Generalità

Che cos'è la matematica 1
L'alfabeto greco 1
Il conto in banca aumenta 2
Problemi semplici difficili 2
Il re dei matematici 6

Algebra lineare

Equazioni lineari in una incognita 3
Prime nozioni astratte 3
Un teorema fondamentale 3
2 equazioni lineari in 2 incognite 4
Esempi 4
La regola di Cramer 4
Algoritmo di eliminazione di Gauß 5
Sistemi con più di una soluzione 6
L'insieme delle soluzioni 6
Esercizi 6
Un po' di algebra esterna 27
Determinanti 28
Multilinearità del determinante 28
Dimostrazione del teorema 27/2 29
Due casi speciali 29

Algebra

Il massimo comune divisore 23
Divisione con resto 24
Sottogruppi di $\mathbb{Z}$ 24
Primo incontro con gli ideali 24
Estensioni di campi 47

Algebra computazionale

L'algoritmo euclideo 23
La moltiplicazione russa 25
Trovare la rappresentazione binaria 25
La potenza russa 25
Lo schema di Horner 26

Trigonometria

Trigonometria oggi 7
Un problema di geodesia 7
Il triangolo 8
Il triangolo rettangolo 8
Le funzioni trigonometriche 8
La dimostrazione indiana 9
Il triangolo isolatero 9
Angoli sul cerchio 9
Il teorema del coseno 10
Il grafico della funzione seno 10
La periodicità di $\sin$ e $\cos$ 10
Altre proprietà di seno e coseno 10

Geometria analitica

Grafica al calcolatore e geometria 7
Distanze in $\mathbb{R}^n$ 11
Il prodotto scalare 11
Ortogonalità 11
Rette nel piano 11
Intersezione di rette nel piano 27
Proiezione di un punto su una retta 29
Punto e retta nel piano 29
La retta in uno spazio vettoriale 30
Due rette in $\mathbb{R}^3$ 30
Il prodotto vettoriale 30
Identità di Graßmann e di Jacobi 31
Area di un parallelogramma 31
Significato geometrico di $v \times w$ 31
Piani nello spazio 32
Proiezione di un punto su un piano 32
Il piano passante per tre punti 33
Il volume 33
Orientamento 33
Coordinate polari nel piano 37
Coordinate cilindriche 37
Coordinate polari nello spazio 37
Rotazioni nel piano 37

I numeri complessi

I numeri complessi 41
La formula di Euler 41
Il campo dei numeri complessi 42
La disuguaglianza di Cauchy-Schwarz 42
La disuguaglianza triangolare 42
La formula di de Moivre 42
Valore assoluto e quoziente di numeri complessi 47
Le equazioni di Cauchy-Riemann 47

Programmazione

I linguaggi di programmazione 12
Tre problemi semplici 12
Procedure e funzioni 12
I numeri di Fibonacci 13
Il sistema di primo ordine 13
Numeri esadecimali 13
Diagrammi di flusso 14
Il linguaggio macchina 14
I linguaggi assembler 14
Basic 15
Fortran 15
Pascal 15
Java 17
Python 17
Programmare in Forth 34
Lisp 46
Il λ-calcolo 46
Prolog 46
Gli altri linguaggi 46

Programmazione in Perl

Perl 17
Programmare in Perl 18
Variabili nel Perl 18
Input dalla tastiera 18
Liste 19
La funzione *grep* del Perl 19
Alcuni operatori per liste 19
Contesto scalare e contesto listale 19
Files e operatore <> 20
Funzioni del Perl 20
Moduli 20
Il modulo files 20
Vero e falso 21
Operatori logici 21
Operatori di confronto 21
Istruzioni di controllo 21
map 21
Nascondere le variabili con my 23
Zeri di una funzione continua 26

Programmazione in C

C 16
C++ 16
Programmare in C 38
Il programma minimo 38
I header generali 38
Il preprocessore 39
printf 39
I commenti 39
Calcoliamo il fattoriale 39
Comandi di compilazione 40
Il makefile 40
Il comando make 40
Come funziona make 40
for 41
Operatori logici 41
Vettori e puntatori 43
Stringhe e caratteri 43
Aritmetica dei puntatori 43
Operatori abbreviati 43
Confronto di stringhe 44
if ... else 44
Puntatori generici 44
Conversioni di tipo 44
Parametri di main 45
Input da tastiera 45
Passaggio di parametri 45
Variabili di classe static 46
Le strutture del C 47

