y) e raggio r, il cui angolo percorre i valori (misurati in gradi) da u a v, è ottenuto dalle istruzioni

```
x y r u v arc
```

L'operazione aggiunge l'arco al cammino corrente (se questo non è annullato con `newpath`) e quindi si avrà un segmento di retta dal punto corrente all'inizio dell'arco. Se il punto corrente fa parte dell'arco, questo segmento di retta naturalmente non è visibile. Esempi:

```
/prova-1 {newpath 0 0 moveto 10 0 lineto 10 10 lineto
  0 0 2 sqrt 10 mul 45 90 arc closepath ---} def

/prova-2 {newpath 10 10 moveto
  0 0 2 sqrt 10 mul 45 90 arc closepath ---} def

/prova-3 {newpath 10 10 moveto
  0 0 2 sqrt 10 mul 45 90 arc ---} def

/prova-4 {newpath 0 0 moveto
  0 0 2 sqrt 10 mul 45 90 arc closepath ---} def

/prova-5 {newpath 0 0 moveto
  0 0 2 sqrt 10 mul 45 90 arc closepath fill} def

impostagrafica $ 3 5 translate prova-1 *
$ 20 5 translate prova-2 *
$ 35 5 translate prova-3 *
$ 50 5 translate prova-4 *
$ 65 5 translate $ rosso prova-5 * $ prova-4 * *
```

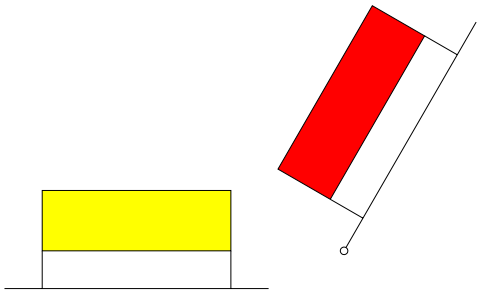
Esaminare ogni singola figura. Perché nella prima figura l'arco inizia nel punto $(10, 10)$?

Esistono anche le funzioni piuttosto simili `arcn`, `arct` e `arcto`, per le quali rimandiamo al manuale di Adobe.

Rotazioni

Il sistema di coordinate può essere ruotato con il comando α `rotate`, dove l'angolo α deve eessere indicato in gradi.

```
/partecolorata {5 5 moveto 30 5 lineto
  30 13 lineto 5 13 lineto closepath} def

/base {0 0 moveto 35 0 lineto
  5 0 moveto 5 5 lineto 30 0 moveto 30 5 lineto} def

impostagrafica partecolorata rgiallo --- base ---
$ 45 5 translate 60 rotate
partecolorata rrosso --- base ---
[0 0] 0.5 cerchio rbianco --- *
```



Nella figura il cerchietto indica il centro di rotazione!
Impariamo adesso come disegnare un cerchio.

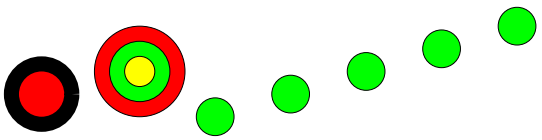
Cerchi

Vogliamo poter indicare il centro del cerchio come vettore $[x\ y]$, per cui l'analisi operazionale diventa:

	$[x\ y]\ r$
`S aload T`	$r\ x\ y$
`PU QU`	$r\ x\ y\ x\ r$
`add`	$r\ x\ y\ x+r$
`PU`	$r\ x\ y\ x+r\ y$
`moveto`	$r\ x\ y$
`RC3`	$x\ y\ r$
`0 360 arc`	

Quindi

```
/cerchio {S aload T PU QU add PU moveto RC3 0 360 arc} def
```



Otteniamo la figura con

```
impostagrafica
$ [5 6] 4 cerchio $ rosso fill * $ 2 setlinewidth --- * *
$ [18 9] 6 cerchio $ rosso fill * --- *
$ [18 9] 4 cerchio $ verde fill * --- *
$ [18 9] 2 cerchio $ giallo fill * --- *
$ 28 3 translate 1 1 5 {[0 0] 2.5 cerchio
  $ verde fill * --- 10 3 translate} for *
```

Se introduciamo le abbreviazioni

```
/rgiallo {$ giallo fill *} def
```

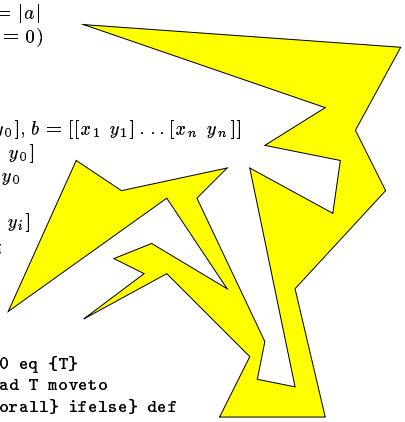
ecc., possiamo anche scrivere

```
impostagrafica
$ [5 6] 4 cerchio rrosso $ 2 setlinewidth --- * *
$ [18 9] 6 cerchio rrosso --- *
$ [18 9] 4 cerchio rverde --- *
$ [18 9] 2 cerchio rgiallo --- *
$ 28 3 translate
1 1 5 {[0 0] 2.5 cerchio
  rverde --- 10 3 translate} for *
```

Poligoni

L'argomento della funzione `poligono` è un vettore di punti, cioè di vettori numerici della forma $[x\ y]$, che vengono uniti da segmenti di rette. La funzione `poligonoc` corrisponde a un poligono chiuso.
È prevista anche la possibilità che il vettore sia vuoto.

	$a = [[x_0\ y_0][x_1\ y_1] \ldots [x_n\ y_n]]$
`U`	$a\ a$
`length`	$a\ n = \lvert a \rvert$
`0 eq`	$a\ (n = 0)$
`{`	$\{\ a$
`T`	$\}$
`{`	$\{\ a$
`Vett.spezza`	$[x_0\ y_0], b = [[x_1\ y_1] \ldots [x_n\ y_n]]$
`S`	$b\ [x_0\ y_0]$
`aload T`	$b\ x_0\ y_0$
`moveto`	b
`{`	$\{\ [x_i\ y_i]$
`aload T`	$x_i\ y_i$
`lineto}`	$\}$
`forall}`	$\}$
`ifelse`	



```
/poligono {U length 0 eq {T}
  {Vett.spezza S aload T moveto
  {aload T lineto} forall} ifelse} def

/poligonoc {poligono closepath} def
```

Ellissi

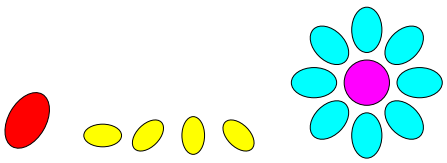
Otteniamo un'ellisse con assi a e b con centro nell'origine e con il primo asse parallelo all'asse delle x, effettuando l'istruzione `1 t scale`, con $t = b/a$, prima del comando `[0 0] a cerchio`. La funzione che adesso definiamo permette di impostare il centro (x, y) e un angolo α di rotazione. Dall'analisi operazionale

	$[x\ y]\ a\ b\ \alpha$
RC4	$a\ b\ \alpha\ [x\ y]$
aload T	$a\ b\ \alpha\ x\ y$
translate	$a\ b\ \alpha$
rotate	$a\ b$
PU	$a\ b\ a$
div	$a\ t = b/a$
1 S	$a\ 1\ t$
scale	a
[0 0] S	$[0\ 0]\ a$
cerchio	

arriviamo alla definizione

```
/ellisse {RC4 aload T translate rotate PU div
  1 S scale [0 0] S cerchio} def
```

L'ellisse richiede in genere l'esecuzione tra `$` e `*`, perché contiene istruzioni che modificano lo stato grafico (`translate`, `rotate`, `scale`).



Otteniamo queste figure con

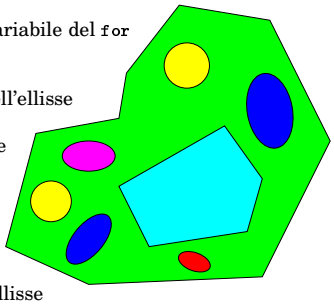
```
impostagrafica
$ [5 5] 4 2.5 60 ellisse rrosso --- *
$ 15 3 translate
0 1 3 {$ 45 mul rotate [0 0] 2.5 1.5 0
  ellisse rgiallo --- * 6 0 translate} for *
$ 50 10 translate
0 1 7 {$ 45 mul rotate [7 0] 3 2 0 ellisse rciano --- *} for
$ [0 0] 3 cerchio rmagenta --- * *
```

L'analisi operazionale dei `for` è molto semplice: Per le ellissi gialle abbiamo

	k % indice variabile del `for`
45 mul	$\alpha = 45k°$
rotate	% rotazione
...	% disegno dell'ellisse
6 0	6 0
translate	% traslazione

e per le ellissi di color ciano

	k
45 mul	$\alpha = 45k°$
rotate	% rotazione
...	% disegno dell'ellisse

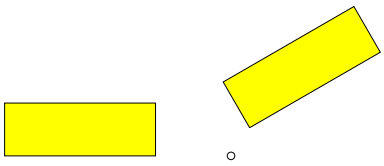


Rettangoli paralleli agli assi

La funzione `rettangolo` ha come argomenti la larghezza a e l'altezza b di un rettangolo parallelo agli assi il cui vertice in basso a sinistra coincide con l'origine.

```
/rettangolo {0 0 moveto PU 0 lineto U
  R3 lineto 0 S lineto 0 0 lineto} def
```

La figura può naturalmente esssere traslata o ruotata:



La figura è stata ottenuta con

```
impostagrafica
$ 5 0 translate 20 7 rettangolo rgiallo --- *
$ 35 0 translate 30 rotate
  $ 4 2 translate 20 7 rettangolo rgiallo --- * *
$ [35 0] 0.5 cerchio rbianco --- *
```

Il cerchietto indica di nuovo il punto di rotazione.

Esercizi per gli scritti

50. Esaminare

```
/Elemento {S [S] {eq} + cvx Esiste} def
```

come alternativa alla funzione dell'esercizio 38 (con `Esiste` come nell'esercizio 41).

51. Definire un operatore `forab` che con la sintassi

```
a b {operazioni} forab
```

corrisponde a un `for` da a a b con incremento 1. Molto facile e utile. Provare con `4 9 {==} forab`.

52. Definire operatori `for1n` e `for0n` che eseguono il `for` da 1 ad n risp. da 0 ad n. Provare con `7 {==} for1n` e `7 {==} for0n`.

53. Quando un punto $z = (a, b)$ del piano reale viene considerato come numero complesso, viene spesso scritto nella forma $z = a + ib$. La legge di moltiplicazione vista a pagina 22 diventa allora

$$(a + ib)(c + id) = ac - bd + i(ad + bc)$$

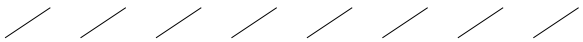
Con i si calcola come se fosse un numero, ma con la regola $i^2 = -1$.
Scrivere una funzione `Com.molt` per la moltiplicazione:

	$[a\ b]\ [c\ d]$
Com.molt	$[ac - bd\ ad + bc]$

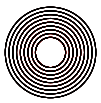
Provare con `[4 1] [3 5] Com.molt ==`.

54. Dimostrare che $(a + ib)^2 = a^2 - b^2 + 2iab$ e scrivere una funzione `Com.q` per il quadrato senza usare `Com.molt`. Provare con `[7 2] Com.q ==`.

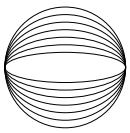
55. Definire una figura `tratto`, ripetendo la quale si ottiene la seguente figura:



56.



57.



Eventuali ombre ai lati della seconda figura sono artefatti di stampa; si tratta semplicemente di ellissi il cui asse b varia da 2 a 8.

58. Anche qui definire prima una figura `tratto`:

59. Analisi operazionale per `rettangolo`.

ALGORITMI E STRUTTURE DI DATI

Corso di laurea in matematica | Anno accademico 2005/06 | Numero 8

Rettangoli centrati

A differenza da `rettangolo` (pagina 36), la funzione `rettangolocr` crea un rettangolo di larghezza a ed altezza b, centrato in un punto (x, y) e ruotato di un angolo α. Se i parametri fossero già noti, potremmo semplicemente scrivere

x y `translate` α `rotate`
$-a/2$ $-b/2$ `translate` a b `rettangolo`

Siccome desideriamo una sintassi

$[x\ y]$ a b α `rettangolocr`

dobbiamo procedere come nella seguente analisi operazionale, in cui, per ripristinare il sistema di coordinate alla fine della creazione del cammino, abbiamo aggiunto le operazioni descritte nella colonna accanto:

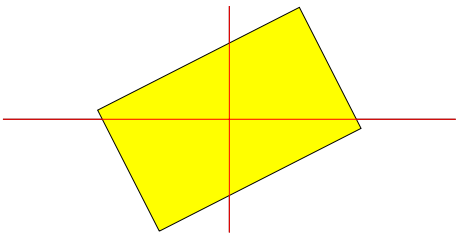
	$[x\ y]\ a\ b\ \alpha$
`matrix currentmatrix`	$[x\ y]\ a\ b\ \alpha\ A$
`R5`	$A\ [x\ y]\ a\ b\ \alpha$
`RC4`	$A\ a\ b\ \alpha\ [x\ y]$
`aload T`	$A\ a\ b\ \alpha\ x\ y$
`translate`	$A\ a\ b\ \alpha$
`rotate`	$A\ a\ b$
`PU`	$A\ a\ b\ a$
`2 div neg`	$A\ a\ b\ -a/2$
`PU`	$A\ a\ b\ -a/2\ b$
`2 div neg`	$A\ a\ b\ -a/2\ -b/2$
`translate`	$A\ a\ b$
`rettangolo`	A
`setmatrix`	

Il programma eseguibile diventa

```
/rettangolocr {matrix currentmatrix R5
  RC4 aload T translate rotate
  PU 2 div neg PU 2 div neg translate rettangolo
  setmatrix} def
```

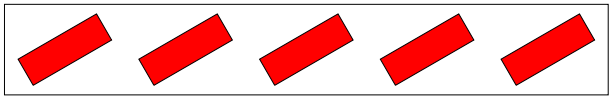
Verifichiamo che il sistema di coordinate viene ripristinato correttamente con

```
impostagrafica
/assi {PU neg 0 moveto S 0 lineto
  0 PU neg moveto 0 S lineto} def

$ 40 25 translate 30 20 assi stroke *

[40 25] 30 18 65 rettangolocr rgiallo ---

$ 40 25 translate 30 20 assi rosso stroke *
```



oppure

```
impostagrafica
$ 1 1 5 {[8 6] 12 4 30 rettangolocr rrosso ---
  16 0 translate} for *
80 12 rettangolo ---
```



In questo numero

Come salvare il sistema di coordinate

`grestore` (`*`) ripristina il vecchio stato grafico, compreso il cammino corrente, per cui perdiamo il cammino costruito all'interno di una coppia `$ ... *`. Come possiamo fare allora, se vogliamo soltanto ripristinare il vecchio sistema di coordinate, mantenendo però il cammino creato? PostScript non prevede comandi espliciti per il salvataggio e ripristino del sistema di coordinate. Si può procedere però nel modo seguente già utilizzato nella definizione di `rettangolocr`.

Il sistema di coordinate è tutto contenuto in una matrice A che si ottiene con `currentmatrix`. Con `matrix currentmatrix` viene posta sullo stack una copia di questa matrice che mediante un apposito `roll` trasferiamo in fondo alla lista degli argomenti. Alla fine delle operazioni per una determinata figura grafica tipicamente A si trova di nuovo in cima allo stack; con `setmatrix` il sistema di coordinate viene di nuovo posto uguale ad A. Lo schema generale è quindi

	$x_1 \dots x_n$
`matrix currentmatrix`	$x_1 \dots x_n\ A$
$n+1$ `1 roll`	$A\ x_1 \dots x_n$
...	A ...
...	A
`setmatrix`	

Usiamo questo metodo per migliorare la funzione `ellisse` in modo che non modifichi il sistema di coordinate:

	$[x\ y]\ a\ b\ \alpha$
`matrix currentmatrix`	$[x\ y]\ a\ b\ \alpha\ A$
`R5`	$A\ [x\ y]\ a\ b\ \alpha\ A$
`RC4`	$A\ a\ b\ \alpha\ [x\ y]$
`aload T`	$A\ a\ b\ \alpha\ x\ y$
`translate`	$A\ a\ b\ \alpha$
`rotate`	$A\ a\ b$
`PU`	$A\ a\ b\ a$
`div`	$A\ a\ t = b/a$
`1 S`	$A\ a\ 1\ t$
`scale`	$A\ a$
`[0 0] S`	$A\ [0\ 0]\ a$
`cerchio`	A
`setmatrix`	

per cui la nuova versione di `ellisse` diventa

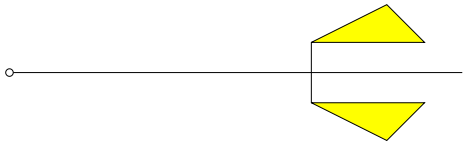
```
/ellisse {matrix currentmatrix R5
  RC4 aload T translate rotate PU div
  1 S scale [0 0] S cerchio setmatrix} def
```

La funzione `assi` viene usata nella forma `a b assi` e disegna una croce di assi nell'origine consistente dei segmenti di retta da $(-a, 0)$ ad $(a, 0)$ e da $(0, -b)$ a $(0, b)$.

Riflessioni

Se la retta in cui vogliamo riflettere una figura è l'asse delle x, è sufficiente il comando `1 -1 scale figura`:

```
impostagrafica
/figura {40 4 moveto 55 4 lineto 50 9 lineto 40 4 lineto
  40 0 lineto} def
$ 1 -1 scale figura rgiallo --- *
0 0 moveto 60 0 lineto --- figura rgiallo ---
[0 0] 0.5 cerchio rbianco ---
```



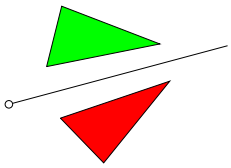
Il cerchietto indica l'origine. Se vogliamo riflettere in una retta passante per l'origine con un angolo α rispetto all'asse delle x, dobbiamo disegnare nelle nuove coordinate la riflessione della figura nelle vecchie coordinate:

nellenuovecoordinate `1 -1 scale` *nellevecchiecoordinate* `figura`

Matematicamente ciò corrisponde a una trasformazione della forma $\varphi \circ \sigma \circ \varphi^{-1}$, un'espressione ricorrente spesso in matematica.

Il comando diventa quindi α `rotate 1 -1 scale` $-\alpha$ `rotate`:

```
impostagrafica 30 0 translate
/figura {5 5 moveto 20 8 lineto 7 13 lineto 5 5 lineto} def
$ 15 rotate 1 -1 scale -15 rotate figura rrosso --- *
$ 15 rotate 0 0 moveto 30 0 lineto --- * figura rverde ---
[0 0] 0.5 cerchio rbianco ---
```



Se, nel caso più generale, la retta non passa necessariamente per l'origine, ma per un punto (x, y), formando sempre un angolo α con l'asse delle x, per passare a nuove coordinate dobbiamo usare

x y `translate` α `rotate`

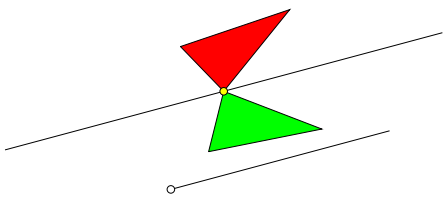
per tornare alle vecchie coordinate invece

$-\alpha$ `rotate` $-x$ $-y$ `translate`

per cui l'operazione corrisponde all'istruzione

x y `translate` α `rotate 1 -1 scale` $-\alpha$ `rotate` $-x$ $-y$ `translate`:

```
impostagrafica 30 0 translate
/figura {5 5 moveto 20 8 lineto 7 13 lineto 5 5 lineto} def
$ 7 13 translate 15 rotate 1 -1 scale
  -15 rotate -7 -13 translate figura rrosso --- *
$ 15 rotate 0 0 moveto 30 0 lineto --- *
$ 7 13 translate 15 rotate -30 0 moveto 30 0 lineto --- *
figura rverde ---
```



La retta, rispetto alla quale riflettiamo, adesso passa per il punto $(7, 13)$, indicato dal cerchietto giallo nel disegno.

Per ottenere una funzione in PostScript per la riflessione eseguiamo l'analisi operazionale

	$[x\ y]\ \alpha$
`S`	$\alpha\ [x\ y]$
`aload T`	$\alpha\ x\ y$
`2 copy`	$\alpha\ x\ y\ x\ y$
`translate`	$\alpha\ x\ y$
`RC3`	$x\ y\ \alpha$
`U`	$x\ y\ \alpha\ \alpha$
`rotate`	$x\ y\ \alpha$
`1 -1 scale`	$x\ y\ \alpha$
`neg`	$x\ y\ -\alpha$
`rotate`	$x\ y$
`neg`	$x\ -y$
`S`	$-y\ x$
`neg`	$-y\ -x$
`S`	$-x\ -y$
`translate`	

per cui definiamo

```
/Geom2.rifl {S aload T 2 copy translate RC3 U rotate
  1 -1 scale neg rotate neg S neg S translate} def
```

Possiamo allora riscrivere il programma per l'ultima figura, sostituendo la terza e la quarta riga con la semplice istruzione

```
$ [7 13] 15 Geom2.rifl figura rrosso --- *
```

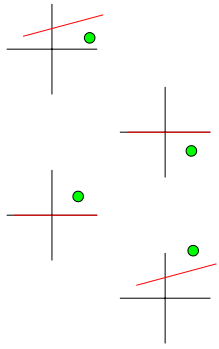
Analizziamo più in dettaglio le varie fasi della trasformazione per un singolo cerchietto che nel disegno appare verde. La retta riflettente è rossa.

Definiamo (solo per questo esperimento) le seguenti operazioni:

```
/phi {1 3 translate 15 rotate} def
/inversadiphi {-15 rotate -1 -3 translate} def
/rifl {1 -1 scale} def

/disegnapunto {$ [5 1.5] 0.7 cerchio rverde --- *} def
/disegnaretta {$ phi -5 0 moveto 6 0 lineto rosso --- *} def
/disegnaassi {$ 6 6 assi --- *} def
```

Esaminare adesso la sequenza di trasformazioni:

```
impostagrafica

$ 55 42 translate disegnaassi
disegnapunto disegnaretta *

$ 70 31 translate disegnaassi
inversadiphi
disegnapunto disegnaretta *

$ 55 20 translate disegnaassi
rifl inversadiphi
disegnapunto disegnaretta *

$ 70 9 translate disegnaassi
phi rifl inversadiphi
disegnapunto disegnaretta *
```



Nota 38.1. Il prodotto di due riflessioni definite da due rette g ed h non parallele è una rotazione attorno al punto di intersezione delle rette g ed h e ogni rotazione del piano può essere così ottenuta.

Il prodotto di due riflessioni definite da due rette parallele è una traslazione e ogni traslazione può essere così ottenuta.

Ciò implica, come si dimostra facilmente, che ogni movimento del piano può essere ottenuto come composizione di al massimo tre riflessioni. Tutta la geometria piana può quindi essere dedotta dalla teoria delle riflessioni.

Dimostrazione. Trevissoi, pagine 27-39.

C. Trevissoi: Grafica al calcolatore col Macintosh. Tesi Univ. Ferrara 1992.

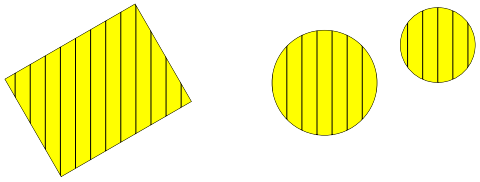
F. Bachmann: Aufbau der Geometrie aus dem Spiegelungsbegriff. Springer 1973.

L'operatore di restrizione clip

Accade spesso che si vorrebbe limitare il disegno a una certa area, sia per ottenere effetti artistici, sia semplicemente perché non si vuole che quest'area venga superata. È piuttosto complicato realizzare algoritmi a questo scopo e quindi è particolarmente gradito che PostScript preveda con `clip` un operatore apposito di facile utilizzo. Questa istruzione infatti ha l'effetto di limitare il disegno successivo all'area che verrebbe riempita con un comando `fill`. Lo stack non viene modificato da `clip`. L'area di disegno può essere anche piuttosto complicata, ad esempio un poligono oppure consistere di più parti sconnesse, come vedremo adesso.

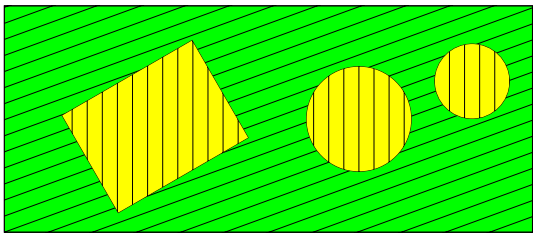
Usiamo una famiglia di rette verticali (quando non vengono ruotate) simile a quelle definite a pagina 8.

```
impostagrafica
/rette {-80 2 80 {-100 PU S moveto 100 lineto} for ---} def

[25 14] 20 15 30 rettangolocr
[55 15] 7 cerchio [70 20] 5 cerchio clip rgiallo rette
```



Nel secondo esempio la famiglia di rette viene girata e anche lo sfondo definito come area di disegno descritta da un rettangolo. Perché vengono disegnati anche i bordi dei cerchi e dei rettangoli?

```
impostagrafica
/rette {-80 2 80 {-100 PU S moveto 100 lineto} for ---} def

$ 5 0 translate 70 30 rettangolo rverde
  $ clip 110 rotate rette * --- *
[25 14] 20 15 30 rettangolocr
[52 15] 7 cerchio [67 20] 5 cerchio clip rgiallo rette
```



charpath

Questo operatore viene utilizzato con la sintassi `s v charpath`, dove `s` è una stringa e `v` un valore booleano (`true` oppure `false`) e aggiunge al cammino corrente il cammino che corrisponde al disegno dei caratteri della stringa. Per il significato del valore booleano si rimanda ai manuali; per i caratteri di tipo *Times*, *Helvetica* e *Symbol* comunque non sussiste differenza tra `true` e `false`.


```
impostagrafica
/rette {-80 2 80 {-100 PU S moveto 100 lineto} for ---} def

8 Font.times
7 12 moveto (siamo) false charpath
0 4 moveto (studenti a ) false charpath
17 Font.times (Ferrara) false charpath clip rmagenta
  60 rotate rette
```

L. Gass/J. Deubert: PostScript language tutorial and cookbook. Addison-Wesley 1986.

Cerchi concentrici

Definiamo una procedura `cerchi` che permette di disegnare una serie di cerchi concentrici. Essa ha come argomenti il centro (come vettore della forma $[x\ y]$, il vettore $[r_1 \dots r_n]$ dei raggi e un vettore $[f_1 \dots f_n]$ di operazioni che per ogni raggio vengono eseguite. Se vogliamo soltanto disegnare cerchi concentrici colorati, queste funzioni saranno della forma `rcolore ---`, ma la procedura `cerchi` è alquanto più generale e permette di eseguire un'operazione qualsiasi per ogni raggio, come adesso vedremo. Effettuiamo prima l'analisi operazionale:

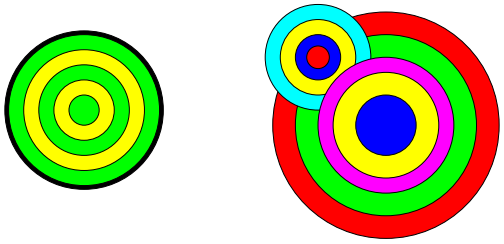
	$[x\ y]\ [r_1 \dots r_n]\ [f_1 \dots f_n]$
`Vett.coppie`	$[x\ y]\ [[r_1\ f_1] \dots [r_n\ f_n]]$
`{`	`{` $[x\ y]\ [r_i\ f_i]$
`aload T`	$[x\ y]\ r_i\ f_i$
`TU`	$[x\ y]\ r_i\ f_i\ [x\ y]$
`RC3`	$[x\ y]\ f_i\ [x\ y]\ r_i$
`cerchio`	$[x\ y]\ f_i$
`exec}`	$[x\ y]$ `}`
`forall`	$[x\ y]$
`T`	

La funzione `Vett.coppie` è stata introdotta a pagina 32. Otteniamo così l'eseguibile

```
/cerchi {Vett.coppie {aload T TU RC3 cerchio
  exec} forall T} def
```

Nella figura a sinistra abbiamo semplicemente creato una famiglia di cerchi concentrici; a destra invece al terzo raggio (che questo raggio abbia il valore 0 è indifferente) non corrisponde una funzione `{rcolore ---}`, ma inserisce un'altra serie di cerchi nella figura.

```
impostagrafica
% Sinistra.
[15 22] [10.5 10 8 6 4 2] [{rnero ---} {rverde ---}
  {rgiallo ---} {rverde ---} {rgiallo ---} {rverde ---}] cerchi
% Destra.
[55 20] [15 12 0 9 7 4] [{rrosso ---} {rverde ---}
  {[46 29] [7 5 3 1.5] [{rciano ---} {rgiallo ---}
    {rblu ---} {rrosso ---}] cerchi}
  {rmagenta ---} {rgiallo ---} {rblu ---}] cerchi
```

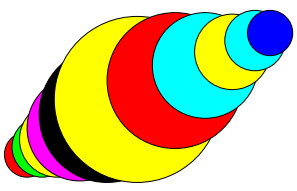


In caso di bisogno si possono facilmente inventare molte variazioni nell'utilizzo di questa funzione oppure creare altre funzioni simili.

Con

```
impostagrafica
[15 8] [3 4 5 7 9 11 9 7 5 4 3] [{rrosso --- 2 1 translate}
{rverde --- 2 1 translate} {rgiallo --- 3 1.5 translate}
{rmagenta --- 3.6 1.8 translate} {rnero --- 4 2 translate}
{rgiallo --- 5 2.5 translate} {rrosso --- 4 2 translate}
{rciano --- 3.6 1.8 translate} {rgiallo --- 3 1.5 translate}
{rciano --- 2 1 translate} {rblu --- 2 1 translate}] cerchi
```

otteniamo

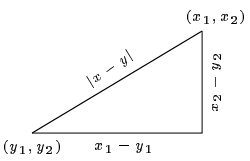


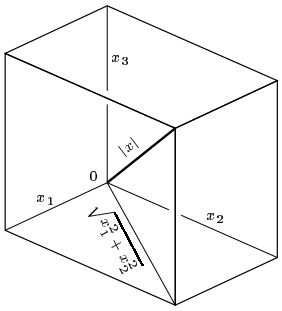
Distanze in $\mathbb{R}^n$

La distanza tra due punti $x = (x_1, x_2)$ e $y = (y_1, y_2)$ del piano reale $\mathbb{R}^2$ si calcola secondo il teorema di Pitagora come

$$|x - y| = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2}.$$

La distanza del punto x dall'origine è quindi $|x| = \sqrt{x_1^2 + x_2^2}$ e viceversa la distanza di x e y è proprio la lunghezza del vettore $x - y$.





Formule del tutto analoghe si hanno nello spazio tridimensionale $\mathbb{R}^3$. Calcoliamo prima la lunghezza $|x|$ di un vettore $x = (x_1, x_2, x_3)$ utilizzando la figura a destra, dalla quale si vede che

$$|x|^2 = (\sqrt{x_1^2 + x_2^2})^2 + x_3^2 = x_1^2 + x_2^2 + x_3^2,$$

per cui

$$|x| = \sqrt{x_1^2 + x_2^2 + x_3^2}.$$

Se adesso $y = (y_1, y_2, y_3)$ è un altro punto, la distanza tra x e y sarà uguale alla lunghezza di $x - y$, quindi

$$|x - y| = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + (x_3 - y_3)^2}.$$

Per ogni $n \geq 1$ possiamo definire lunghezze e distanze in $\mathbb{R}^n$ nello stesso modo. Per $x = (x_1, \ldots, x_n) \in \mathbb{R}^n$ poniamo

$$|x| := \sqrt{x_1^2 + \cdots + x_n^2},$$

e se $y = (y_1, \ldots, y_n)$ è un altro punto, la distanza tra x e y è la lunghezza di $x - y$, cioè

$$|x - y| = \sqrt{(x_1 - y_1)^2 + \cdots + (x_n - y_n)^2}.$$

Il prodotto scalare

Siano come sopra $x = (x_1, \ldots, x_n)$ e $y = (y_1, \ldots, y_n)$ due punti di $\mathbb{R}^n$. Calcoliamo la lunghezza $|x+y|$ della somma dei due vettori; questo è anche in statistica il punto di partenza per la definizione del *coefficiente di correlazione* che, nonostante il nome prometta molto di più, non è altro che un mezzo per confrontare x, y e $x + y$!

$$\begin{aligned} |x + y|^2 &= \sum_{k=1}^{n} (x_k + y_k)^2 = \sum_{k=1}^{n} x_k^2 + \sum_{k=1}^{n} y_k^2 + 2 \sum_{k=1}^{n} x_k y_k \\ &= |x|^2 + |y|^2 + 2 \sum_{k=1}^{n} x_k y_k \end{aligned}$$

L'espressione $\sum_{k=1}^{n} x_k y_k$ si chiama il *prodotto scalare* dei vettori x ed y. Esso è di fondamentale importanza per tutta la geometria. Introduciamo le abbreviazioni

$$\|x, y\| := (x, y) := x_1 y_1 + \cdots + x_n y_n$$

La seconda è più diffusa della prima, comporta però il pericolo di confusione con la coppia (x, y) che proprio nella statistica multidimensionale appare spesso contemporaneamente.

Sostituendo y con $-y$ otteniamo

$$|x - y|^2 = |x|^2 + |y|^2 - 2\|x, y\|.$$

I due punti x ed y formano insieme all'origine 0 un triangolo (eventualmente degenerato) i cui lati hanno le lunghezze $|x|$, $|y|$ e $|x - y|$. Assumiamo che il triangolo non sia degenerato e sia α l'angolo opposto al lato di lunghezza $|x - y|$. Per il teorema del coseno abbiamo

$$|x - y|^2 = |x|^2 + |y|^2 - 2|x||y| \cos \alpha, \text{ da cui } \|x, y\| = |x||y| \cos \alpha$$

In particolare $\|x, y\| = 0 \iff \cos \alpha = 0$.

Ortogonalità

Proposizione 40.1. *Due vettori x ed y di $\mathbb{R}^n$ sono ortogonali se e solo se $\|x, y\| = 0$.*

Dimostrazione. La formula $\|x, y\| = |x||y| \cos \alpha$ è sempre valida (esercizio 66).

Disuguaglianze fondamentali

Teorema 40.2. *Siano $x = (x_1, \ldots, x_n)$ e $y = (y_1, \ldots, y_n)$ due punti di $\mathbb{R}^n$. Allora*

$$|\|x, y\|| \leq |x||y|$$

*Questa è una delle disuguaglianze più importanti di tutta la matematica e prende il nome di **disuguaglianza di Cauchy-Schwarz**.*

Dimostrazione. Possiamo ricondurre questa fondamentale disuguaglianza al caso $n = 2$. Infatti i due vettori stanno su un piano e il prodotto scalare si esprime mediante l'angolo α che essi formano in questo piano:

$$\|x, y\| = |x||y| \cos \alpha$$

e siccome $|\cos \alpha| \leq 1$ abbiamo

$$|\|x, y\|| = |x||y||\cos \alpha| \leq |x||y|$$

Proposizione 40.3. *Siano ancora $x = (x_1, \ldots, x_n)$ ed $y = (y_1, \ldots, y_n)$ due punti di $\mathbb{R}^n$. Allora*

$$|x + y| \leq |x| + |y|$$

*Questa seconda disuguaglianza fondamentale è detta **disuguaglianza triangolare.***

Dimostrazione. Ciò è una facile conseguenza della formula

$$|x + y|^2 = |x|^2 + |y|^2 + 2\|x, y\|$$

per il prodotto scalare e della disuguaglianza di Cauchy-Schwarz:

$$\begin{aligned} |x + y|^2 &= |x|^2 + |y|^2 + 2\|x, y\| \leq \\ &\leq |x|^2 + |y|^2 + 2|x||y| = (|x| + |y|)^2 \end{aligned}$$

per cui anche $|x + y| \leq |x| + |y|$.

Il segno del prodotto scalare

Nota 40.4. Nella disuguaglianza di Cauchy-Schwarz anche a sinistra dobbiamo mettere il segno di valore assoluto, perché il prodotto scalare può essere negativo.

Infatti il segno del prodotto scalare ha una importantissima interpretazione geometrica: Siano come finora $x = (x_1, \ldots, x_n)$ e $y = (y_1, \ldots, y_n)$ due punti di $\mathbb{R}^n$, entrambi diversi da 0. Come nella dimostrazione della disuguaglianza di Cauchy-Schwarz sia α l'angolo che i due vettori formano in un piano comune (un tale piano esiste sempre ed è univocamente determinato se i due vettori non sono paralleli). Sappiamo che $\|x, y\| = |x||y| \cos \alpha$ e per ipotesi $|x| > 0$ e $|y| > 0$. Ciò implica che $\|x, y\|$ e $\cos \alpha$ hanno lo stesso segno; in particolare

$$\|x, y\| \geq 0 \iff \cos \alpha \geq 0$$

Fissiamo adesso x. Allora i vettori $y \in \mathbb{R}^n$ per i quali vale $\cos \alpha = 0$ sono esattamente i vettori ortogonali ad x. Essi formano l'*iperpiano ortogonale* ad x (una retta ortogonale ad x in $\mathbb{R}^2$, un piano ortogonale ad x in $\mathbb{R}^3$). Come si vede dalle figure a pagina 29, per $y = x$ il coseno di α è uguale ad 1, e se, partendo da $y = x$, avviciniamo y all'iperpiano ortogonale di x, il coseno diventa sempre più piccolo, rimanendo però positivo fino a quando non tocchiamo l'iperpiano ortogonale. Se y passa invece dall'altra parte dell'iperpiano, il coseno di α diventa negativo.