Programmazione in PostScript

Forth e PostScript 34
Lo stack 34
Usare ghostscript 35
Il comando run 35
Usare def 35
Diagrammi di flusso per lo stack 35
Lo stack dei dizionari 35
Argomenti di una macro 36
show e selectfont 36
if e ifelse 36
Cerchi con PostScript 36

Unix e Emacs

Comandi Emacs 22
Scrivere programmi con Emacs 22
Semplici comandi Unix 22

Varia

Tirocini all'ARDSU 25
Linux Day al dipartimento di Matematica 42

ALGORITMI E STRUTTURE DI DATI

Corso di laurea in matematica | Anno accademico 2001/2002 | Numero 1 ◇ 24 Settembre 2001

Che cos'è la matematica?

Dividiamo questa domanda in due sottodomande, cercando di indicare prima i costituenti elementari della matematica, poi come la matematica deve essere usata.

I componenti elementari del ragionamento matematico sono le coppie ipotesi-tesi; in questo senso la matematica non conosce affermazioni assolute, ma soltanto proposizioni che si compongono ogni volta di un preciso elenco delle ipotesi che vengono fatte, e poi di una altrettanto precisa specificazione dell'enunciato che secondo quella proposizione ne consegue. A questo punto non è detto che la proposizione sia valida, bisogna ancora dimostrarla, e ciò significa, nella matematica, dimostrare che la tesi segue dalle ipotesi unite agli assiomi e ai risultati già ottenuti e alle regole logiche che dobbiamo applicare. Gli assiomi sono enunciati che vengono messi all'inizio di una teoria, senza dimostrazione; ogni altro enunciato deve essere invece dimostrato.

È importante che bisogna sempre dimostrare una proposizione - che è sempre nella forma ipotesi-tesi! - nella sua interezza, cioè che si tratta di dimostrare la validità dell'implicazione e non la validità della tesi. L'enunciato `A implica B` *può essere vero, anche se* `B` *non e vero. Ad esempio in logica si impara che, se l'ipotesi* `A` *è falsa, allora la proposizione* `A implica B` *è sempre vera. Quindi l'affermazione* `se 3 è uguale a 3.1, allora io mi chiamo Piero` *è sempre vera, indipendentemente da come mi chiamo io. Nella pratica matematica ciò significa che da una premessa errata si può, con un po' di pazienza, dedurre qualunque cosa.*

La validità si riferisce quindi sempre a tutta la proposizione `A implica B`*, cioè alla coppia ipotesi-tesi.*

Mentre il matematico puro cerca soprattutto di arricchire l'edificio delle teorie matematiche con nuovi concetti o con dimostrazioni, talvolta assai difficili, di teoremi, il matematico applicato deve anche saper usare la matematica. Nelle scienze naturali e sociali, le quali pongono problemi molto complessi, uno dei compiti più importanti è spesso la separazione degli elementi essenziali di un fenomeno dagli aspetti marginali. In queste scienze le informazioni disponibili sono quasi sempre incomplete, cosicché possiamo ogni volta descrivere soltanto una piccola parte della realtà. Anche quando disponiamo di conoscenze dettagliate, queste si presentano in grande quantità, sono complesse e multiformi e richiedono concetti ordinatori per poterle interpretare. Ciò significa che bisogna estrarre e semplificare.

Un modello matematico di un fenomeno ha soprattutto lo scopo di permettere di comprendere meglio quel fenomeno, quindi di metterne in evidenza cause e effetti e comportamenti quantitativi, di individuarne i tratti essenziali e i meccanismi fondamentali. In un certo senso la matematica consiste di tautologie, e nel modello matematico si tenta di evidenziare le tautologie contenute nel fenomeno studiato. La teoria cerca di comprendere i processi e legami funzionali di un campo del sapere.

La mente umana pensa in modelli. Anche quando non facciamo matematica della natura, cerchiamo di comprendere la natura mediante immagini semplificate. La teoria inizia già nell'istante in cui cominciamo a porci la domanda quali siano gli aspetti essenziali di un oggetto o di un fenomeno. La matematica non è dunque altro che un modo sistematico e controllato di eseguire questi processi di astrazione e semplificazione costantemente attivi nel nostro intelletto.