Avendo $\|x, y\|$ e $\cos \alpha$ però lo stesso segno, otteniamo il seguente importante enunciato geometrico:

Teorema 40.5. *Siano $x = (x_1, \ldots, x_n)$ e $y = (y_1, \ldots, y_n)$ due punti di $\mathbb{R}^n$, entrambi diversi da zero. Allora $\|x, y\| > 0$ se e solo se y si trova dalla stessa parte dell'iperpiano ortogonale ad x come x stesso.*

Rette e segmenti

Una retta R in $\mathbb{R}^n$ può essere rappresentata nella forma

$$R = \{p_0 + tw \mid t \in \mathbb{R}\} = p_0 + \mathbb{R}w$$

con $p_0, w \in \mathbb{R}^n$ e $w \neq 0$, equivalente alla rappresentazione parametrica

$$\begin{aligned} x_1 &= p_{0\,1} + tw_1 \\ &\dots \\ x_n &= p_{0\,n} + tw_n \end{aligned}$$

se $w = (w_1, \dots, w_n)$ e $p_0 = (p_{0\,1}, \dots, p_{0\,n})$.

In $\mathbb{R}^2$ possiamo scrivere $R = z_0 + \mathbb{R}w$ e

$$\begin{aligned} x &= x_0 + tu \\ y &= y_0 + tv \end{aligned}$$

se $w = (u, v)$ e $z_0 = (x_0, y_0)$.

La retta che congiunge due punti p_0 e p_1 distinti in $\mathbb{R}^n$ è l'insieme $p_0 + \mathbb{R}(p_1 - p_0)$. Se i due punti non sono distinti, questo insieme non è una retta, ma contiene soltanto l'unico punto dato.

Se come parametri usiamo solo i valori $t \in [0, 1]$, invece della retta otteniamo il segmento di retta che congiunge i due punti.

Equazione di una retta nel piano

Siano $z_0 = (x_0, y_0) \in \mathbb{R}^2$ e $w = (u, v) \in \mathbb{R}^2$ con $w \neq 0$. Allora il vettore $w^* = (-v, u)$ è ortogonale alla retta $R := z_0 + \mathbb{R}w$ e un punto z appartiene alla retta se e solo se $z - z_0$ è ortogonale a w^*.

Sia $w^* = (a, b)$, cioè $a = -v$ e $b = u$. Allora z appartiene ad R se e solo se x ed y soddisfano l'equazione

$$a(x - x_0) + b(y - y_0) = 0$$

che può essere scritta anche nella forma

$$ax + by = ax_0 + by_0$$

Siccome $w = (b, -a) \neq 0$, anche $(a, b) \neq 0$.

Sia viceversa data un'equazione della forma

$$ax + by = c$$

con $a, b \in \mathbb{R}$ non entrambi uguali a 0. Allora possiamo facilmente trovare $x_0, y_0 \in$ tali che $ax_0 + by_0 = c$ e vediamo che l'equazione descrive la retta $z_0 + \mathbb{R}w$ con $z_0 := (x_0, y_0)$ e $w := (b, -a)$.

Proiezione su una retta

Siano dati una retta $R = p_0 + \mathbb{R}w$ in $\mathbb{R}^n$ (con $w \neq 0$) e un punto $p \in \mathbb{R}^n$. Vogliamo calcolare la proiezione ortogonale m di p su R.

Il punto m deve essere in primo luogo un punto della retta e quindi della forma

$$m = p_0 + tw$$

inoltre il vettore $p - m$ deve essere ortogonale a w, cioè

$$\|p - m, w\| = 0$$

ossia

$$\|p, w\| = \|m, w\| = \|p_0 + tw, w\| = \|p_0, w\| + t\|w, w\|$$

Siccome $w \neq 0$, ciò è equivalente a $t = \dfrac{\|p, w\| - \|p_0, w\|}{\|w, w\|}$

e quindi abbiamo la formula fondamentale

$$t = \frac{\|p - p_0, w\|}{\|w, w\|}$$

Da essa otteniamo la proiezione con

$$m = p_0 + \frac{\|p - p_0, w\|}{\|w, w\|} w$$

Quando $|w| = 1$ (ciò si può sempre ottenere sostituendo w con $w/|w|$), abbiamo la rappresentazione

$$m = p_0 + \|p - p_0, w\| w$$

Dalla derivazione si vede anche che t e con esso m sono univocamente determinati. La distanza di p dalla retta è uguale a $|p - m|$.

Tutto ciò è valido in $\mathbb{R}^n$ per ogni n.

È geometricamente chiaro e facile da dimostrare che $m = p$ se p è un punto della retta.

Definiamo in PostScript una funzione per il calcolo del prodotto scalare di due vettori in $\mathbb{R}^n$:

	$[a_1 \dots a_n]\ [b_1 \dots b_n]$
`Vett.coppie`	$[[a_1\ b_1] \dots [a_n\ b_n]]$
`{aload T mul} Vett.map`	$[a_1 b_1 \dots a_n b_n]$
`Vett.somma`	$a_1 b_1 + \dots + a_n b_n$

```
/Geom.scalare {Vett.coppie {aload T mul} Vett.map
  Vett.somma} def
```

e una funzione per la moltiplicazione di un vettore con un numero:

	$v\ t$
`[S]`	$v\ [t]$
`{mul}`	$v\ [t]$ `{mul}`
`+ cvx`	v `{`t `mul}`
`Vett.map`	tv

```
/Geom.mult {[S] {mul} + cvx Vett.map} def
```

ottenendo così la funzione per il calcolo della proiezione di un punto su una retta in $\mathbb{R}^n$:

	$p\ p_0\ w$
`RC3`	$p_0\ w\ p$
`TU`	$p_0\ w\ p\ p_0$
`Vett.sub`	$p_0\ w\ p - p_0$
`PU`	$p_0\ w\ p - p_0\ w$
`Geom.scalare`	$p_0\ w\ \|p - p_0, w\|$
`PU U`	$p_0\ w\ \|p - p_0, w\|\ w\ w$
`Geom.scalare`	$p_0\ w\ \|p - p_0, w\|\ \|w, w\|$
`div`	$p_0\ w\ t$
`Geom.mult`	$p_0\ tw$
`Vett.add`	$p_0 + tw$

```
/Geom.proretta {RC3 TU Vett.sub PU Geom.scalare
  PU U Geom.scalare div Geom.mult pstack Vett.add} def
```

Riflessione in un punto

Siano p ed m due punti in $\mathbb{R}^n$. Diciamo che un punto p' è la riflessione di p in m, se m è il punto di dimezzamento del segmento tra p e p', cioè se

$$m = \frac{p + p'}{2}$$

Ciò implica che p' è univocamente determinato con $p' = 2m - p$.

Possiamo quindi definire una funzione in PostScript che realizza questa operazione:

	$p\ m$
`2`	$p\ m\ 2$
`Geom.mult`	$p\ 2m$
`S`	$2m\ p$
`Vett.sub`	$2m - p$

```
/Geom.riflinpunto {2 Geom.mult S Vett.sub} def
```

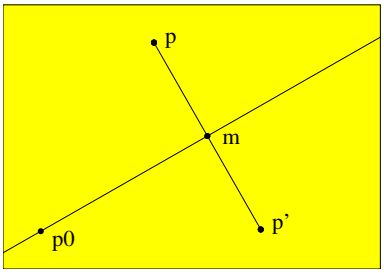
Riflessione in una retta

Siano p un punto ed R una retta in $\mathbb{R}^n$. Diciamo che un punto p' è la riflessione di p in R, se p' è la riflessione di p nella proiezione m di p sulla retta.

Possiamo facilmente combinare le funzioni `Proi` e `Riflpunto` per realizzare questa operazione in PostScript:

	$p\ p_0\ w$
`TU`	$p\ p_0\ w\ p$
`R3`	$p\ p\ p_0\ w$
`Geom.proretta`	$p\ m$
`Geom.riflinpunto`	p'

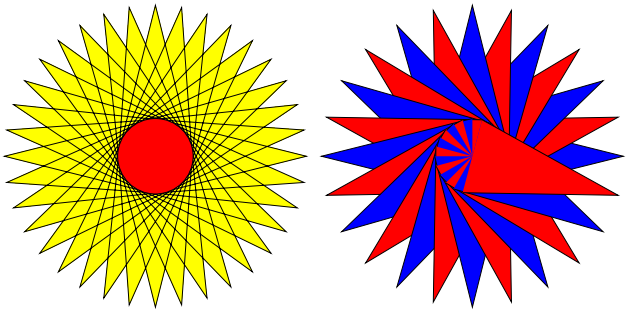
```
/Geom.riflinretta {TU R3 Geom.proretta Geom.riflinpunto} def
```



Otteniamo la figura con

```
impostagrafica 12 0 translate
50 35 rettangolo rgiallo clip ---
/p0 [5 5] def /w [35 20] def /p [20 30] def

/a p0 w -2 Geom.mult Vett.add def
/b p0 w 2 Geom.mult Vett.add def
[a b] poligono ---

/p' p p0 w Geom.riflinretta def
[p p'] poligono ---
/m p p0 w Geom.proretta def
/r 0.4 def
[p p' m p0] {r cerchio rnero} forall

3 Font.times
p aload T moveto 1.5 0 rmoveto (p) show
p' aload T moveto 1.5 0 rmoveto (p') show
p0 aload T moveto 1.5 -2 rmoveto (p0) show
m aload T moveto 2 -1 rmoveto (m) show
```

Serie circolari di rette



```
impostagrafica 20 0 translate
/punta {[[0 -5] [20 0] [0 5]] poligono} def
$ 36 {punta rgiallo 10 rotate} repeat --- *
[0 0] 5 cerchio rrosso ---
$ 42 0 translate 1 1 24
{$ punta Alg.pari {rrosso} {rblu} ifelse --- *
  15 rotate} for *
```

Esercizi per gli scritti

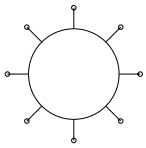
60. $f = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \dots$ sia una serie di potenze formale. Definendo il prodotto come per i polinomi, è definita la serie $(1+x)f = b_0 + b_1 x + b_2 x^2 + b_3 x^3 + \dots$

Come sono legati i b_k agli a_k?

61. Dedurre una nuova dimostrazione della formula di ricorsione per i numeri binomiali (teorema 13.12).

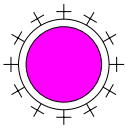
62. Analisi operazionale per `assi` (pagina 37).

63.



Definire prima una figura `spillo`.

64.



Definire prima una figura `croce` utilizzando `assi`.

65. Siano $x, y, z \in \mathbb{R}^n$ e $t \in \mathbb{R}$. Allora

$$\begin{aligned}\|x, y\| &= \|y, x\| \\ \|x + y, z\| &= \|x, z\| + \|y, z\| \\ \|tx, y\| &= t\|x, y\|\end{aligned}$$

66. x, y ed α siano definiti come a pagina 40 per il prodotto scalare. La formula $\|x, y\| = |x||y| \cos\alpha$ rimane valida anche se il triangolo formato da 0, x ed y è degenerato.

67. Trovare l'equazione della retta che congiunge i punti $(-2, 3)$ e $(7, 5)$.

68. Trovare l'equazione della retta che congiunge i punti $(7, 2)$ e $(10, 1)$.

69. Rappresentare la retta con equazione $3x + 11y = 7$ in forma parametrica.

70. Calcolare la proiezione ortogonale di $(2, 6)$ sulla retta $(1, 3) + \mathbb{R}(8, 2)$.

71. Calcolare in $\mathbb{R}^5$ la proiezione ortogonale di $(8, 7, 5, 10, 4)$ sulla retta $(1, 5, 2, 3, 6) + \mathbb{R}(2, 0, 1, 1, 5)$.

72. Calcolare la proiezione ortogonale di $(4, 1, 7)$ sulla retta che congiunge i punti $(3, 5, 1)$ e $(8, 4, 5)$.

73. Calcolare la riflessione del punto $(3, 5)$ nella retta che congiunge $(7, 2)$ e $(5, 4)$.

74. Completare

$x\ y$
$x\ y\ 2x^2$
$x\ y\ 2x^2\ 6xy$
$x\ y\ \ 2x^2 + 6xy$
$x\ y\ \ 2x^2 + 6xy + 7y^2$
$2x^2 + 6xy + 7y^2\ \ x\ y$
$2x^2 + 6xy + 7y^2\ \ x\ y^2$
$2x^2 + 6xy + 7y^2\ \ y^2\ x^2$
$2x^2 + 6xy + 7y^2\ \ x^2 + y^2$
$\dfrac{x^2 + 6xy + 7y^2}{x^2 + y^2}$

Secondo scritto venerdì 18 novembre, ore 16.

Terzo scritto venerdì 2 dicembre, ore 16.

ALGORITMI E STRUTTURE DI DATI

Corso di laurea in matematica | Anno accademico 2005/06 | Numero 9

Coordinate baricentriche su una retta

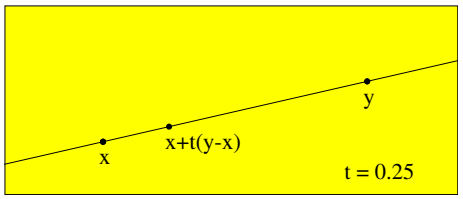
Abbiamo visto che la retta che congiunge due punti x ed y distinti in $\mathbb{R}^n$ è l'insieme

$$x + \mathbb{R}(y - x) = \{x + t(y - x) \mid t \in \mathbb{R}\}$$

Questo insieme è definito anche nel caso che $y = x$ e coincide in tal caso con il punto x. Per $y \neq x$ il parametro t per ogni punto p della forma $p = x + t(y - x)$ è univocamente determinato e si chiama la *coordinata baricentrica* di p rispetto ad x ed y; viceversa, anche quando $x = y$, ogni $t \in \mathbb{R}$ definisce naturalmente un unico punto $p = x + t(y - x)$. Per ottenere questo punto definiamo una funzione in PostScript nel seguente modo:

	$x\ t\ y$
`TU`	$x\ t\ y\ x$
`Vett.sub`	$x\ t\ \ y - x$
`S`	$x\ \ y - x\ \ t$
`Geom.mult`	$x\ \ t(y - x)$
`Vett.ad`	$x + t(y - x)$

```
/Geom.xty {TU Vett.sub S Geom.mult Vett.add} def
```



Per $t = 0$ si ottiene il punto x, per $t = 1$ il punto y, per $t = \frac{1}{2}$ il baricentro $\frac{x+y}{2}$ dei punti x ed y.

Rotazione di un punto nel piano

Nei corsi di Geometria si impara che il punto p' che si ottiene da un punto $p = (x, y)$ del piano mediante una rotazione attorno all'origine (e in senso antiorario) per l'angolo α è dato da

$$p' = \rho_\alpha(p) := (x\cos\alpha - y\sin\alpha, x\sin\alpha + y\cos\alpha)$$

Nel campo complesso questa rotazione corrisponde alla moltiplicazione di $x + iy$ con $\cos\alpha + i\sin\alpha$ (esercizio 75).

Se la rotazione avviene invece attorno ad un altro centro u, otteniamo il punto $u + \rho_\alpha(p - u)$.

Chiameremo `Geom.rotorigine` la funzione che effettua la rotazione attorno all'origine e `Geom.rot` la funzione per la rotazione attorno un centro u.

`Geom.rotorigine` esegue quindi la trasformazione

$$[x\ y]\ \ \alpha \longmapsto [x\cos\alpha - y\sin\alpha \quad x\sin\alpha + y\cos\alpha]$$

	$[x\ y]\ \alpha$
`U cos`	$[x\ y]\ \alpha, c = \cos\alpha$
`S sin`	$[x\ y]\ c, s = \sin\alpha$
	% Dobbiamo costruire $[xc - ys \quad xs + yc]$
`RC3`	$c\ s\ [x\ y]$
`aload T`	$c\ s\ x\ y$
`PU`	$c\ s\ x\ y\ x$
`4 index`	$c\ s\ x\ y\ x\ c$
`mul`	$c\ s\ x\ y\ xc$
`PU`	$c\ s\ x\ y\ xc\ y$
`4 index`	$c\ s\ x\ y\ xc\ y\ s$
`mul`	$c\ s\ x\ y\ xc\ ys$
`sub`	$c\ s\ x\ y, xc - ys$
`R5`	$xc - ys, c\ s\ x\ y$
`RC4`	$xc - ys, s\ x\ y\ c$
`mul`	$xc - ys, s\ x\ yc$
`R3`	$xc - ys, yc\ s\ x$
`mul`	$xc - ys, yc\ xs$
`add`	$xc - ys, xs + yc$
`2 array astore`	$[xc - ys \quad xs + yc]$


```
/Geom.rotorigine {U cos S sin RC3 aload T PU 4 index mul
  PU 4 index mul sub R5 RC4 mul R3 mul add
  2 array astore} def
```

Possiamo provare la funzione con

```
/p [1 0] 60 Geom.rotorigine def
p 30 Geom.rotorigine ==
% [0.0 1.0]
```

Per definire `Geom.rot` usiamo l'analisi operazionale

	$p\ u\ \alpha$
`RC3`	$u\ \alpha\ p$
`TU`	$u\ \alpha\ p\ u$
`Vett.sub`	$u\ \alpha, p - u$
`S`	$u, p - u, \alpha$
`Geom.rotorigine`	$u\ \rho_\alpha(p - u)$
`Vett.add`	$u + \rho_\alpha(p - u)$

da cui otteniamo l'eseguibile

```
/Geom.rot {RC3 TU Vett.sub S Geom.rotorigine Vett.add} def
```

Prova:

```
[3 1] [2 1] 30 Geom.rot [2 1] 60 Geom.rot ==
% [2.0 2.0]
```

Il vettore magico z^*

A pagina 41 per $z = (x, y) \in \mathbb{R}^2$ abbiamo introdotto il vettore $z^* = (-y, x)$ che si ottiene da z attraverso una rotazione per $90°$ attorno all'origine. In PostScript potremmo utilizzare `Geom.rotorigine` per ottenere z^*. Questa costruzione è però così importante che le dedichiamo una nuova apposita funzione:

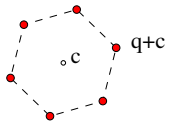
	$[x\ y]$
`aload T`	$x\ y$
`neg`	$x\ -y$
`S`	$-y\ x$
`2 array astore`	$[-y\ x]$

```
/Geom2.90 {aload T neg S 2 array astore} def
```

Questo vettore riapparirà continuamente non solo nella geometria elementare del piano, ma anche in molti contesti della geometria complessa e della meccanica!

Poligoni regolari

Definiamo due funzioni che creano i vertici di un poligono regolare. La prima, `Geom.ptigonocentro`, viene usata nella forma `c q n Geom.ptigonocentro` e pone sullo stack un vettore che contiene i vertici di un poligono regolare con n vertici e centro c; il primo elemento di questo vettore di vertici è $q + c$.