Il modello matematico, una volta concepito, se sviluppato correttamente, si mostra poi di una esattezza naturale che spesso impressiona l'utente finale che è tentato di adottare gli enunciati matematici come se essi corrispondessero precisamente ai fenomeni modellati. Ma ciò non è affatto vero: La precisione del modello matematico è soltanto una precisione interna, tautologica, e la semplificazione, quindi verità approssimata e parziale, che sta all'inizio del modello, si conserva, e più avanza lo sviluppo matematico, maggiore è il pericolo che iterando più volte l'errore, questo sia cresciuto in misura tale da richiedere un'interpretazione estremamente prudente dei risultati matematici. Proprie le teorie più avanzate, più belle quindi per il matematico puro, sono spesso quelle più lontane dalla realtà. Questo automatismo della matematica può essere però anche fonte di nuovi punti di vista e svelare connessioni nascoste.

Un modello matematico è perciò solo un ausilio per la riflessione, per controllare il contenuto e la consistenza logica di un pensiero o di una ricerca. In altre parole, modelli sono strumenti intellettuali e non si possono da essi aspettare descrizioni perfette della realtà. Essi non forniscono risposte complete, ma indicano piuttosto quali siano le domande che bisogna porre.

L'astrattezza intrinseca della matematica comporta da un lato che essa rimanga sempre diversa dalla realtà, offre però dall'altro lato la possibilità di generalizzare i risultati ottenuti nelle ricerche in un particolare campo applicativo o anche uno strumento della matematica pura a problemi apparentemente completamente diversi, se questi hanno proprietà formali in comune con il primo campo.

Questa settimana

1 Che cos'è la matematica?
L'alfabeto greco

2 Il conto in banca aumenta
Problemi semplici difficili

3 Equazioni lineari in una incognita
Prime nozioni astratte
Un teorema fondamentale

4 2 equazioni lineari in 2 incognite
Esempi
La regola di Cramer

5 Algoritmo di eliminazione di Gauß

6 Sistemi con più di una soluzione
L'insieme delle soluzioni
Esercizi
Il re dei matematici

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. Per ogni lettera sono indicati il nome italiano, la forma minuscola e quella maiuscola. La sigma minuscola ha due forme: alla fine della parola si scrive ς, altrimenti σ. In matematica si usa solo la σ.

Mikros (μικρός) significa piccolo, megas (μέγας) 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 $a_0a_1a_2a_3$, mentre A^i_α è un oggetto con due indici, per uno dei quali abbiamo usato una lettera greca.

Il conto in banca aumenta

I matematici usano il simbolo $\Rightarrow$ come abbreviazione di *implica* (e $\Leftrightarrow$ per la doppia implicazione), quindi $A \Rightarrow B$ è un'abbreviazione per A implica B.

Usiamo questa abbreviazione nel seguente ragionamento, che mostra come piccoli errori possono generare errori grandi, se vengono eseguite varie operazioni a partire da essi, anche se queste operazioni sono formalmente corrette. Con un po' di fantasia quindi il seguente algoritmo può essere utilizzato per aumentare il nostro conto in banca (e viceversa si vede che anche nelle operazioni bancarie conoscenze in analisi numerica possono essere alquanto utili). Naturalmente, volendo, anche partendo da $3 = 3.0000000001$ si arriva a $1 = 1000$, ed errori così piccoli sono comuni nell'aritmetica dei calcolatori.

Assumiamo che $3 = 3.1$.

Allora anche $30 = 31$, quindi $1 = 2$ (come si vede sottraendo 29), cosicché $1000 = 2000$.

Però se $1 = 2$, possiamo anche addizionare 2 a sinistra e 1 a destra, ottenendo $1002 = 2001$, e se adesso sottraiamo 1001 da entrambi i lati, otteniamo $1 = 1000$.

Problemi semplici difficili

L'uso di modelli nelle scienze naturali oppure in ingegneria e economia comporta due difficoltà: In primo luogo bisogna concepire modelli adatti e interpretare i risultati matematici ottenuti nel modo adeguato. Ma le difficoltà possono essere anche di carattere matematico. Diamo alcuni esempi di problemi puramente matematici di semplice formulazione, in cui non si pongono sottili questioni di interpretazione, la cui soluzione è però talmente difficile che i matematici non sanno nemmeno da che parte cominciare per risolverli. Di questi problemi ne esistono migliaia. Alcuni sono poco più che giochetti, ma problemi altrettanto difficili appaiono in molti modelli matematici di problemi importanti della fisica e della biologia.

Il problema dei tre corpi, cioè il comportamento di tre corpi sufficientemente vicini e di massa approssimativamente uguali sotto l'influsso delle forze gravitazionali che essi esercitano l'uno sull'altro, è matematicamente estremamente difficile e ha più di una soluzione - pochi anni fa sono state trovate nuove soluzioni. Sole, terra e luna non sono un esempio, perché il sole ha massa molto più grande degli altri due, cosicché il sistema può essere descritto dal moto del baricentro di terra e luna attorno al sole, e poi della luna attorno alla terra, si decompone quindi in due problemi di due corpi, e questi obbediscono alle leggi di Kepler.

Il problema dei primi gemelli. Un numero naturale (0,1,2,...) diverso da 1 si dice primo, se non è divisibile (senza resto) da nessun numero naturale tranne 1 e se stesso. Ad esempio i primi ≤ 40 sono $2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37$. Si può dimostrare che esiste un numero infinito di primi (*). Due numeri primi che si distinguono di 2, come 3 e 5, 11 e 13, 17 e 19, 29 e 31, 41 e 43, si chiamano primi gemelli. Nessun matematico è finora riuscito a dimostrare che esistono infiniti primi gemelli, anche se, come si vede dagli esempi, sembra che ce ne siano parecchi e si è tutti convinti che devono essere infiniti.

(*) Ecco la dimostrazione, risalente addirittura a Euclide e quindi vecchia circa 2300 anni, che esiste un numero infinito di numeri primi. Dimostriamo che, se si assume il contrario, si arriva a una contraddizione.

Facciamo prima una considerazione ausiliaria - dimostriamo che ogni numero naturale $n > 1$ è diviso da un numero primo. Infatti i divisori di n sono tutti compresi tra 1 e n, e certamente tra quelli che sono maggiori di 1 ne esiste uno più piccolo, che chiamiamo p. Ogni divisore di p però è anche divisore di n, quindi p non può avere un divisore d con $1 < d < p$, altrimenti d sarebbe un divisore di n più piccolo di p e maggiore di 1. Ciò significa però che p è primo.

Assumiamo adesso che esista solo un numero finito di numeri primi. Allora possiamo formare il prodotto di tutti questi che chiamiamo P. $P + 1$ è certamente maggiore di 1, quindi, per quello che abbiamo appena dimostrato, esiste anche un numero primo p che divide $P + 1$, ad esempio $P + 1 = px$. D'altra parte però p, essendo primo, deve essere uno dei fattori di P (che è appunto il prodotto di tutti i primi), e quindi p divide anche P, ad esempio $P = py$. Ciò implica $1 = P + 1 - P = px - py = p(x - y)$, quindi p divide 1, ma allora $p \leq 1$, una contraddizione.

Il problema dei numeri perfetti. Un numero naturale si dice perfetto, se è uguale alla somma dei suoi divisori diversi da se stesso. Ad esempio $6 = 1 + 2 + 3$ e $28 = 1 + 2 + 4 + 7 + 14$ sono numeri perfetti. Esistono infiniti numeri perfetti? Esiste almeno un numero perfetto dispari? Non si sa.

Il problema del 3x+1. Questo è sicuramente il problema più inutile della matematica. Sembra incredibile che, da quando è stato posto più di 60 anni fa, nessuno sia riuscito a risolverlo.

Partiamo con un numero naturale maggiore di 1 qualsiasi. Se è pari, lo dividiamo per 2, altrimenti lo moltiplichiamo per 3 e aggiungiamo 1. Con il numero così ottenuto ripetiamo l'operazione, e ci fermiamo solo quando arriviamo a 1. Ad esempio partendo con 7 otteniamo $22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1$.

In genere si arriva a 1 molto presto. Ma con il seguente programma in Perl si scopre che, se si parte con 27, il ciclo è piuttosto lungo.

```
sub T {my $a=shift; if ($a%2==0) {$a/2}
  else {3*$a+1}}
sub L {my $a=shift; my @lista=();
  while ($a> 1)
  {push(@lista,$a); $a=T($a)}
  @lista}
for (2..30) {@lista=L($_); if (@lista> 10)
  {print "@lista\n"}}
```

Sembra che alla fine l'algoritmo si fermi sempre, cioè che prima o poi si arrivi sempre ad 1. Ma nessuno riesce a dimostrarlo.