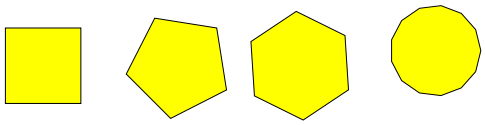
Quindi dobbiamo semplicemente ruotare $p = q + c$ attorno a c per gli angoli $k\alpha$ per $k = 1, \ldots, n-1$ con $\alpha = 360/n$. Come spesso accade, il bisogno di disporre i parametri nel modo corretto fa apparire la funzione più complicata di quanto in verità sia.

	$c\ q\ n$
`R3`	$n\ c\ q$
`PU Vett.add`	$n\ c, p = q + c$
`RC3`	$c\ p\ n$
`360`	$c\ p\ n\ 360$
`PU`	$c\ p\ n\ 360\ n$
`div`	$c\ p\ n, \alpha = 360/n$
`R4`	$\alpha\ c\ p\ n$
`[ R3`	$\alpha\ c\ [\ p\ n$
`1 sub`	$\alpha\ c\ [\ p, n-1$
`{`	$\{\ \alpha\ c\ [\ p_1 \ldots p_k\ k$
`PU S U`	$\alpha\ c\ [\ p_1 \ldots p_k\ p_k\ k\ k$
`4 add index`	$\alpha\ c\ [\ p_1 \ldots p_k\ p_k\ k\ \alpha$
`S`	$\alpha\ c\ [\ p_1 \ldots p_k\ p_k\ \alpha\ k$
`3 add index`	$\alpha\ c\ [\ p_1 \ldots p_k\ p_k\ \alpha\ c$
`S`	$\alpha\ c\ [\ p_1 \ldots p_k\ p_k\ c\ \alpha$
`Geom.rot}`	$\alpha\ c\ [\ p_1 \ldots p_{k+1}\ \}$
`for1n`	$\alpha\ c\ [\ p_1 \ldots p_n$
`]`	$\alpha\ c\ [p_1 \ldots p_n]$
`R3 T T`	$[p_1 \ldots p_n]$

```
/Geom.ptigonocentro {R3 PU Vett.add RC3 360 PU div R4
  [ R3 1 sub {PU S U 4 add index S 3 add index S Geom.rot}
  for1n ] R3 T T} def
```

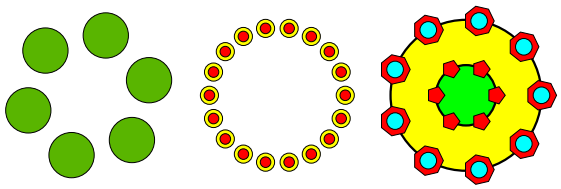
Per ottenere il corrispondente poligono adesso è sufficiente combinare questa funzione con `poligonoc`:

```
/gonocentro {Geom.ptigonocentro poligonoc} def
```



```
impostagrafica
[14 8] [5 5] 4 gonocentro
[32 8] [5 5] 5 gonocentro
[48 8] [6 4] 6 gonocentro
[66 10] [6 0] 13 gonocentro
rgiallo ---
```

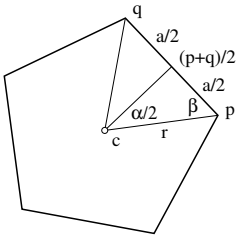
La possibilità di creare punti disposti nei vertici di un poligono offerta dalla funzione `Geom.ptigonocentro` (e dalla funzione `Geom.ptigonolato` che dobbiamo ancora definire) ha però molti altri usi come vediamo dai seguenti esempi:



```
impostagrafica
[15 12] [8 2] 6 Geom.ptigonocentro
  {3 cerchio $ 0.35 0.71 0 setrgbcolor fill * ---} forall
[40 12] [9 0] 18 Geom.ptigonocentro
  {U 1.2 cerchio rgiallo 0.7 --- cerchio rrosso ---} forall
$ [65 12] 10 cerchio 0.3 setlinewidth rgiallo --- *
[65 12] [10 0] 9 Geom.ptigonocentro
  {U [2 0] 7 gonocentro rrosso ---
  1.1 cerchio rciano ---} forall
$ [65 12] 4 cerchio 0.3 setlinewidth rverde --- *
[65 12] [4 0] 6 Geom.ptigonocentro
  {[1.2 0] 5 gonocentro rrosso ---} forall
```

Il centro di un poligono regolare

Dato un n-poligono regolare con due vertici adiacenti p e q, denotiamo con a la lunghezza di un suo lato, con r il raggio del cerchio circoscritto e con c il centro del poligono. $\alpha = 360/n$ sia di nuovo l'angolo centrale, β l'angolo laterale come nella figura.



Allora

$$\alpha = \frac{360}{n}$$

$$\beta = \frac{180 - \alpha}{2} = 90 - \frac{180}{n}$$

$$a = |q - p| = 2r \cos\beta$$

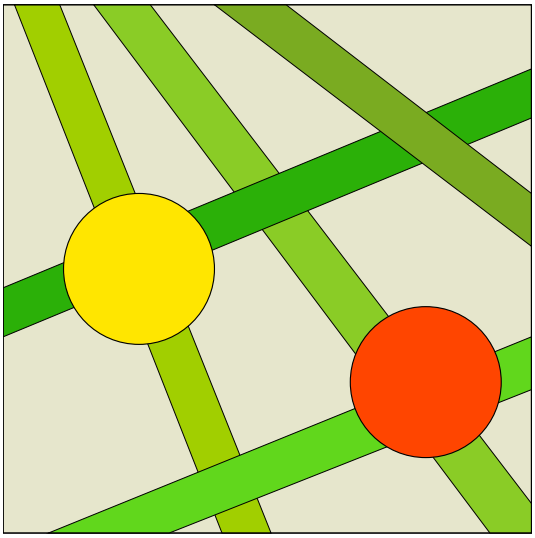
$$r = \frac{a}{2\cos\beta}$$

$$c = \frac{p+q}{2} + \lambda(q-p)^*$$

con $\lambda > 0$ tale che $\lambda|(q-p)^*| = r\sin\beta$. Siccome $|(q-p)^*| = |q-p| = a$, ciò significa $\lambda a = r\sin\beta$, ossia $\lambda \cdot 2r\cos\beta = r\sin\beta$, cosicché $\lambda = \dfrac{\tan\beta}{2}$ e

$$c = \frac{p+q}{2} + (q-p)^* \frac{\tan\beta}{2}$$

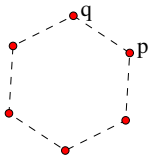
dove $(q-p)^*$ è definito come a pagina 43.



Il listato per questa figura si trova a pagina 51.

Costruzione di un poligono regolare da un suo lato

Talvolta è utile anche poter definire un poligono regolare mediante un suo lato, cioè due vertici p e q adiacenti (oltre al numero n di lati). La funzione `Geom.ptigonolato` viene usata con la sintassi `p q n Geom.ptigonolato`.



Utilizzando le formule a pagina 44 troviamo il centro con

$$c = \frac{p+q}{2} + (q-p)^* \frac{\tan\beta}{2}$$

Definiamo prima la funzione che calcola il centro del poligono:

	$p\ q\ n$
`R3`	$n\ p\ q$
`2 copy`	$n\ p\ q\ p\ q$
`0.5 S Geom.xty`	$n\ p\ q, m = (p+q)/2$
`SUT`	$n\ m\ q\ p$
`Vett.sub`	$n\ m, q-p$
`Geom2.90`	$n\ m, v = (q-p)^*$
	% Vogliamo $m + v \cdot (\tan\beta)/2$
`RC3`	$m\ v\ n$
`90 180`	$m\ v\ n\ 90\ 180$
`RC3`	$m\ v\ 90\ 180\ n$
`div`	$m\ v\ 90\ 180/n$
`sub`	$m\ v, \beta = 90 - 180/n$
`tangradi`	$m\ v, t = \tan\beta$ % Pagina 46.
`2 div`	$m\ v, t/2$
`Geom.mult`	$m\ w = v \cdot (\tan\beta)/2$
`Vett.add`	$m + w$

```
/Geom.centrodelpoligono {R3 2 copy 0.5 S Geom.xty
  SUT Vett.sub Geom2.90 RC3 90 180
  RC3 div sub tangradi 2 div Geom.mult Vett.add} def
```

Adesso otteniamo `Geom.ptigonolato` con

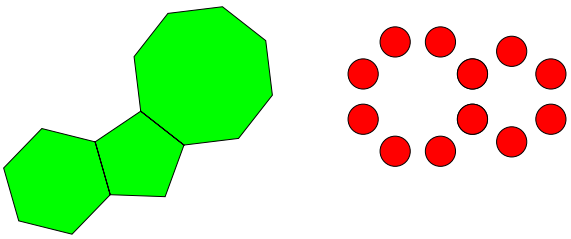
	$p\ q\ n$
`TU`	$p\ q\ n\ p$
`PU`	$p\ q\ n\ p\ n$
`5 2 roll`	$p\ n\ p\ q\ n$
`Geom.centrodelpoligono`	$p\ n\ c$
`R3`	$c\ p\ n$
`S`	$c\ n\ p$
`TU`	$c\ n\ p\ c$
`Vett.sub`	$c\ n, p - c$
`S`	$c, p - c, n$
`Geom.ptigonocentro`	

```
/Geom.ptigonolato {TU PU 5 2 roll Geom.centrodelpoligono
  R3 S TU Vett.sub S Geom.ptigonocentro} def
```

Possiamo così definire

```
/gonolato {Geom.ptigonolato poligonoc} def
```

La funzione `gonolato` è particolarmente utile quando si vogliono disegnare poligoni regolari con lati in comune:



```
impostagrafica
/p [16 7] def /q [14 14] def
/ptipentagono q p 5 Geom.ptigonolato def
/s ptipentagono 3 get def
/r ptipentagono 4 get def
p q 6 gonolato q p 5 gonolato r s 8 gonolato
rverde ---

/u [64 17] def /v [64 23] def
/sinistra u v 8 Geom.ptigonolato def
/destra v u 6 Geom.ptigonolato def
/punti sinistra destra + def
punti {2 cerchio rrosso ---} forall
```

Suddivisione di un intervallo

Per rappresentare grafici di funzioni e curve parametrizzate abbiamo bisogno di vettori che consistono di punti $x_0 = a, x_1, \ldots, x_n = b$ di un intervallo $[a, b]$ con $x_k = x_0 + kdx$ e $dx = (b-a)/n$. Nella funzione `::` creiamo prima il vettore $[0\ 1\ \ldots n]$ a cui applichiamo, mediante `Vett.map`, la funzione $\bigcirc_k\ a + kdx$. Siccome dx è variabile, dobbiamo usare la tecnica di programmazione funzionale vista a pagina 32.

	$a\ b\ n$
`0`	$a\ b\ n\ 0$
`PU`	$a\ b\ n\ 0\ n$
`Alg.intervallo`	$a\ b\ n, v = [0\ 1 \ldots n]$
`R4`	$v\ a\ b\ n$
`S`	$v\ a\ n\ b$
`TU`	$v\ a\ n\ b\ a$
`sub`	$v\ a\ n, b - a$
`S`	$v\ a, b - a, n$
`div`	$v\ a, dx = (b-a)/n$
`[S]`	$v\ a\ [dx]$
`{mul} +`	$v\ a, [dx\ \texttt{mul}]$
`S [S]`	$v, [dx\ \texttt{mul}], [a]$
`+`	$v\ [dx\ \texttt{mul}\ a]$
`{add} + cvx`	$v\ \{dx\ \texttt{mul}\ a\ \texttt{add}\}$
`Vett.map`	$[a\ a+dx\ a+2dx \ldots a+ndx]$

```
/:: {0 PU Alg.intervallo R4 S TU sub S div [S] {mul} +
  S [S] + {add} + cvx Vett.map} def
```

PostScript è molto impreciso numericamente, per cui talvolta con operazioni semplici vengono generati numeri decimali molto lunghi; per questa ragione definiamo anche una funzione `:` che esegue la suddivisione con un arrotondamento a 2 cifre decimali.

```
/: {0 PU Alg.intervallo R4 S TU sub S div [S] {mul} +
  S [S] + {add 100 mul round 100 div} + cvx Vett.map} def
```

Definiamo anche una funzione di arrotondamento generica da usare con la sintassi `x fattore Alg.arrotonda` , dove x è il numero da arrotondare, `fattore` un fattore che è uguale a 100 se vogliamo arrotondare a 2 cifre decimali (esercizio 78).

```
/Alg.arrotonda {U R3 mul round S div} def
```

Questa funzione può essere utilizzata per arrotondare i risultati di una funzione,

```
1 7 Alg.intervallo {sqrt} Vett.map
{1000 Alg.arrotonda} Vett.map ==
% [1.0 1.414 1.732 2.0 2.236 2.449 2.646]
```

oppure insieme a `::` quando non vogliamo usare l'impostazione a due cifre decimali di `:` come ad esempio in

```
0 1 7 : ==
% [0.0 0.14 0.29 0.43 0.57 0.71 0.86 1.0]

0 1 7 :: {1000 Alg.arrotonda} Vett.map ==
% [0.0 0.143 0.286 0.429 0.571 0.714 0.857 1.0]
```

Disegnare il grafico di una funzione

Per disegnare il grafico di una funzione $f : \mathbb{R} \longrightarrow \mathbb{R}$ creiamo prima con : (oppure ::) il vettore $x = [x_0 \dots x_n]$ dei punti in cui vogliamo calcolare la funzione. Da essi otteniamo un vettore di punti $[[x_0\ f(x_0)] \dots [x_n\ f(x_n)]]$ nel modo seguente:

	$x\ f$
PU	$x\ f\ x$
R3	$x\ x\ f$
Vett.map	$x\ [f(x_0) \dots f(x_n)]$
Vett.coppie	$[[x_0\ f(x_0)] \dots [x_n\ f(x_n)]]$

```
/Fun.ptigrafico {PU R3 Vett.map Vett.coppie} def
```

Adesso è sufficiente applicare `poligono`:

```
/grafico {Fun.ptigrafico poligono} def
```

Se a questo punto vogliamo disegnare il grafico di $\sin x$ per $x \in [-2\pi, 2\pi]$, incontriamo però due piccole difficoltà:

In primo luogo nelle funzioni trigonometriche di PostScript l'angolo viene indicato in gradi e quindi dobbiamo moltiplicare l'argomento con $180/\pi$ per ottenere le funzioni matematiche a cui siamo abituati:

```
/cosmat {180dpi mul cos} def

/sinmat {180dpi mul sin} def

/tanmat {180dpi mul tangradi} def
```

Definiamo anche alcune costanti trigonometriche, tenendo conto del fatto che PostScript nell'aritmetica a virgola mobile apparentemente lavora solo con 6 cifre significative:

```
/pi 3.141592 def

/2pi 6.283185 def

/4pi 12.56637 def

/pid180 0.017453 def

/180dpi 57.29578 def % Corretto sarebbe 57.2957795.

/numeroe 2.71828 def
```

Oltre a ciò, spesso dobbiamo ingrandire (o rimpicciolire) una figura; l'uso di `scale` modifica però anche lo spessore della matita. Definiamo perciò una funzione `ingrandisci` in cui questo effetto è compensato:

```
/ingrandisci {U scale 1 spessore} def
```

Si noti che `ingrandisci` effettua un ingrandimento relativo.

Per determinare l'ingrandimento attuale dobbiamo prima calcolare l'ingrandimento formale come la radice quadrata del valore assoluto del determinante della matrice corrente:

```
/ingrandimentoformale {5 dict begin
  /M matrix currentmatrix def /a M 0 get def
  /b M 1 get def /c M 2 get def /d M 3 get def
  a d mul b c mul sub abs sqrt end} def

/ingrandimento {ingrandimentoformale
  ingrandimentoiniziale div} def
```

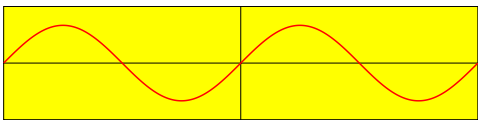
Abbiamo modificato anche `impostagrafica` per poter calcolare l'ingrandimento iniziale:

```
/impostagrafica {2.83 2.83 scale
  /ingrandimentoiniziale ingrandimentoformale def
  0.1 setlinewidth} def
```

Definiamo quindi una funzione `spessore` per impostare lo spessore della matita; a differenza da `ingrandisci` essa definisce lo spessore assoluto della matita.

```
/spessore {0.1 mul ingrandimento div setlinewidth} def
```

Adesso finalmente possiamo rappresentare il grafico di $\sin x$:



```
impostagrafica
5 ingrandisci 8 1.6 translate
/b 2pi def /a b neg def
[0 0] b 2 mul 3 0 rettangolocr rgiallo clip ---
8 2 assi ---
a b 400 : {sinmat} grafico $ rosso 2 spessore --- *
```

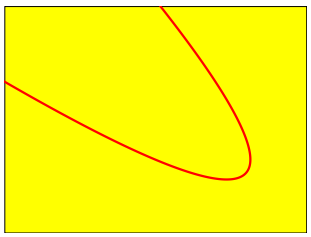
Studiare con attenzione ogni singola istruzione.

Introduciamo ancora due funzioni ausiliare: `Geom.lun` calcola la lunghezza $|v|$ di un vettore $v \in \mathbb{R}^n$ e `tangradi` (usata a pagina 45 e in `tanmat`) la tangente di un angolo espresso in gradi (non esiste una funzione `tan` in PostScript):

```
/Geom.lun {U Geom.scalare sqrt} def

/tangradi {U sin S cos div} def
```

Parabole

Otteniamo una parabola semplicemente come grafico della funzione $\bigcirc_x^{\ x^2}$, eventualmente dopo traslazione e rotazione.



```
impostagrafica 20 0 translate
40 30 rettangolo rgiallo clip ---
4 ingrandisci
$ 8 2 translate 50 rotate
-4 4 400 : {U mul} grafico 3 spessore rosso --- *
```

La funzione exp di PostScript

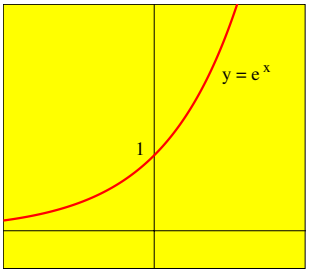
La funzione esponenziale verrà introdotta a pagina 49; essa appare anche nella definizione delle funzioni iperboliche a pagina 47.