Questi esempi, di cui l'ultimo è particolarmente semplice, mostrano che anche quando si riesce a trasformare un problema scientifico, ingegneristico o economico in un problema matematico, non è affatto detto che la matematica riesca a fornire una soluzione. Spesso si renderanno necessarie ulteriori semplificazioni perché il modello più realistico può essere formulato ma non risolto matematicamente.

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:

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

Per la definizione dei numeri reali rimandiamo al corso di Analisi I. È anche 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 numero intero è un numero naturale oppure il negativo di un numero naturale).

Prime nozioni astratte

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

$$\begin{aligned} a_1x + b_1y &= c_1 \\ a_2x + b_2y &= c_2 \end{aligned}$$

significa trovare tutte le coppie (x, y) di numeri reali che soddisfano entrambe le equazioni. Denotiamo le due equazioni con E_1 ed E_2 e definiamo, per numeri reali x, y,

$$\begin{aligned} S_1(x,y) &:= a_1x + b_1y,\ D_1(x,y) = c_1 \\ S_2(x,y) &:= a_2x + b_2y,\ D_2(x,y) = c_2 \end{aligned}$$

dove $:=$ significa *uguale per definizione*. Allora possiamo scrivere il sistema nella forma

$$\begin{aligned} S_1(x,y) &= D_1(x,y) \\ S_2(x,y) &= D_2(x,y) \end{aligned}$$

(S ... sinistra, D ... destra). E adesso introduciamo alcune di quelle nozioni astratte, che sono caratteristiche per la matematica e il modo di pensare del matematico e che permettono formulazioni di enunciati e dimostrazioni precise: La somma $E_1 + E_2$ delle due equazioni è l'equazione

$$S_1(x,y) + S_2(x,y) = D_1(x,y) + D_2(x,y),$$

la loro differenza $E_1 - E_2$ è l'equazione

$$S_1(x,y) - S_2(x,y) = D_1(x,y) - D_2(x,y),$$

e, per un numero reale α, l'equazione αE_1 è data da

$$\alpha S_1(x,y) = \alpha D_1(x,y).$$

A questo punto risulta però, per numeri reali α_1, α_2, definita anche l'equazione $\alpha_1 E_1 + \alpha_2 E_2$:

$$\alpha_1 S_1(x,y) + \alpha_2 S_2(x,y) = \alpha_1 D_1(x,y) + \alpha_2 D_2(x,y).$$

Con $\mathrm{sol}(E_1, E_2)$ denotiamo l'insieme di tutte le soluzioni del sistema.

Un teorema fondamentale

Siamo sempre nelle stesse ipotesi, con α_1 e α_2 due numeri reali, di cui però adesso - attenzione! - α_1 deve essere $\neq 0$. Affermiamo che allora

$$\mathrm{sol}(E_1, E_2) = \mathrm{sol}(\alpha_1 E_1 + \alpha_2 E_2, E_2),$$

cioè che il sistema

$$\begin{aligned} \alpha_1 S_1(x,y) + \alpha_2 S_2(x,y) &= \alpha_1 D_1(x,y) + \alpha_2 D_2(x,y) \\ S_2(x,y) &= D_2(x,y) \end{aligned}$$

oppure, scritto per esteso,

$$\begin{aligned} (\alpha_1 a_1 + \alpha_2 a_2)x + (\alpha_1 b_1 + \alpha_2 b_2)y &= \alpha_1 c_1 + \alpha_2 c_2 \\ a_2x + b_2y &= c_2 \end{aligned}$$

in cui abbiamo sostituito la prima equazione con $\alpha_1 E_1 + \alpha_2 E_2$, possiede le stesse soluzioni di quello originale.

Dimostrazione: (1) (x, y) sia una soluzione del sistema originale, cioè un elemento di $\mathrm{sol}(E_1, E_2)$. Ma allora $S_1(x,y) = D_1(x,y)$ e $S_2(x,y) = D_2(x,y)$, e quindi anche $\alpha_1 S_1(x,y) + \alpha_2 S_2(x,y) = \alpha_1 D_1(x,y) + \alpha_2 D_2(x,y)$. Ciò significa che (x, y) soddisfa la prima equazione del nuovo sistema. La seconda equazione è la stessa in entrambi i sistemi.