In PostScript `exp` viene usata con la sintassi `b x exp` con b reale ≥ 0 ed $x \in \mathbb{R}$ che corrisponde al numero reale b^x. Non esiste una funzione apposita per l'esponenziale della matematica; definiamo perciò una funzione

```
/esp {2.71828 S exp} def
```

Come sempre la precisione non è esaltante:

```
10 ln esp ==
% 9.99998569
```



Octobrina elegans

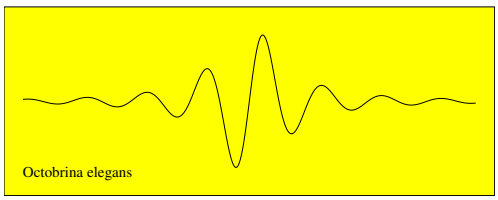
Rappresentiamo il grafico della funzione

$$\bigcirc_x \frac{2\sin 4x}{1+x^2}$$

a cui in PostScript diamo il nome `Oct`:

```
/Oct {U 4 mul sinmat 2 mul S U mul 1 add div} def
```

nell'intervallo $[-6, 6]$



mediante le istruzioni

```
impostagrafica
5 ingrandisci 8 2 translate
[0 0] 13 5 0 rettangolocr rgiallo clip ---
-6 6 400 : {Oct} grafico ---
0.4 Font.times
-6 -2 moveto (Octobrina elegans) show
```

Il grafico dell'octobrina è così elegante perché presenta una simmetria non perfetta. Se a prima vista la curva può sembrare simmetrica, guardando più attentamente vediamo invece che le due metà si distinguono nel segno; infatti questa funzione è dispari - una funzione f si chiama *dispari*, se

$$f(-x) = -f(x)$$

- perché il seno è una funzione dispari e il quadrato una funzione pari; d'altra parte il fattore smorzante $\frac{1}{1+x^2}$ riduce l'ampiezza della curva quando $|x|$ diventa più grande e attenua così la differenza tra le due parti.

Combinando la funzione ad altre funzioni elementari abbiamo ottenuto i grafici a pagina 48 che corrispondono alle seguenti funzioni di t, con $f = \bigcirc_t \frac{2\sin 4t}{1+t^2}$:

O. geminata	$f^2(t)$
O. montuosa	$f(t) + \cos t$
O. voraginosa	$f(t) + \log(1+t^2)$
O. irregularis	$f(t) + f(t^2)$
O. divisa	$f(t) - f(t^2)$
O. pulcherrima	$f(t)f(t^2)$
O. sellulata	$f(\cos t)$
O. munita	$\frac{2f(t)}{1+f^2(t)}$
O. modulata	$f(\sin t + \cos t)$
O. turrita	$f(t^2) + \cos t$
O. tortuosa	$f(t^2) + \sin t$
O. repentina	$f(t^2) + \tanh t$
O. solitaria	$0.7(f(t) + f(t^2) + f(t^3))$
O. sinuosa	$f(t) + f(t+1) + f(t+2)$
O. assurgens	$f(t) + \tanh t$
O. simplex	$f(\tanh t)$
O. laboriosa	$f(t) + (\tanh t)(\cos t)$
O. tranquilla	$f(t) + (\tanh t)(\sin t)$

La prima serie delle Octobrinidae a pagina 48 è stata ottenuta con

```
impostagrafica
5 ingrandisci 1.5 0 translate 0.5 Font.times
13 46 rettangolo rgiallo clip 1.5 spessore ---
/x -6 6 400 : def
$ 6.5 42.5 translate 0.9 ingrandisci 2 spessore
[{OctGeminata} {OctMontuosa 0.3 add}
  {OctVoraginosa 0.3 sub} {OctIrregularis}
  {OctDivisa 0.8 add} {OctPulcherrima}
  {OctSellulata} {OctMunita} {OctModulata}]
  {x S grafico --- 0 -5.5 translate} forall *
$ 6.5 0 translate
5.1 40.9 7 : {-6.5 PU moveto 6.5 S lineto ---} forall *
0.4 41.4 8 : [(Octobrina modulata) (Octobrina munita)
  (Octobrina sellulata) (Octobrina pulcherrima)
  (Octobrina divisa) (Octobrina irregularis)
  (Octobrina voraginosa) (Octobrina montuosa)
  (Octobrina geminata)] Vett.coppie
{aload T 1 RC3 moveto show} forall
```

Si noti come, ad esempio in `{OctMontuosa 0.3 add}`, sono stati effettuati piccoli spostamenti del grafico.

Funzioni iperboliche

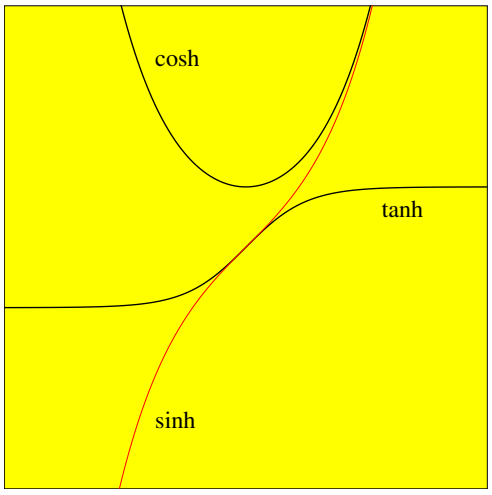
Le funzioni iperboliche cosh, sinh e tanh sono definite da

$$\cosh x := \frac{e^x + e^{-x}}{2}$$

$$\sinh x := \frac{e^x - e^{-x}}{2}$$

$$\tanh x := \frac{\sinh x}{\cosh x} = \frac{e^{2x} - 1}{e^{2x} + 1}$$

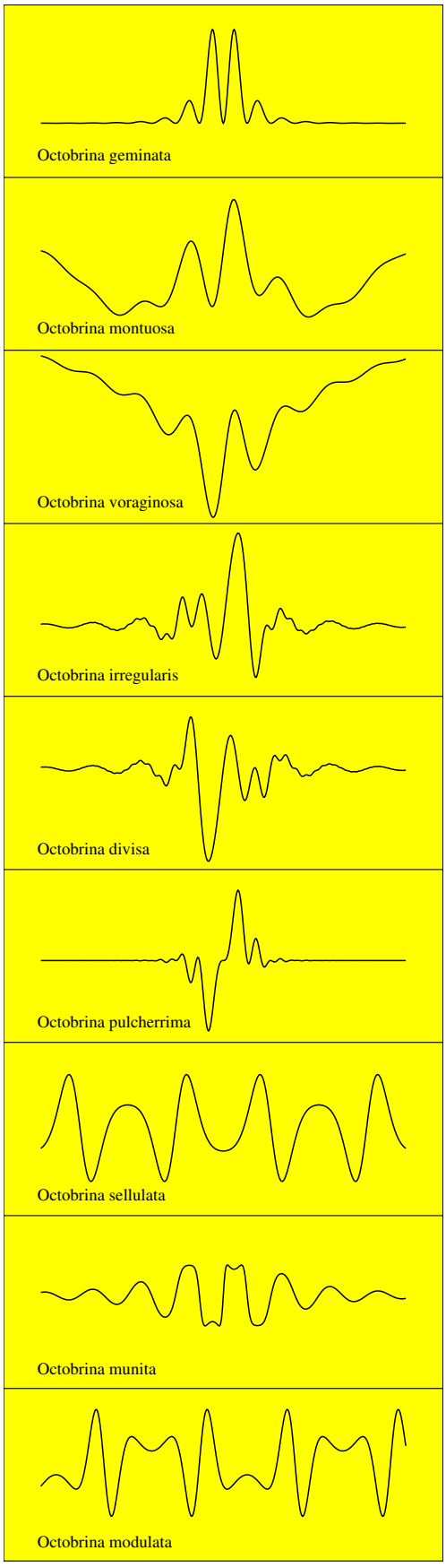
Tutte e tre hanno importanti applicazioni tecniche; la tangente iperbolica tanh è anche nota sotto il nome di *funzione sigmoidea* e viene spesso utilizzata per modellare la crescita di popolazioni (*crescita logistica*) o la formazione di impulsi, ad esempio nelle *reti neuronali*. Abbiamo utilizzato la tangente iperbolica in alcune Octobrinidae.



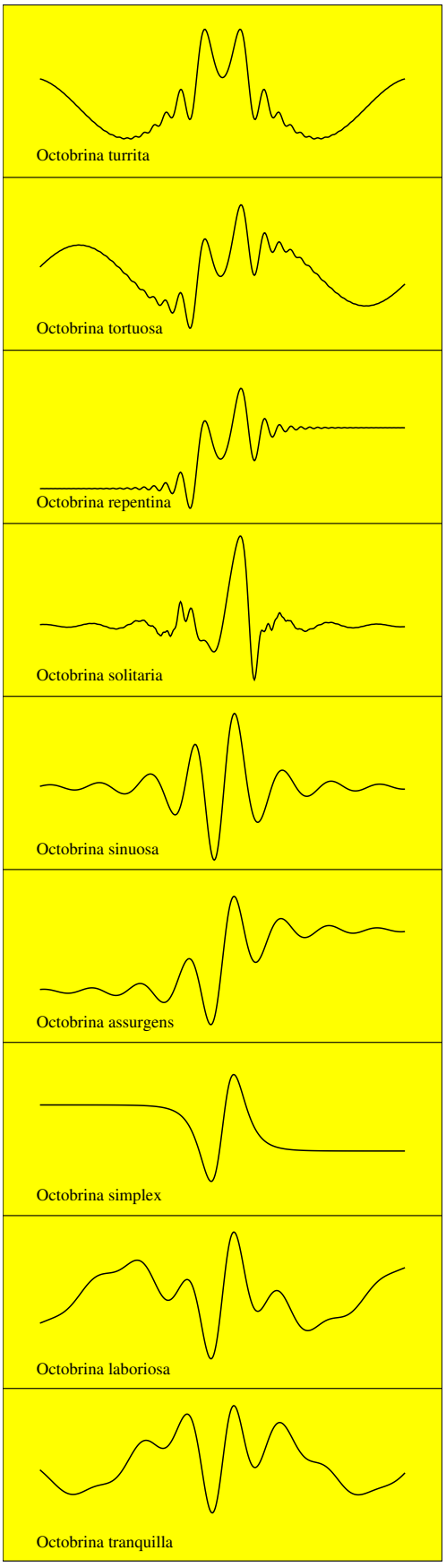
La figura è stata ottenuta con

```
impostagrafica 10 0 translate
8 ingrandisci
8 8 rettangolo rgiallo clip ---
$ /x -4 4 500 : def 4 4 translate
x {cosh} grafico x {tanh} grafico 1.8 spessore ---
1 spessore x {sinh} grafico rosso --- *
0.4 Font.times 2.5 7 moveto (cosh) show
6.25 4.5 moveto (tanh) show
2.5 1 moveto (sinh) show
```

Octobrinidae I



Octobrinidae II



La derivata

$f : U \longrightarrow \mathbb{R}$ sia una funzione a valori reali definita su un aperto U di $\mathbb{R}$ (quasi sempre U sarà un intervallo aperto, anche infinito, di $\mathbb{R}$ oppure un'unione di un numero finito di intervalli aperti). Sia $x_0 \in U$. Se il limite

$$f'(x_0) := \lim_{h \longrightarrow 0} \frac{f(x_0 + h) - f(x_0)}{h}$$

esiste, allora si dice che la funzione f è differenziabile in x_0 ed $f'(x_0)$ si chiama la derivata di f in x_0.

Segue abbastanza direttamente dalla definizione che, se anche g è una funzione definita in U e se esistono le derivate $f'(x_0)$ e $g'(x_0)$, allora anche $(f+g)'(x_0)$ esiste e si ha

$$(f+g)'(x_0) = f'(x_0) + g'(x_0)$$

Si dimostra inoltre che esiste anche $(fg)'(x_0)$ e che

$$(fg)'(x_0) = f'(x_0)g(x_0) + f(x_0)g'(x_0)$$

Qui fg è la funzione $\underset{x}{\bigcirc} f(x)g(x)$, da non confondere con la composizione delle due funzioni. Se D è l'insieme dei punti in cui f e g sono entrambe differenziabili, possiamo considerare f' e g' come funzioni definite su D e scrivere più brevemente

$$(f+g)' = f' + g'$$
$$(fg)' = f'g + fg'$$

Per una funzione costante $f(x_0 + h)$ e $f(x_0)$ coincidono, e vediamo che la derivata di una funzione costante è uguale a zero.

Consideriamo la funzione identica. In questo caso il limite da studiare è

$$\lim_{h \longrightarrow 0} \frac{(x_0 + h) - x_0}{h} = \lim_{h \longrightarrow 0} \frac{h}{h} = 1$$

perciò la funzione identica id ha derivata 1 in ogni punto.

Assumiamo adesso che la f sia differenziabile in x_0 e consideriamo le funzioni f^2, f^3 e f^4. Per la regola del prodotto abbiamo

$$(f^2)' = f'f + ff' = 2ff'$$
$$(f^3)' = f'f^2 + f(f^2)' = f'f^2 + f \cdot 2ff' = 3f'f^2$$
$$(f^4)' = f'f^3 + f(f^3)' = f'f^3 + f \cdot 3f'f^2 = 4f'f^3$$

Intravvediamo la formula generale

$$(f^n)' = nf'f^{n-1}$$

che infatti si dimostra facilmente per ogni $n \in \mathbb{N}$ con induzione. Se poniamo f uguale all'identità otteniamo la derivata di un monomio

$$(\mathrm{id}^n)' = n\,\mathrm{id}^{n-1}$$

che, in modo meno preciso ma più intuitivo, talvolta è scritta nella forma

$$(x^n)' = nx^{n-1}$$

Per un numero reale a abbiamo

$$(af)' = af'$$

perché la derivata della funzione costante $\underset{x}{\bigcirc} a$ è uguale a zero, come abbiamo visto.

Ciò mostra, insieme alla regola per la somma, che la formazione della derivata è un operatore lineare e ci permette di calcolare la derivata di una funzione polinomiale. Sia infatti

$$f = \underset{x}{\bigcirc} a_0 + a_1x + a_2x^2 + \cdots + a_nx^n$$

Usando la formula per la derivata di un monomio e la linearità otteniamo

$$f' = \underset{x}{\bigcirc} a_1 + 2a_2x + \cdots + na_nx^{n-1}$$

Consideriamo adesso il quoziente $\varphi := \frac{f}{g}$ di due funzioni differenziabili, dove restringiamo U in modo tale che non contenga zeri di g. Siccome $\varphi g = f$, dalla regola del prodotto abbiamo $\varphi' g + \varphi g' = f'$ e quindi

$$\left(\frac{f}{g}\right)' = \frac{f' - \varphi g'}{g} = \frac{f' - \frac{f}{g}g'}{g} = \frac{f'g - fg'}{g^2}$$

Nei corsi di Analisi si dimostra (e la dimostrazione richiede un po' di attenzione!) la *regola della catena*, molto importante, per la *composizione* di due funzioni differenziabili:

$$(g \circ f)'(x_0) = f'(x_0) \cdot g'(f(x_0))$$

che in modo astratto può essere scritta anche così (nel dominio dove tutto è definito):

$$(g \circ f)' = f' \cdot (g' \circ f)$$

La funzione esponenziale

Esiste una funzione che è la derivata di se stessa? Nei corsi di Analisi si impara che una tale funzione, cioè una funzione φ con $\varphi' = \varphi$ esiste veramente e che è unica se chiediamo che $\varphi(0) = 1$. Questa funzione si chiama la funzione *esponenziale* e viene denotata con exp. Essa può essere definita per ogni numero reale - in verità può essere estesa su tutto $\mathbb{C}$, è quindi una funzione

$$\exp : \mathbb{R} \longrightarrow \mathbb{R}$$

oppure, rivelando allora la sua vera natura e la profonda parentela con le funzioni trigonometriche, una funzione

$$\exp : \mathbb{C} \longrightarrow \mathbb{C}$$

exp è quindi una soluzione dell'equazione differenziale

$$\varphi' = \varphi$$

e se con D denotiamo l'operatore (di cui sappiamo che è lineare) di differenziazione (che è definito su un qualche spazio vettoriale di funzioni di cui la funzione esponenziale fa parte), vediamo che exp è soluzione dell'equazione lineare

$$D\varphi = \varphi$$

e quindi un un punto fisso dell'operatore lineare D e non ci sorprende che, come si verifica facilmente, l'insieme delle soluzioni forma un sottospazio vettoriale in quello spazio di funzioni. Si dimostra anzi che le soluzioni formano una retta e che quindi le funzioni che coincidono con la propria derivata sono esattamente i multipli scalari $a \exp$ della funzione esponenziale. Per $a = 0$ otteniamo la funzione costante zero che effettivamente anch'essa è la derivata di se stessa.

Sia $\varphi = a \exp$. Allora $\varphi(0) = a$ e vediamo che ogni soluzione della nostra equazione differenziale può essere scritta nella forma

$$\varphi = \varphi(0) \cdot \exp$$

Si dimostra nei corsi di Analisi che $\exp(x)$ coincide con e^x, dove il numero e (detto anche base del logaritmo naturale) è definito da

$$e = \lim_{n \longrightarrow \infty} \left(1 + \frac{1}{n}\right)^n = 2.718281828459\ldots$$

Si noti la simmetria nelle sette cifre (8281828) che seguono 2.71. Purtroppo PostScript ci permette solo di lavorare con 2.71828.

Il legame con l'analisi complessa è dato dalla formula di Euler,

$$e^{iz} = \cos z + i \sin z$$

valida per ogni $z \in \mathbb{C}$. Essa vale in particolare per $z = \alpha \in \mathbb{R}$; in questo caso $e^{i\alpha} = (x, y)$ è un punto sulla circonferenza unitaria con $x = \cos\alpha$, $y = \sin\alpha$.

Anche nel campo complesso valgono le relazioni $\cos(-z) = \cos z$ e $\sin(-z) = -\sin z$; da esse si trovano le formule che esprimono $\cos z$ e $\sin z$ in termini di e^{iz} e e^{-iz} già viste a pagina 30.

Curve piane parametrizzate

Una *curva parametrizzata* in $\mathbb{R}^n$ è un'applicazione $\varphi : I \longrightarrow \mathbb{R}^n$, dove I è un intervallo (aperto o chiuso, finito o infinito, ma più spesso finito e chiuso) di $\mathbb{R}$. Normalmente si chiede che la φ sia almeno continua o anche che sia due volte differenziabile con derivate continue. Lo studio delle curve continue rientra nel campo della *topologia* e riguarda in primo luogo proprietà di deformabilità di oggetti geometrici (ci sono applicazioni in robotica: ad esempio la questione, se i bracci di un robot possano o no passare in modo continuo da una posizione a un'altra, può essere formulata e trattata con gli strumenti della topologia), lo studio delle curve e superficie differenziabili fa parte della *geometria differenziale*.

Quando n è uguale a 2, si parla di curve piane. In questo caso, 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

$$x = t$$
$$y = f(t)$$

Definiamo una funzione `Geom.pticurva`, simile a `Fun.ptigrafico`, che, utilizzata nella forma t $[x\ y]$ `Geom.ptigrafico` con $t = [t_0 \ldots t_n]$, crea il vettore $[[x(t_0)\ y(t_0)] \ldots [x(t_n)\ y(t_n)]]$. Attenzione: Stavolta $[x\ y]$ non è un punto del piano, ma una coppia di funzioni.

	t $[x\ y]$
`aload T`	t x y
`TU`	t x y t
`S`	t x t y
`Vett.map`	t x $y(t)$ % $y(t)$ è il vettore degli $y(t_i)$.
`R3`	$y(t)$ t x
`Vett.map`	$y(t)$ $x(t)$
`S`	$x(t)$ $y(t)$
`Vett.coppie`	$[[x(t_0)\ y(t_0)] \ldots [x(t_n)\ y(t_n)]]$

```
/Geom.pticurva {aload T TU S Vett.map R3 Vett.map
  S Vett.coppie} def
```

Di nuovo adesso è sufficiente applicare `poligono`:

```
/curva {Geom.pticurva poligono} def
```

La funzione `Geom.pticurva` può essere anche utilizzata per posizionare altre figure lungo una curva.

Iperboli

Siano $a, b, t \in \mathbb{R}$ con $a, b > 0$ ed $x = a\cosh t$, $y = b\sinh t$. Allora

$$\frac{x^2}{a^2} - \frac{y^2}{b^2} = 1$$

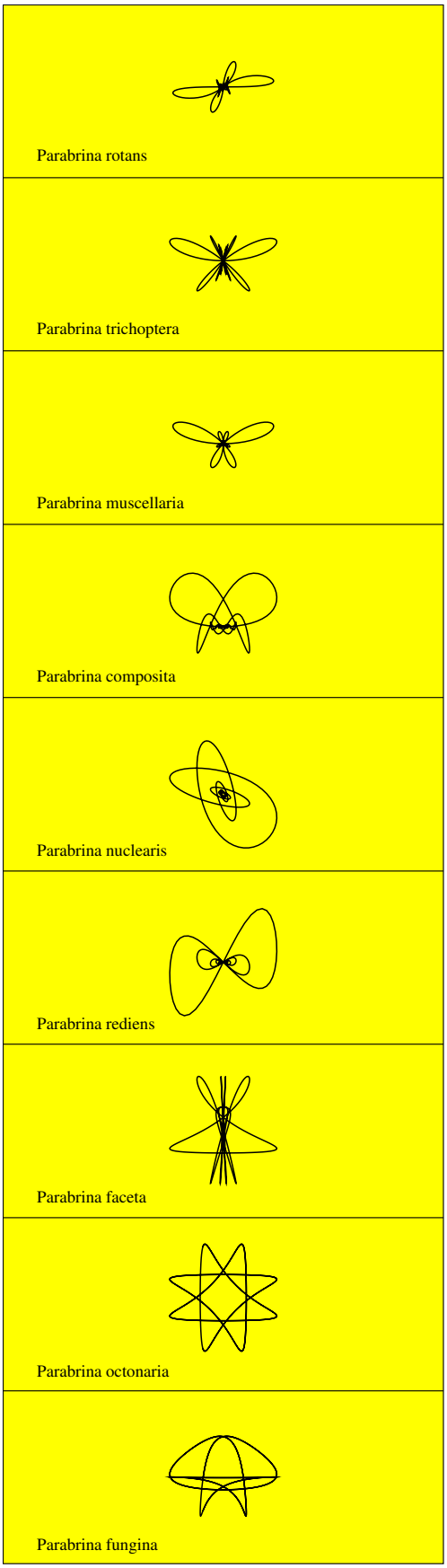
e vediamo che il punto (x, y) si trova su un'iperbole di asse reale a e asse immaginaria b. Si può dimostrare che viceversa ogni punto del ramo destro dell'iperbole può essere rappresentato in questa forma. In altre parole,

$$x = a\cosh t$$
$$y = b\sinh t$$

con $-\infty < t < \infty$ è una parametrizzazione del ramo destro dell'iperbole in forma normale. Per ottenere il ramo sinistro si può ad esempio sostituire a con $-a$. Per $a = b$ otteniamo un'iperbole equilatera con equazione

$$x^2 - y^2 = a^2$$

Parabrinidae



Parametrizzazione delle Parabrinidae

Le curve della famiglia delle *Parabrinidae* si ottengono utilizzando la funzione

$$f = \bigcirc_t \frac{2\sin 4t}{1+t^2}$$

dell'Octobrina in entrambe le coordinate della parametrizzazione:

P. rotans	$x = (\cos t) f(t)$
	$y = (\sin t^2) f(t)$
P. trichoptera	$x = f(t)$
	$y = t f(t)$
P. muscellaria	$x = (\cos t) f(t)$
	$y = (\sin t) f(t)$
P. composita	$x = f(t)$
	$y = f(t^2)$
P. nuclearis	$x = f(t)$
	$y = f(t-1)$
P. rediens	$x = f(t)$
	$y = f(2t)$
P. faceta	$x = f(t)$
	$y = f(\cos t)$
P. octonaria	$x = f(\cos t)$
	$y = f(\sin t)$
P. fungina	$x = f(\cos t)$
	$y = f(1+\sin t)$

Di queste funzioni solo le ultime due sono periodiche, perciò, nonostante le apparenze, le prime sette curve non sono chiuse e ogni allargamento dell'intervallo di definizione ne cambierebbe l'aspetto, anche se i cambiamenti diventano impercettibili per l'influsso smorzante del fattore $\frac{1}{1+x^2}$ contenuto in f.

Le figure a pagina 50 sono state ottenute con

```
impostagrafica
5 ingrandisci 1.5 0 translate 0.5 Font.times
13 46 rettangolo rgiallo clip 1.5 spessore ---
/t -6 6 400 : def
$ 6.5 42.5 translate 0.9 ingrandisci 2 spessore
[ParaRotans ParaTrichoptera ParaMuscellaria
ParaComposita ParaNuclearis ParaRediens
ParaFaceta ParaOctonaria ParaFungina]
  {t S curva --- 0 -5.5 translate} forall *
$ 6.5 0 translate
5.1 40.9 7 : {-6.5 PU moveto 6.5 S lineto ---} forall *
0.4 41.4 8 : [(Parabrina rotans) (Parabrina trichoptera)
(Parabrina muscellaria) (Parabrina composita)
(Parabrina nuclearis) (Parabrina rediens)
(Parabrina faceta) (Parabrina octonaria)
(Parabrina fungina)] Vett.inv Vett.coppie
{aload T 1 RC3 moveto show} forall
```

Ogni Parabrinida corrisponde a una coppia $[x\ y]$ di funzioni, ad esempio

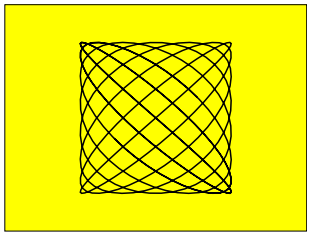
```
/ParaRediens [{Oct} {2 mul Oct}] def
```

Figure di Lissajous

Le figure di Lissajous sono curve piane con una parametrizzazione della forma

$$x(t) = \sin(mt+a)$$
$$y(t) = \sin(nt+b)$$

per $m, n, a, b \in \mathbb{R}$.



```
impostagrafica 20 0 translate
40 30 rettangolo rgiallo clip ---
10 ingrandisci 2 1.5 translate
/x {8 mul sinmat} def /y {7 mul sinmat} def
/t -4 4 400 : def
t [{x} {y}] curva 2 spessore ---
```

Le curve di Lissajous corrispondono, come si vede dalla parametrizzazione, a oscillazioni rispetto a due assi ortogonali tra di loro (l'asse delle x e l'asse delle y). È facile rappresentarle tramite un oscilloscopio. Esse descrivono anche il moto senza attrito di una pallina in una conca ellittica, cioè tale che le curve di livello siano ellissi.

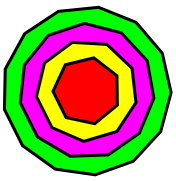
Il semplice listato dell'ultima figura a pagina 44:

```
impostagrafica 6 0 translate
/colora {$ setrgbcolor fill *} def /pc {poligonoc} def
70 70 rettangolo 0.9 0.9 0.8 colora clip ---
[[29 0] [35.5 0] [7.5 70] [1.5 70]] pc 0.63 0.81 0 colora ---
[[12 70] [19.5 70] [73 0] [64.5 0]] pc 0.54 0.8 0.15 colora ---
[[6 0] [22 0] [70 19] [70 26]] pc 0.38 0.84 0.11 colora ---
[[70 61.5] [70 55] [0 26] [0 32.5]] pc
  0.17 0.69 0.04 colora ---
[[28 70] [37.5 70] [70 45] [70 38]] pc
  0.48 0.67 0.13 colora  ---
1.5 spessore [18 35] 10 cerchio 1 0.9 0 colora ---
[56 20] 10 cerchio 1 0.27 0 colora ---
70 70 rettangolo ---
```

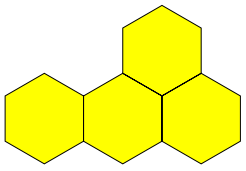
Esercizi per gli scritti

75. La rotazione di (x, y) per l'angolo α nel piano corrisponde alla moltiplicazione (nel campo complesso) di $x + iy$ con $\cos\alpha + i\sin\alpha$.

76.



77.



78. Analisi operazionale per `Alg.arrotonda`. Spiegare argomenti, modo di operare e risultato della funzione.

79. Analisi operazionale per `Oct` (*Octobrina elegans*).

80. Utilizzando `Oct`, scrivere la funzione per *Octobrina sinuosa*.

81. Utilizzando `Oct`, scrivere la funzione per *Octobrina modulata*.

82. Definire `ParaRotans`.

83. Definire `ParaOctonaria`.

ALGORITMI E STRUTTURE DI DATI

Corso di laurea in matematica | Anno accademico 2005/06 | Numero 10

Gruppoidi e semigruppi

Definizione 52.1. Un *gruppoide* è una coppia (G, μ), dove G è un insieme e $\mu : G \times G \longrightarrow G$ è un'applicazione. Una tale applicazione si chiama una moltiplicazione o *composizione* su G. Spesso invece di $\mu(a, b)$ si scrive semplicemente ab o $a \cdot b$.

Sottintendendo μ, spesso si dice anche che G è un gruppoide, benché sullo stesso insieme possano esistere molte composizioni. Infatti, se G è finito e $|G| = n$, allora il numero delle composizioni su G è uguale a $|G^{G\times G}| = |G|^{|G\times G|} = n^{n^2}$.

Su un insieme con 2 elementi esistono quindi già $2^4 = 16$ composizioni, su un insieme con 3 elementi $3^9 = 19683$, su un insieme con 4 elementi 4^{16}, cioè più di 4 miliardi.

Definizione 52.2. Un *semigruppo* è un gruppoide $G = (G, \mu)$ associativo, in cui vale cioè la legge associativa

$$\mu(a, \mu(b, c)) = \mu(\mu(a, b), c)$$

per ogni $a, b, c \in G$. Abbreviando possiamo scrivere $a(bc) = (ab)c$.

Perciò in un semigruppo si può scrivere semplicemente abc per $a(bc)$ e $(ab)c$. Si vede facilmente che anche in espressioni più lunghe si possono lasciar via le parentesi, perché ogni possibile modo di metterle porterebbe allo stesso risultato:

$$((ab)((cd)d))f = (a(b(cd)))(ef)$$

per cui in un semigruppo per entrambe le espressioni si può scrivere $abcdef$.

Il concetto di semigruppo non è importante solo nella matematica pura, ma anche in alcuni campi dell'informatica, ad esempio nella teoria dei linguaggi formali, nella teoria degli automi, nell'analisi dei testi.

Definizione 52.3. Un elemento e di un gruppoide G si chiama

neutro a destra, se $ae = a$ per ogni $a \in G$,
neutro a sinistra, se $ea = a$ per ogni $a \in G$,
neutro, se è neutro sia a destra che a sinistra, cioè se $ea = ae = a$ per ogni $a \in G$.

Proposizione 52.4. *Il gruppoide G possieda un elemento neutro a destra e ed un elemento neutro a sinistra f. Allora $e = f$.*

Dimostrazione. e è neutro a destra, perciò $fe = f$.
f è neutro a sinistra, perciò $fe = e$.

Corollario 52.5. *Se un gruppoide G possiede un elemento neutro, questo è l'unico elemento neutro di G. Non esistono nemmeno altri elementi neutri solo a sinistra o solo a destra.*

Nota 52.6. Un grupppoide può invece contenere più di un elemento neutro a sinistra o più di un elemento neutro a destra. Infatti, se su un insieme qualsiasi G definiamo $ab := b$ per ogni $a, b \in G$, allora G diventa un semigruppo in cui ogni elemento è neutro a sinistra.

Definizione 52.7. Un *monoide* è un semigruppo G che possiede un elemento neutro, talvolta denotato con 1_G.

Definizione 52.8. Un'applicazione $\varphi : G \longrightarrow H$ tra gruppoidi G ed H si chiama un *omomorfismo*, se per ogni $a, b \in G$ vale $\varphi(ab) = \varphi(a)\varphi(b)$. Simbolicamente la condizione significa

$$a \longmapsto u \text{ e } b \longmapsto v \implies ab \longmapsto uv$$

Se G ed H sono semigruppi, φ si chiama anche un *omomorfismo di semigruppi*.

Da un *omomorfismo di monoidi* $\varphi : G \longrightarrow H$ si chiede invece non solo che G ed H siano monoidi e che φ sia un omomorfismo di semigruppi, ma anche che $\varphi(1_G) = 1_H$.

Un omomorfismo $G \longrightarrow G$ (G sia un qualsiasi tipo di struttura per cui è definito il concetto di omomorfismo) si chiama anche un *endomorfismo* di G.

Definizione 52.9. Un'applicazione $\varphi : G \longrightarrow H$ tra gruppoidi G ed H si chiama un *isomorfismo*, se φ è biiettiva e sia φ che φ^{-1} sono omomorfismi. Simbolicamente:

$$a \longleftrightarrow u \text{ e } b \longleftrightarrow v \implies ab \longleftrightarrow uv$$

In questo numero

Osservazione 52.10. *G ed H siano monoidi e $\varphi : G \longrightarrow H$ un isomorfismo di gruppoidi. Allora φ è anche un isomorfismo di monoidi, φ e φ^{-1} sono cioè entrambi omomorfismi di monoidi.*

Dimostrazione. Esercizio 84.

Proposizione 52.11. *$\varphi : G \longrightarrow H$ sia un omomorfismo biiettivo di gruppoidi. Allora anche φ^{-1} è un omomorfismo, quindi φ è un isomorfismo.*

Dimostrazione. Sia $\psi := \varphi^{-1}$. Siano $u, v \in H$.
Per la suriettività di φ esistono $a, b \in G$ tali che $u = \varphi(a)$, $b = \varphi(v)$, quindi $a = \psi(u), b = \psi(v)$.
φ è un omomorfismo, perciò $uv = \varphi(a)\varphi(b) = \varphi(ab)$, per cui $\psi(uv) = ab = \psi(u)\psi(v)$.

Definizione 52.12. Un gruppoide G si dice *commutativo*, se $ab = ba$ per ogni $a, b \in G$.

Proposizione 52.13. *$\varphi : G \longrightarrow H$ sia un omomorfismo suriettivo di gruppoidi.*

(1) Se G è un semigruppo, anche H è un semigruppo.
(2) Se G è commutativo, anche H è commutativo.

Dimostrazione. Esercizio 85.

Nota 52.14. Esempi di importanti gruppoidi non associativi:

(1) $(\mathbb{R}, \mu)$ con $\mu(a, b) := \dfrac{a+b}{2}$. Questa composizione non associativa, che quindi non definisce un semigruppo, può essere usata per formulare le leggi di Mendel in genetica e viene studiata nella teoria delle *algebre genetiche*.
(2) X sia un insieme non vuoto e $G := (\mathcal{P}(X), \mu)$ con $\mu(A, B) := A \setminus B$. Siccome $(A \setminus B) \setminus B = A \setminus (B \cup B) = A \setminus B$, mentre $A\setminus(B\setminus B) = A\setminus\emptyset = A$, vediamo che G non è associativo.
(3) G sia l'insieme delle matrici $n \times n$ con la moltiplicazione $\mu(A, B) := AB - BA$, dove AB è il comune prodotto di matrici. Allora (G, μ) è un gruppoide non associativo, molto importante; infatti si tratta di un'*algebra di Lie*, uno dei concetti più fondamentali della teoria dei gruppi e della geometria.
(4) $(\mathbb{R}, \mu)$ con $\mu(a, b) := \begin{cases} a - b & \text{se } a \geq b \\ 0 & \text{se } a < b \end{cases}$
Questo gruppoide non associativo viene talvolta utilizzato nell'analisi numerica astratta.

Nota 52.15. Se G è un gruppoide finito, di cardinalità non troppo grande, è spesso possibile compilare la sua *tavola di moltiplicazione*:

	a	b	...	y	...
a	a^2	ab	...	ay	...
b	ba	b^2	...	by	...
...	...	...	...	...	...
x	xa	xb	...	xy	...
...	...	...	...	...	...

Il monoide libero generato da un alfabeto

Definizione 53.1. 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.

Si può dimostrare che ogni monoide è immagine omomorfa di un monoide libero.

Teorema 53.2. *H sia un monoide e $\varphi_0 : A \longrightarrow H$ un'applicazione qualsiasi. Allora esiste esiste un unico omomorfismo di monoidi $\varphi : A^* \longrightarrow H$ che su A coincide con φ_0.*

Dimostrazione. (1) È chiaro che φ deve avere la proprietà che $\varphi(\varepsilon) = 1_H$ e

$$\varphi(a_1 \dots a_n) = \varphi(a_1) \dots \varphi(a_n) = \varphi_0(a_1) \dots \varphi_0(a_n)$$

per $a_1, \dots, a_n \in A$. Questo è quindi l'unico modo possibile per definire φ.

(2) Bisogna però dimostrare che funziona, cioè che φ in questo modo risulta ben definito. Ma ciò segue dal fatto che ogni parola $v \in A^+$ ha un'unica rappresentazione della form $v = a_1 \dots a_n$ con $a_1 \dots a_n \in A$.

(3) Dimostriamo che φ è veramente un omomorfismo di monoidi.

Per definizione vale $\varphi(\varepsilon) = 1_H$.

Siano $v = a_1 \dots a_n$, $w = b_1 \dots b_m$ con $n, m \geq 1$ ed $a_i, b_j \in A$. Allora

$$\begin{aligned}\varphi(vw) &= \varphi(a_1 \dots a_n b_1 \dots b_m) \\ &= \underbrace{\varphi_0(a_1) \dots \varphi_0(a_n)}_{\varphi(v)} \underbrace{\varphi_0(b_1) \dots \varphi_0(b_m)}_{\varphi(w)} = \varphi(v)\varphi(w)\end{aligned}$$

Dimostrare da soli che $\varphi(v\varepsilon) = \varphi(v)\varphi(\varepsilon)$ e $\varphi(\varepsilon v) = \varphi(\varepsilon)\varphi(v)$ per ogni $v \in A^*$.

A. Clifford/G. Preston: The algebraic theory of semigroups II. AMS 1967.

M. Lothaire: Combinatorics on words. Cambridge UP 1997. Monsieur Lothaire è uno pseudonimo e indica un gruppo di autori legati al seminario lorenese di calcolo combinatorio.

Sistemi di Lindenmayer

A sia un insieme finito. Gli endomorfismi di A^* sono noti allora anche come *sistemi di Lindenmayer*. Per il teorema 53.2 una qualsiasi applicazione $A \longrightarrow A^*$ determina univocamente un sistema di Lindenmayer. Siccome, come non è difficile dimostrare, A^* determina A (cioè se A e B sono due insiemi tali che $A^* = B^*$, allora $A = B$), viceversa ogni sistema di Lindenmayer determina a sua volta univocamente l'applicazione $A \longrightarrow A^*$.