Si vede che in questo primo punto non abbiamo usato l'ipotesi che α_1 sia $\neq 0$.

(2) (x, y) sia un elemento di $\mathrm{sol}(\alpha_1 E_1 + \alpha_2 E_2, E_2)$, cioè una soluzione di

$$\begin{aligned} \alpha_1 S_1(x,y) + \alpha_2 S_2(x,y) &= \alpha_1 D_1(x,y) + \alpha_2 D_2(x,y) \\ S_2(x,y) &= D_2(x,y) \end{aligned}$$

Allora nella prima equazione, a destra, invece di $D_2(x, y)$ possiamo anche scrivere $S_2(x, y)$, e quindi quella equazione diventa

$$\alpha_1 S_1(x,y) + \alpha_2 S_2(x,y) = \alpha_1 D_1(x,y) + \alpha_2 S_2(x,y),$$

e ciò implica

$$\alpha_1 S_1(x,y) = \alpha_1 D_1(x,y).$$

Ma, per ipotesi, $\alpha_1 \neq 0$, per cui vediamo che anche

$$S_1(x,y) = D_1(x,y).$$

Il teorema fondamentale in forma generale

Siano date m equazioni lineari in n incognite:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= c_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= c_2 \\ &\cdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n &= c_n \end{aligned}$$

I coefficienti a_{ij} sono numeri reali, le incognite sono $x_1, \ldots, x_n$. Risolvere questo sistema significa trovare tutte le n-ple $(x_1, \ldots, x_n)$ che soddisfano tutte le m equazioni. Denotiamo con $(E_1, \ldots, E_m)$ il sistema stesso e con $\mathrm{sol}(E_1, \ldots, E_m)$ l'insieme di tutte le soluzioni.

Anche qui denotiamo la i-esima equazione con E_i e con $S_i(x_1, \ldots, x_n)$ la parte sinistra, con $D_i(x_1, \ldots, x_n)$ la parte destra della i-esima riga. Di nuovo possiamo scrivere il sistema nella forma (che servirà nella dimostrazione, non nell'enunciato del teorema):

$$\begin{aligned} S_1(x_1, \ldots, x_n) &= D_1(x_1, \ldots, x_n) \\ &\cdots \\ S_m(x_1, \ldots, x_n) &= D_m(x_1, \ldots, x_n) \end{aligned}$$

Teorema: Siano $\alpha_1, \ldots, \alpha_m$ numeri reali e $\alpha_i \neq 0$.

Sia $(F_1, \ldots, F_m)$ il sistema che si ottiene sostituendo nel sistema originale l'i-esima equazione con $\alpha_1 E_1 + \cdots + \alpha_m E_m$. Allora

$$\mathrm{sol}(E_1, \ldots, E_m) = \mathrm{sol}(F_1, \ldots, F_m)$$

Dimostrazione: Esercizio. Prima per $i = 1$, poi per i generale.

Due equazioni lineari in due incognite

Usiamo il teorema fondamentale per risolvere prima il sistema di due equazioni in due incognite. Lasciamo come esercizio il caso molto facile che $a_1 = 0$.

Assumiamo quindi che $a_1 \neq 0$ e formiamo l'equazione $E_3 := a_1E_2 - a_2E_1$. Per il teorema fondamentale $\mathrm{sol}(E_1, E_2) = \mathrm{sol}(E_1, E_3)$. Scritta per esteso l'equazione E_3 diventa

$$\underline{a_1a_2x} + a_1b_2y - \underline{a_2a_1x} - a_2b_1y = a_1c_2 - a_2c_1,$$

cioè

$$(a_1b_2 - a_2b_1)y = a_1c_2 - a_2c_1,$$

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

$$\begin{aligned} a_1x + b_1y &= c_1 \\ (a_1b_2 - a_2b_1)y &= a_1c_2 - a_2c_1 \end{aligned}$$