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), che vedremo fra poco: Si parte da un elemento *iniziatore*, il triangolo equilatero, e da un elemento *generatore* (che contiene le leggi), costituito 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, rimpicciolita 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.

O. Deussen: Computergenerierte Pflanzen. Springer 2003.

D. Gambi: Sistemi di Lindenmayer e automi cellulari. Tesi Univ. Ferrara 1991.

R. Nibbi: Topologia dei frattali. Tesi Univ. Ferrara 1991.

P. Prusinkiewicz/A. Lindenmayer: The algorithmic beauty of plants. Springer 1990.

G. Rozenberg/A. Salomaa: The mathematical theory of L-systems. Academic Press 1980.

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:

$A = \{0, 1\}$
iniziatore: 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.

Un ramo di applicazione piuttosto recente e interessante della dinamica simbolica è l'analisi dei testi (*combinatoria delle parole*), ad esempio 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.

H. Furstenberg: Recurrence in ergodic theory and combinatorial number theory. Princeton UP 1981.

W. Gottschalk/G. Hedlund: Topological dynamics. AMS 1968.

D. Lind/B. Marcus: An introduction to symbolic dynamics and coding. Cambridge UP 1995.

A. Lanzoni: Metodi della dinamica simbolica in informatica. Tesi Univ. Ferrara 1998.

Sistemi dinamici e dinamica simbolica

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 $X = U \cup V$ di X e che

$$\begin{array}{llll} x_0 \in U & x_1 \in U & x_2 \in V & x_3 \in U \\ x_4 \in V & x_5 \in U & x_6 \in U & x_7 \in V \\ x_8 \in V & x_9 \in V & x_{10} \in V & x_{11} \in V \\ & \ldots & & \end{array}$$

Allora possiamo associare a questo movimento la successione

$$UUVUVUUVUUVV\ldots$$

o, più brevemente, $001010010011\ldots$,

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.

La funzione generale Linden.fun

Siccome per il teorema 53.2 un sistema di Lindenmayer $\varphi : A^* \longrightarrow A^*$ è dato da un'applicazione $\varphi_0 : A \longrightarrow A^*$, possiamo semplicemente usare v `{`φ_0`}` `Vett.map` per ottenere la riscrittura $\varphi(v)$ di v secondo φ_0. Per la successione di Morse possiamo definire φ_0 (o `Linden.morse`) con

```
/Linden.morse {0 eq {0 1} {1 0} ifelse} def
```

Per stampare una stringa di cifre (cioè numeri compresi tra 0 e 9) possiamo, con un semplice miglioramento della funzione `Str.dabin` vista a pagina 10, usare

```
/Linden.stampa {{48 add} Vett.map Str.davettore print
  (\n\n) print} def
```

Adesso con

```
[0] 6 {{Linden.morse} Vett.map U Linden.stampa} repeat T
```

otteniamo l'output

```
01

0110

01101001

0110100110010110

01101001100101101001011001101001

0110100110010110100101100110100110010110011010010110100110010110
```

In PostScript un sistema di Lindenmayer è dato da un vettore

$$v = [[a_1\ \alpha_1] \ldots [a_n\ \alpha_n]]$$

con $a_1, \ldots, a_n \in A$ tutti distinti, in cui ogni α_i è una successione di elementi di A (non racchiusa tra parentesi quadre) che possiamo identificare con una parola (possibilmente vuota) in A^*.

Per la successione di Morse potremme ad esempio usare

$$v = [[0\ 0\ 1]\ [1\ 1\ 0]]$$

L'ordine in cui gli elementi di v sono elencati naturalmente non ha importanza. Se un elemento di A non appare tra gli a_j, rimane invariante sotto φ_0. Quindi

$$\varphi_0(x) = \begin{cases} \alpha_i & \text{se } x = a_i \\ x & \text{se } x \neq a_i \text{ per ogni } i \end{cases}$$

Dobbiamo però trasformare v in un'applicazione. A questo scopo creiamo una funzione `Linden.fun`, da usare come in questo esempio:

```
/Morse [[0 0 1] [1 1 0]] Linden.fun def
```

Adesso `Morse` può essere utilizzata al posto di `Linden.morse`.

Per capire come la funzione `Linden.fun` deve essere definita, consideriamo i casi più semplici, scrivendo a destra in ogni riga l'applicazione che dobbiamo ottenere e tenendo conto del fatto che $\varphi_0(x)$ deve essere anche definita se x non è uno degli a_j:

$v = []$ `{}`
$v = [[a\ \alpha]]$ `{U` a `eq {T` α`} {} ifelse}`

come mostra l'analisi operazionale

	x
`U`	$x\ x$
`a eq`	$x\ (x = a)$
`{`	$\{\ x$
`T` α `}`	$\alpha\ \}$
`{`	x
`}`	$x\ \}$
`ifelse`	$\varphi_0(x)$

e, per un vettore di regole con due elementi,

$v = [[a\ \alpha]\ [b\ \beta]]$ `{U` a `eq {T` α`}`
`{U` b `eq {T` β`} {} ifelse} ifelse}`

Vediamo che `Linden.fun` può essere definita in modo ricorsivo:

$v = [[a_1\ \alpha_1] \ldots [a_n\ \alpha_n]]$ `{U` a_1 `eq {T` α_1`}`
$[[a_2\ \alpha_2] \ldots [a_n\ \alpha_n]]$ `Linden.fun`

Se continuiamo ad usare lettere greche per sequenze non racchiuse tra parentesi quadre, per

$v = [[a\ \alpha]\ \lambda]$ e $\{\varphi\} = [\lambda]$ `Linden.fun`

dobbiamo quindi arrivare a

`{U` a `eq {T` α`} {`φ`} ifelse}`

L'analisi operazionale per realizzare questo obiettivo richiede un po' di pazienza, ma non è difficile e il risultato ci ricompenserà ampiamente della nostra perseveranza.

	v
`U length`	$v\ n$
`0 eq`	$v\ (n = 0)$
`{`	$\{\ v$
`T {}}`	`{{}}`
`{`	$\{\ v$
`Vett.spezza`	$[a\ \alpha]\ [\lambda]$
`Linden.fun`	$[a\ \alpha]\ \{\varphi\}$
`[S]`	$[a\ \alpha]\ [\{\varphi\}]$
`S`	$[\{\varphi\}]\ [a\ \alpha]$
`Vett.spezza`	$[\{\varphi\}]\ a\ [\alpha]$
`{U}`	$[\{\varphi\}]\ a\ [\alpha]$ `{U}`
`RC3`	$[\{\varphi\}]\ [\alpha]$ `{U}` a
`[S]`	$[\{\varphi\}]\ [\alpha]$ `{U}` $[a]$
`+`	$[\{\varphi\}]\ [\alpha]\ [$`U` $a]$
`{eq} +`	$[\{\varphi\}]\ [\alpha]\ [$`U` a `eq]`
`{T}`	$[\{\varphi\}]\ [\alpha]\ [$`U` a `eq] {T}`
`RC3`	$[\{\varphi\}]\ [$`U` a `eq] {T}` $[\alpha]$
`+ cvx`	$[\{\varphi\}]\ [$`U` a `eq]` $\{$`T` $\alpha\}$
`[S]`	$[\{\varphi\}]\ [$`U` a `eq]` $[\{$`T` $\alpha\}]$
`+`	$[\{\varphi\}]\ [$`U` a `eq` $\{$`T` $\alpha\}]$
`S`	$[$`U` a `eq` $\{$`T` $\alpha\}]\ [\{\varphi\}]$
`+`	$[$`U` a `eq` $\{$`T` $\alpha\}\ \{\varphi\}]$
`{ifelse} + cvx}`	$\{$`U` a `eq` $\{$`T` $\alpha\}\ \{\varphi\}$ `ifelse}`
`ifelse`	... % Obiettivo.

per cui definiamo

```
/Linden.fun {U length 0 eq {T {}} {Vett.spezza Linden.fun
  [S] S Vett.spezza {U} RC3 [S] + {eq} + {T} RC3 + cvx [S] +
  S + {ifelse} + cvx} ifelse} def
```

Il metodo della tartaruga

Sistemi di Lindenmayer vengono spesso rappresentati mediante il metodo della tartaruga (a sua volta suscettibile di molte variazioni) che può essere realizzato molto facilmente utilizzando *nomi di funzioni* invece di simboli astratti come elementi di A. Una volta ottenuto come risultato della riscrittura un vettore $[/f_1 \dots /f_n]$, è sufficiente applicare il comando `{cvx exec} forall` per fare in modo che le funzioni che appaiono nel vettore vengano eseguite. Applichiamo questa idea per rappresentare graficamente in tre modi diversi una sezione iniziale finita della successione di Morse (oppure di una qualsiasi successione a due lettere).

Nel primo metodo 0 corrisponde a un trattino ad altezza zero sull'asse delle x, 1 a un trattino ad altezza 1. Con

```
impostagrafica

/zero {currentpoint T 0 lineto 1 0 rlineto} def

/uno {currentpoint T 1 lineto 1 0 rlineto} def

/Morse [[/zero /zero /uno] [/uno /uno /zero]] Linden.fun def

10 0 moveto
[/zero] 6 {{Morse} Vett.map} repeat
{cvx exec} forall ---
```

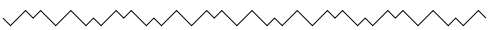
otteniamo

Nel secondo metodo 0 corrisponde a un rettangolo verde, 1 a un rettangolo rosso.

```
impostagrafica

/zero {1 0 translate $ 1 1 rettangolo rverde --- *} def

/uno {1 0 translate $ 1 1 rettangolo rrosso --- *} def

/Morse [[/zero /zero /uno] [/uno /uno /zero]] Linden.fun def

10 0 translate
[/zero] 6 {{Morse} Vett.map} repeat
{cvx exec} forall
```

Nella terza rappresentazione ogni lettera corrisponde a un movimento verso destra, in basso se la lettera è uguale a 0, in alto se è uguale a 1.

```
impostagrafica

/zero {1 -1 rlineto} def

/uno {1 1 rlineto} def

/Morse [[/zero /zero /uno] [/uno /uno /zero]] Linden.fun def

10 0 moveto
[/zero] 6 {{Morse} Vett.map} repeat
{cvx exec} forall ---
```

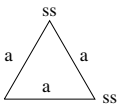


Il fiocco di neve di Koch

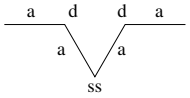
Consideriamo le seguenti semplici operazioni grafiche:

```
/a {1 0 rlineto} def
/s {60 rotate} def
/d {-60 rotate} def
```

`a` descrive un movimento in avanti, `s` una rotazione di 60 gradi verso sinistra, `d` una rotazione di 60 gradi verso destra. Per disegnare un equilatero possiamo usare la sequenza `a s s a s s a`:



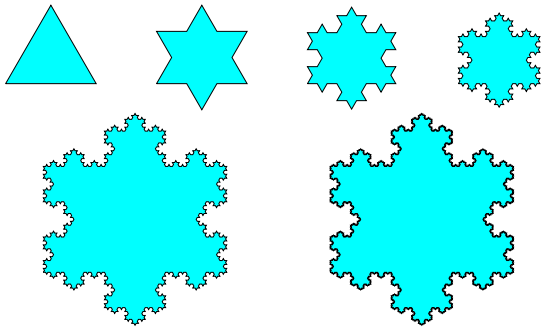
Nel fiocco di neve di Koch vogliamo adesso sostituire ogni segmento di retta con una linea spezzata la cui definizione si vede dal seguente disegno:



a d a s s a d a

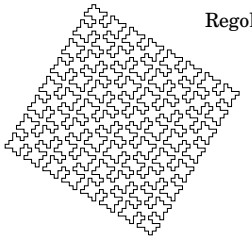
Possiamo quindi utilizzare le seguenti istruzioni:

```
impostagrafica

/a {1 0 rlineto} def
/s {60 rotate} def
/d {-60 rotate} def

% Il secondo argomento di disegna e' l'ingrandimento.
/disegna {$ ingrandisci {cvx exec} forall rciano --- *} def

/triangolo [/a /s /s /a /s /s /a] def

5 33 moveto triangolo 12 disegna

/Koch [[/a /a /d /a /s /s /a /d /a]] Linden.fun def

/prima triangolo {Koch} Vett.map def
/seconda prima {Koch} Vett.map def
/terza seconda {Koch} Vett.map def
/quarta terza {Koch} Vett.map def
/quinta quarta {Koch} Vett.map def

25 33 moveto prima 4 disegna
45 33 moveto seconda 1.3 disegna
65 33 moveto terza 0.4 disegna
10 8 moveto quarta 0.3 disegna
48 8 moveto quinta 0.1 disegna
```



Si vede che, se scegliamo gli ingrandimenti in modo tale che il diametro delle figure rimanga sempre uguale (in questo caso riducendo ogni volta la lunghezza dei singoli segmenti a un terzo di quella precedente), dopo pochi passaggi esse cambiano in modo sempre meno percebile e convergono infatti, come si può dimostrare, a una figura limite nella metrica di Hausdorff a cui abbiamo già accennato a pagina 53.

Iniziatore: $asxasasxa$
Regola: $x \longmapsto xadasadxasasaxadasadx$



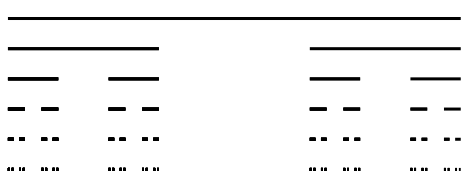
L'insieme di Cantor

Sia $X_0 := [0,1]$ l'intervallo unitario chiuso. Lo suddividiamo in tre intervalli di lunghezza uguale e togliamo l'interno dell'intervallo medio, ottenendo così l'insieme $X_1 = [0, \frac{1}{3}] \cup [\frac{2}{3}, 1]$. A ciascuno dei due intervalli rimasti applichiamo lo stesso procedimento, ottenendo

$$X_2 = [0, \tfrac{1}{9}] \cup [\tfrac{2}{9}, \tfrac{1}{3}] \cup [\tfrac{2}{3}, \tfrac{5}{9}] \cup [\tfrac{8}{9}, 1]$$

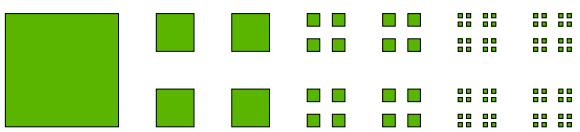
Ripetendo questa costruzione, otteniamo una successione decrescente $X_0 \supset X_1 \supset X_2 \ldots$ di sottoinsiemi di cui si può dimostrare che converge nella metrica di Hausdorff a un insieme limite $X = \bigcap_{n=0}^{\infty} X_n$ che si chiama l'*insieme di Cantor*.

```
impostagrafica 10 0 translate

% Nero.
/n {1 0 rlineto $ 4 spessore --- *} def

% Bianco.
/b {1 0 rmoveto} def

/Cantor [[/n /n /b /n] [/b /b /b /b]] Linden.fun def

/X0 [/n] def
/X1 X0 {Cantor} Vett.map def
/X2 X1 {Cantor} Vett.map def
/X3 X2 {Cantor} Vett.map def
/X4 X3 {Cantor} Vett.map def
/X5 X4 {Cantor} Vett.map def

/y 22 def /ingr 60 def
[X0 X1 X2 X3 X4 X5]
  {0 y moveto $ ingr ingrandisci
  {cvx exec} forall *
  /ingr ingr 3 div def /y y 4 sub def} forall
```



In modo simile è definito l'insieme di Cantor nel piano:

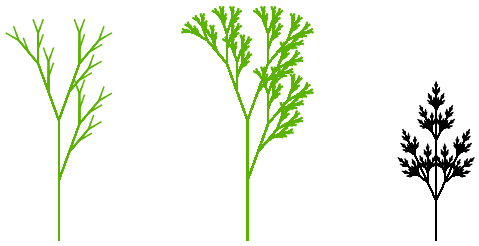
```
impostagrafica 10 10 translate

% Quadrato.
/q {$ [0 0] 3 3 0 rettangolocr
  $ 0.35 0.71 0 setrgbcolor fill * --- *} def

/riduci {1 3 div ingrandisci} def

% Sinistra in alto, sinistra in basso ecc.
/sa {-1 1 translate riduci} def
/sb {-1 -1 translate riduci} def
/da {1 1 translate riduci} def
/db {1 -1 translate riduci} def

/Cantor
  [[/q /$ /sa /q /* /$ /sb /q /* /$ /da /q /* /$ /db /q /*]]
  Linden.fun def

/X0 [/q] def
/X1 X0 {Cantor} Vett.map def
/X2 X1 {Cantor} Vett.map def
/X3 X2 {Cantor} Vett.map def

5 ingrandisci
[X0 X1 X2 X3] {$ {cvx exec} forall * 4 0 translate} forall
```



Ramificazioni

Nell'ultimo esempio si vede come le operazioni `$` e `*` possono essere utilizzate all'interno delle regole di riscrittura. Esse permettono anche di rappresentare le ramificazioni che tipicamente si osservano nelle piante, come nel seguente esempio.

```
impostagrafica 0.35 0.7 0 setrgbcolor

/a {0 1 rlineto $ 2 spessore --- *} def
/s {20 rotate} def
/d {-20 rotate} def
/g {} def % Elemento strutturale (gemma).

/Pianta [[/a /a /a]
  [/g /a /$ /d /g /* /a /$ /s /g /* /d /g]] Linden.fun def

/X4 [/g] 4 {{Pianta} Vett.map} repeat def
/X7 X4 3 {{Pianta} Vett.map} repeat def
/x 20 def /ingr 1 def
[X4 X7] {x 2 moveto $ ingr ingrandisci {cvx exec} forall *
  /ingr ingr 8 div def /x x 25 add def} forall
```



La terza figura è stata ottenuta con

```
impostagrafica

/a {0 1 rlineto $ --- *} def
/s {25 rotate} def
/d {-25 rotate} def
/g {} def % Elemento strutturale (gemma).

/Pianta [[/a /a /a]
  [/g /a /$ /s /g /* /$ /d /g /* /a /g]] Linden.fun def

/X [/g] 7 {{Pianta} Vett.map} repeat def
70 2 moveto 1 12 div ingrandisci
X {cvx exec} forall}
```

Qui, come già nell'ultima figura a pagina 55, abbiamo usato *elementi strutturali* che non corrispondono a operazioni grafiche, ma a strutture dell'organismo che si vuole disegnare. Si possono introdurre elementi che descrivono colori, età o fasi di sviluppo, crescita nello spazio ecc.

Esercizi per gli scritti

84. G ed H siano monoidi e $\varphi : G \longrightarrow H$ un isomorfismo di gruppoidi. Allora φ è anche un isomorfismo di monoidi, φ e φ^{-1} sono cioè entrambi omomorfismi di monoidi.

85. $\varphi : G \longrightarrow H$ sia un omomorfismo *suriettivo* di gruppoidi.

(1) Se G è un semigruppo, anche H è un semigruppo.
(2) Se G è commutativo, anche H è commutativo.

86. Il gruppoide $(\mathbb{R}, \mu)$ con $\mu(a,b) := \dfrac{a+b}{2}$ non è associativo.

87. Il prodotto di due matrici $A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ e $B = \begin{pmatrix} a' & b' \\ c' & d' \end{pmatrix}$ è definito come

$$AB := \begin{pmatrix} aa' + bc' & ab' + bd' \\ ca' + dc' & cb' + dd' \end{pmatrix}$$

Definire $A * B := AB - BA$ e trovare tre matrici A, B, C tali che $(A * B) * C \neq A * (B * C)$.

ALGORITMI E STRUTTURE DI DATI

Corso di laurea in matematica | Anno accademico 2005/06 | Numero 11

Creazione di figure mediante sistemi dinamici

La generazione di figure mediante sistemi di Lindenmayer è un caso speciale di un principio generale che può essere formulato nel modo seguente.

Siano $\mathcal{F}$ un sottoinsieme di $\mathcal{P}(\mathbb{R}^m)$ e $T : \mathcal{F} \longrightarrow \mathcal{F}$ un'applicazione. Partendo con una figura $X_0 \in \mathcal{F}$ otteniamo allora una successione $X_0, X_1, X_2, \ldots$ di elementi di $\mathcal{F}$ ponendo $X_{n+1} := T(X_n)$. Se su $\mathcal{F}$ è data una metrica, sotto certe ipotesi si può dimostrare che la successione $\bigcirc_n X_n$ in questa metrica converge sempre a un insieme limite X che non dipende da X_0.

L'esempio più famoso è descritto nel libro di Barnsley e nella tesi di Roberta Nibbi: Sia $\mathcal{F}$ l'insieme dei sottoinsiemi compatti non vuoti di $[0,1]^m$ e siano date contrazioni $\tau_1, \ldots, \tau_n : [0,1]^m \longrightarrow [0,1]^m$, cioè applicazioni tali che $|\tau_k(x) - \tau_k(y)| \leq \alpha|x - y|$ per ogni $x, y \in [0,1]^m$ ed ogni k, per un numero α con $0 \leq \alpha < 1$. Se allora per $A \in \mathcal{F}$ definiamo $T(A) := \tau_1(A) \cup \ldots \cup \tau_n(A)$, otteniamo un'applicazione $T : \mathcal{F} \longrightarrow \mathcal{F}$ con le proprietà a cui abbiamo acennato.

L'insieme di Cantor si ottiene ad esempio da $\tau_1, \tau_2 : [0,1] \longrightarrow [0,1]$ con $\tau_1(x) := x/3$, $\tau_2(x) = (x+2)/3$. Se partiamo con $X_0 = [0,1]$, otteniamo gli insiemi X_n visti a pagina 56.

Per le applicazioni in biologia possono essere interessanti, per un sottoinsieme $\mathcal{F}$ di $\mathcal{P}(\mathbb{R}^m)$, anche applicazioni $T : \mathcal{F} \longrightarrow \mathcal{F}$ che non hanno la proprietà che $\bigcirc_n T^n(X_0)$ converga a un punto limite. Teoricamente ogni evoluzione deterministica (in cui cioè ogni stato determina univocamente lo stato successivo) può essere descritta da un tale *sistema dinamico*. La teoria dei sistemi dinamici è la *dinamica topologica* (quando viene effettuata utilizzando soprattutto i concetti della topologia generale) o *teoria ergodica* (quando si usano soprattutto gli strumenti della teoria della misura) che comprende la dinamica simbolica, cioè lo studio delle applicazioni $A^{\mathbb{N}} \longrightarrow A^{\mathbb{N}}$ oppure $A^{\mathbb{Z}} \longrightarrow A^{\mathbb{Z}}$ per un alfabeto finito A.

R. Nibbi: Topologia dei frattali. Tesi Univ. Ferrara 1991.

M. Barnsley: Fractals everywhere. Academic Press 1988.

P. Walters: An introduction to ergodic theory. Springer 2000.

K. Petersen: Ergodic theory. Cambridge UP 1989.

M. Brin/G. Stuck: Introduction to dynamical systems. Cambridge UP 2002.

A. Katok/B. Hasselblatt: Introduction to the modern theory of dynamical systems. Cambridge UP 1996.

M. Denker/C. Grillenberger/K. Sigmund: Ergodic theory on compact spaces. Springer 1976.

filenameforall

Come annunciato a pagina 17, possiamo utilizzare la funzione `filenameforall` per percorrere una directory o più precisamente un insieme di files il cui nome è di una certa forma.

Per eseguire il `run` su tutti i files della nostra libreria, sotto Linux possiamo ad esempio usare

```
(/home/roberto/Algoritmi/Moduli/*)
  {run} 1000 string filenameforall
```

`*` indica una successione qualsiasi, anche vuota, di caratteri. La sintassi generale del comando è

```
forma f stringa filenameforall
```

dove `f` è una funzione. Per ogni file il cui nome è della forma indicata, questo nome viene copiato nella stringa ed `f` eseguita su questa stringa.

Per ottenere il catalogo di una cartella possiamo usare

```
/File.catalogo {{print (\n) print}
  1000 string filenameforall} def
```

Esempi:

```
(./*) File.catalogo
(/*) File.catalogo
(/home/roberto/Posta/*) File.catalogo
```

In questo numero

Lettura di un file

Per leggere un file si usa `readstring` con la sintassi

```
p stringa readstring
```

dove `p` è un *descrittore di file* (quindi non il nome di un file, come adesso vedremo); il file corrispondente a `p` è un file di testo il cui contenuto viene scritto nella stringa; se la lunghezza del testo è maggiore della lunghezza m della stringa, solo i primi m caratteri del file vengono trascritti.

`readstring` pone sullo stack la parte riempita della stringa e in più un valore booleano, che risulta uguale a `true` se tutta la stringa data come secondo argomento della funzione è stata riempita e `false` altrimenti.

In genere (se non vogliamo appositamente leggere solo una parte iniziale del file) dovremo quindi calcolare la lunghezza del file con la funzione `File.lun` che adesso definiamo.

A questo scopo utilizziamo la funzione `status` di PostScript. Essa, nella sintassi `nome status` , restituisce `false` se il file non esiste, altrimenti una quintupla `x n y z true` , di cui usiamo solo il numero n di bytes contenuti nel file. La nostra funzione `File.lun` restituisce n, se il file esiste, altrimenti `false`:

	nome
status	... esito
false eq	... (esito=false)
{	{
false}	false}
{	{*x n y z* true
T T	*x n*
TPU}	*n*}
ifelse	false oppure *n*

```
/File.lun {status false eq {false} {T T TPU} ifelse} def
```

Per creare il descrittore di un file con accesso in lettura (*read*) dobbiamo usare `nome (r) file` ; il file viene chiuso con `closefile`. Possiamo così definire `File.leggi`:

	nome
U File.lun	*nome e* % *e* ... false o *n*
U	*nome e e*
false eq	*nome e e* = false
{	{ *nome e*
T T ()}	() }
{	{ *nome n*
S	*n nome*
(r) file	*n p*
U	*n p p*
RC3	*p p n*
string	*p p s*
readstring	*p testo v* % *v* ... valore booleano
T	*p testo*
S closefile}	*testo* }
ifelse	*testo*

```
/File.leggi {U File.lun U false eq {T T ()} {S (r) file
  U RC3 string readstring T S closefile} ifelse} def
```

Il risultato della funzione è una stringa che è vuota quando il file non esiste.

Scrittura su un file

In modo simile, creando un descrittore di file con accesso in scrittura (*write*), definiamo una funzione per scrivere un testo su un file:

	testo nome
`(w) file`	*testo p*
`U`	*testo p p*
`RC3`	*p p testo*
`writestring`	*p*
`closefile`	

```
\File.scrivi {(w) file U RC3 writestring closefile} def
```

Usando la modalità di accesso in aggiunta (*add*), definiamo

```
/File.aggiungi {(a) file U RC3 writestring closefile} def
```

Esempi:

```
(uno\ndue\ntre) (prova) File.scrivi
(\nquattro) (prova) File.aggiungi
```

Lavoro interattivo al terminale

PostScript prevede anche alcuni file speciali; i descrittori dei più importanti di questi sono:

`%stdin%`	standard input
`%stdout%`	standard output
`%lineedit%`	I/O interattivo

`%lineedit%` permette di lavorare in modo interattivo al terminale come nel seguente esempio:

```
/s 200 string def
(Come ti chiami? ) print flush
(%lineedit) (r) file s readstring
T % per togliere il valore booleano
(Ciao, ) print print (!\n) print
```

Si noti l'uso di `flush` per forzare l'esecuzione immediata dell'output sul terminale.

La funzione rettangolobs

Abbiamo definito già due funzioni per la creazione di rettangoli, `rettangolo` (a pagina 36) e `rettangolocr` (a pagina 37). Generalizzando leggermente la prima otteniamo una funzione `rettangolobs` il cui primo argomento è il punto in basso a sinistra del rettangolo parallelo agli assi che vogliamo definire e che viene usata con la sintassi

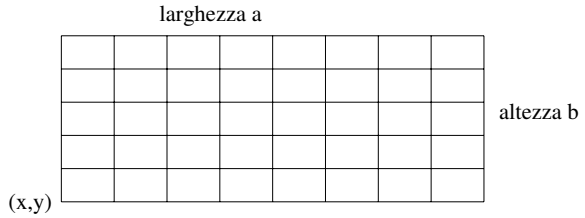
```
[x y] a b rettangolobs
```

	$[x\ y]\ a\ b$
`RC3`	$a\ b\ [x\ y]$
`aload T`	$a\ b\ x\ y$
`moveto`	$a\ b$
`PU 0`	$a\ b\ a\ 0$
`rlineto`	$a\ b$
`0 PU`	$a\ b\ 0\ b$
`rlineto`	$a\ b$
`S`	$b\ a$
`neg 0`	$b\ -a\ 0$
`rlineto`	b
`0 S`	$0\ b$
`neg`	$0\ -b$
`rlineto`	

```
/rettangolobs {RC3 aload T moveto PU 0 rlineto 0 PU
  rlineto S neg 0 rlineto 0 S neg rlineto} def
```

Tabelle rettangolari

Analiticamente rappresentiamo una tabella come un vettore della forma $t = [[x\ y]\ a\ b\ n\ m]$. La funzione `tabella`, che vogliamo definire, è usata con la sintassi `t tabella` e crea il tracciato che corrisponde a una tabella ad n righe ed m colonne in un rettangolo di larghezza a ed altezza b il cui punto in basso a sinistra è uguale ad (x, y):



Per calcolare i vettori vx e vy che contengono i valori x_j e y_i delle ascissi e delle ordinate usiamo la funzione `:` che abbiamo introdotto a pagina 45. L'analisi operazionale richiede solo pazienza, ma non è difficile:

	$t = [[x\ y]\ a\ b\ n\ m]$
`aload T`	$[x\ y]\ a\ b\ n\ m$
`RC5`	$a\ b\ n\ m\ [x\ y]$
`aload T`	$a\ b\ n\ m\ x\ y$
`PU U`	$a\ b\ n\ m\ x\ y\ x\ x$
`7 index`	$a\ b\ n\ m\ x\ y\ x\ x\ a$
`add`	$a\ b\ n\ m\ x\ y\ x, x + a$
`RC5`	$a\ b\ n\ x\ y\ x, x + a, m$
`:`	$a\ b\ n\ x\ y\ vx$
`PU`	$a\ b\ n\ x\ y\ vx\ y$
`5 index`	$a\ b\ n\ x\ y\ vx\ y\ b$
`add`	$a\ b\ n\ x\ y\ vx, y' = y + b$
`S`	$a\ b\ n\ x\ y\ y'\ vx$
`{`	$\{\ a\ b\ n\ x\ y\ y'\ x_j$
`U QU`	$a\ b\ n\ x\ y\ y'\ x_j\ x_j\ y$
`moveto`	$a\ b\ n\ x\ y\ y'\ x_j$
`PU`	$a\ b\ n\ x\ y\ y'\ x_j\ y'$
`lineto}`	$a\ b\ n\ x\ y\ y'\ \}$
`forall`	$a\ b\ n\ x\ y\ y'$
`T U`	$a\ b\ n\ x\ y\ y$
`RC5`	$a\ n\ x\ y\ y\ b$
`add`	$a\ n\ x\ y, y + b$
`RC4`	$a\ x\ y, y + b, n$
`:`	$a\ x\ vy$
`PU`	$a\ x\ vy\ x$
`RC4`	$x\ vy\ x\ a$
`add`	$x\ vy, x' = x + a$
`S`	$x\ x'\ vy$
`{`	$\{\ x\ x'\ y_i$
`TU`	$x\ x'\ y_i\ x$
`PU`	$x\ x'\ y_i\ x\ y_i$
`moveto`	$x\ x'\ y_i$
`PU S`	$x\ x'\ x'\ y_i$
`lineto}`	$x\ y\ x'\ \}$
`forall`	$x\ x'$
`T T`	

```
/tabella {aload T RC5 aload T
  PU U 7 index add RC5 :
  PU 5 index add S {U QU moveto PU lineto} forall
  T U RC5 add RC4 :
  PU RC4 add S
  {TU PU moveto PU S lineto} forall T T} def
```

La figura è stata ottenuta con

```
impostagrafica
[[10 0] 56 22 5 8] tabella ---
3 Font.times
3 -1 moveto ((x,y)) show
68 11 moveto (altezza b) show
23 24 moveto (larghezza a) show
```

Funzioni ausiliarie per tabelle

Definiamo una serie di funzioni che calcolano alcuni parametri associati a una tabella $t = [[x\ y]\ a\ b\ n\ m]$.

`tabxy`	$[x\ y]$
`tabx`	x
`taby`	y
`taba`	a
`tabb`	b
`tabn`	n
`tabm`	m
`tabdx`	larghezza dx di una casella
`tabdy`	altezza dy di una casella
`tabxj`	margine sinistro x_j della j-esima colonna
`tabyi`	margine inferiore y_i della i-esima riga
`tabij`	$[x_j\ y_i]\ dx\ dy$
`tabi`	$[x\ y_i]\ a\ dy$
`tabj`	$[x_j\ y]\ dx\ b$
`tabr`	$[x\ y]\ a\ b$

Con `tabij` otteniamo quindi i parametri della casella che si trova nella i-esima riga e j-esima colonna; `tabi` ci fornisce i parametri della i-esima riga, `tabj` quella della j-esima colonna.

`tabr` restituisce semplicemente i parametri del rettangolo esterno nella forma $[x\ y]\ a\ b]$.

```
/tabxy {0 get} def

/tabx {tabxy 0 get} def

/taby {tabxy 1 get} def

/taba {1 get} def

/tabb {2 get} def

/tabn {3 get} def

/tabm {4 get} def
```

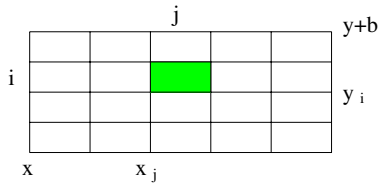
	$t\ \%\ dx = a/m$
`U`	$t\ t$
`taba`	$t\ a$
`S`	$a\ t$
`tabm`	$a\ m$
`div`	dx

$dy = b/n$ si ottiene in modo analogo; definiamo perciò:

```
/tabdx {U taba S tabm div} def

/tabdy {U tabb S tabn div} def
```

Come si vede dalla figura, si ha

$$x_j = x + (j-1)\cdot dx$$
$$y_i = y + b - i\cdot dy$$



	$t\ j$
`1 sub`	$t,\ j' = j-1$
`PU`	$t\ j'\ t$
`tabdx`	$t\ j'\ dx$
`mul`	$t\ j'dx$
`S`	$j'dx\ t$
`tabx`	$j'dx\ x$
`add`	x_j

	$t\ i$
`P`	$t\ i\ t$
`tabdy`	$t\ i\ dy$
`mul`	$t\ idy$
`neg`	$t,\ -idy$
`PU`	$t,\ -idy,\ t$
`tabb`	$t,\ -idy,\ b$
`add`	$t\ b' = b - idy$
`S`	$b'\ t$
`taby`	$b'\ y$
`add`	y_i

```
/tabxj {1 sub PU tabdx mul S tabx add} def

/tabyi {PU tabdy mul neg PU tabb add S taby add} def
```

Adesso possiamo definire `tabij`:

	$t\ i\ j$
`TU`	$t\ i\ j\ t$
`S`	$t\ i\ t\ j$
`tabxj`	$t\ i\ x_j$
`TU`	$t\ i\ x_j\ t$
`RC3`	$t\ x_j\ t\ i$
`tabyi`	$t\ x_j\ y_i$
`2 array astore`	$t\ [x_j\ y_i]$
`PU`	$t\ [x_j\ y_i]\ t$
`tabdx`	$t\ [x_j\ y_i]\ dx$
`RC3`	$[x_j\ y_i]\ dx\ t$
`tabdy`	$[x_j\ y_i]\ dx\ dy$

```
/tabij {TU S tabxj TU RC3 tabyi 2 array astore
  PU tabdx RC3 tabdy} def
```

In modo simile otteniamo `tabi` e `tabj`:

	$t\ i$
`PU tabx`	$t\ i\ x$
`TU`	$t\ i\ x\ t$
`RC3 tabyi`	$t\ x\ y_i$
`2 array astore`	$t\ [x\ y_i]$
`PU taba`	$t\ [x\ y_i]\ a$
`RC3 tabdy`	$[x\ y_i]\ a\ dy$

```
/tabi {PU tabx TU RC3 tabyi 2 array astore
  PU taba RC3 tabdy} def
```

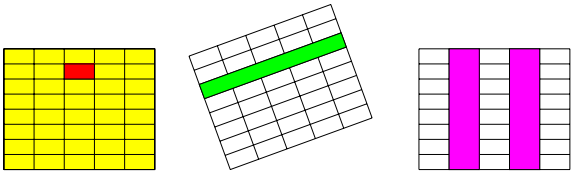
	$t\ j$
`PU`	$t\ j\ t$
`S tabxj`	$t\ x_j$
`PU taby`	$t\ x_j\ y$
`2 array astore`	$t\ x_j\ y]$
`PU tabdx`	$t\ x_j\ y]\ dx$
`RC3`	$x_j\ y]\ dx\ t$
`tabb`	$x_j\ y]\ dx\ b$

```
/tabj {PU S tabxj PU taby 2 array astore
  PU tabdx RC3 tabb} def
```

Infine definiamo

```
/tabr {aload T T T} def
```

Per trasformare un'espressione `[x y] a b` in un rettangolo, dobbiamo usare `rettangolobs`:



```
impostagrafica
/t [[0 0] 20 16 8 5] def

t tabr rettangolobs rgiallo t tabella ---
t 2 3 tabij rettangolobs rrosso ---

30 0 translate
$ 20 rotate t tabella ---
t 3 tabi rettangolobs rverde --- *
25 0 translate
t tabella ---
t 2 tabj rettangolobs rmagenta ---
t 4 tabj rettangolobs rmagenta ---
```