Il numero $a_1b_2 - a_2b_1$ 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

$$y = \frac{a_1c_2 - a_2c_1}{a_1b_2 - a_2b_1}.$$

Se per numeri reali a, b, c, d poniamo

$$\begin{vmatrix} a & b \\ c & d \end{vmatrix} := ad - bc,$$

allora 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

$$\begin{aligned} x &= \frac{c_1 - b_1y}{a_1} = \\ &= \frac{c_1 - b_1\dfrac{a_1c_2 - a_2c_1}{a_1b_2 - a_2b_1}}{a_1} = \\ &= \frac{a_1b_2c_1 - \underline{a_2b_1c_1} - b_1a_1c_2 + \underline{b_1a_2c_1}}{a_1(a_1b_2 - a_2b_1)} = \\ &= \frac{a_1b_2c_1 - b_1a_1c_2}{a_1(a_1b_2 - a_2b_1)} = \frac{b_2c_1 - b_1c_2}{a_1b_2 - a_2b_1} = \frac{\begin{vmatrix} c_1 & b_1 \\ c_2 & b_2 \end{vmatrix}}{\begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix}} \end{aligned}$$

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

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

Risultato: $x = \dfrac{69}{30}$, $y = \dfrac{8}{30}$.

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):

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

Nel corso di Geometria I anche in questo caso più generale verrà definito 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 che questo determinante è diverso da 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.

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 fondamentale che abbiamo visto a pagina 4.

Esempio 1: Consideriamo il sistema

$$\begin{aligned} 3x + 2y - z &= 10 \ldots E_1 \\ 4x - 9y + 2z &= 6 \ \ \ldots E_2 \\ x + y - 14z &= 2 \ \ \ldots E_3 \end{aligned}$$

Con la notazione che abbiamo già più volte utilizzata poniamo $E_4 := 4E_1 - 3E_2$. Per il teorema fondamentale allora $\mathrm{sol}(E_1, E_2, E_3) = \mathrm{sol}(E_4, E_2, E_3)$, perché il coefficiente con cui E_1 appare in E_4 è diverso da 0. Esplicitamente E_4 è uguale 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

$$\begin{aligned} 4x - 9y + 2z &= 6 \ \ \ldots E_2 \\ x + y - 14z &= 2 \ \ \ldots E_3 \\ 35y - 10z &= 22 \ldots E_4 \end{aligned}$$

Nell'ultima equazione la variabile x in E_4 è sparita, è stata *eliminata*. Ripetiamo questa operazione sostituendo l'equazione E_2 con $E_5 := E_2 - 4E_3$ (ciò è possibile perché in E_5 la E_2 appare con un coefficiente $\neq 0$). Esplicitamente E_5 è uguale a

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

cioè a

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

Perciò il sistema originale ha le stesse soluzioni come il sistema

$$\begin{aligned} x + y - 14z &= 2 \ \ \ldots E_3 \\ 35y - 10z &= 22 \ \ \ldots E_4 \\ -13y + 58z &= -2 \ldots E_5 \end{aligned}$$

Adesso formiamo $E_6 := 13E_4 + 35E_5$ che può sostituire sia la E_4 che la E_5. Sostituiamo la E_5. La E_6 è data da

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

cioè da

$$1900z = 216.$$

Otteniamo così il sistema

$$\begin{aligned} x + y - 14z &= 2 \ \ \ldots E_3 \\ 35y - 10z &= 22 \ \ \ldots E_4 \\ 1900z &= 216 \ldots E_6 \end{aligned}$$

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 la E_4, otteniamo

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

e infine dalla E_3

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

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~~	E_1
~~4~~	~~−9~~	~~2~~	~~6~~	E_2
1	1	−14	2	E_3
0	35	−10	22	$E_4 = 4E_1^* + 3E_2$
~~0~~	~~−13~~	~~58~~	~~−2~~	~~$E_5 = E_2^* - 4E_3$~~
0	0	1900	216	$E_6 = 13E_4 + 35E_5^*$

L'asterisco indica ogni volta l'equazione cancellata in quel punto.

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 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	E_1
1	4	−1	2	3	E_2
~~3~~	~~−1~~	~~2~~	~~1~~	~~7~~	E_3
~~0~~	~~−13~~	~~5~~	~~−5~~	~~−5~~	~~$E_4 = E_1^* - 2E_2$~~
0	−13	5	−5	−2	$E_5 = E_3^* - 3E_2$
0	0	0	0	−3	$E_6 = E_4^* - E_5$

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

Esercizio: Risolvere il sistema

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

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~~	~~E_1~~
1	2	−10	4	E_2
~~6~~	~~3~~	~~−17~~	~~13~~	~~E_3~~
~~0~~	~~−9~~	~~43~~	~~−11~~	~~$E_4 = E_1^* - 4E_2$~~
0	9	−43	11	$E_5 = 6E_2 - E_3^*$
0	0	0	0	$E_6 = E_4^* + E_5$

Stavolta non abbiamo una contraddizione, piuttosto l'ultima equazione $0 = 0$ è 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 con la rappresentazione parametrica

$$x(t) = \frac{14}{9} + \frac{4}{9}t$$

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

$$z(t) = t$$

Per ogni numero reale t si ottiene un punto $(x(t), y(t), z(t))$ che è una soluzione del nostro sistema, e viceversa ogni soluzione è di questa forma.

Esercizio: Risolvere il sistema lineare

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

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 e la definizione precisa del concetto di spazio affine verranno date nel corso di Geometria I.

Esercizi

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

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

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

$$\begin{aligned} 4x_1 + 8x_2 + 17x_3 + 100x_4 + 2x_5 &= 0 \\ 3x_1 + 5x_2 + 32x_3 + 33x_4 + 34x_5 &= 1 \\ 2x_2 + x_3 + 100x_4 + 2x_5 &= 4 \\ 2x_1 + 3x_2 + 15x_3 + 6x_4 + 5x_5 &= -1 \\ 4x_1 + 8x_2 + x_3 + 7x_4 + 2x_5 &= 2 \end{aligned}$$

$$\begin{aligned} 121x_1 + 58x_2 + 85x_3 + 45x_4 &= 1 \\ 2x_1 + 175x_2 + 619x_3 + 218x_4 &= 0 \\ 415x_1 + 26x_2 + 510x_3 + 77x_4 &= 2 \\ 8x_1 + 15x_2 + 315x_3 + 418x_4 &= -1 \end{aligned}$$

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

Il re dei matematici

Carl Friedrich Gauß (1777-1855) è considerato il re dei matematici. La lettera β 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).

È stato forse il primo a concepire le geometrie non euclidee, ha dato una semplice spiegazione 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.

ALGORITMI E STRUTTURE DI DATI

Corso di laurea in matematica | Anno accademico 2001/2002 | Numero 2 ◇ 1 Ottobre 2001

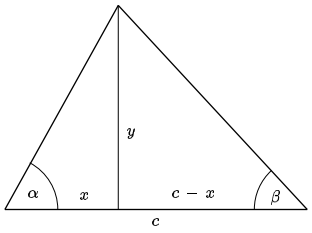
Trigonometria oggi

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, non nella ricerca ovviamente, ma nelle applicazioni.

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, astronomia, cristallografia, classificazione dei movimenti nello spazio, grafica al calcolatore.

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. 9). 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

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

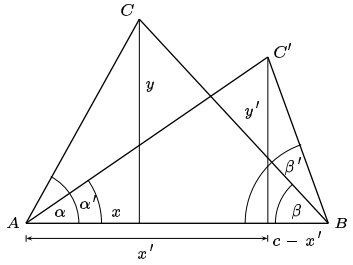
e quindi > 0. Possiamo perciò applicare la regola di Cramer e otteniamo

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

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, 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

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

Questa settimana

7 Trigonometria oggi
Un problema di geodesia
Grafica al calcolatore e geometria

8 Il triangolo
Il triangolo rettangolo
Le funzioni trigonometriche

9 La dimostrazione indiana
Il triangolo isolatero
Angoli sul cerchio

10 Il teorema del coseno
Il grafico della funzione seno
La periodicità di sin e cos
Altre proprietà di seno e coseno

11 Distanze in $\mathbb{R}^n$
Il prodotto scalare
Ortogonalità
Rette nel piano

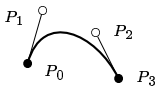
Grafica al calcolatore e geometria

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 affine 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.

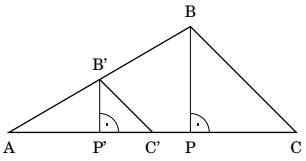
I concetti dell'analisi e della geometria differenziale portano all'introduzione e allo studio delle curve e superficie di Bézier,



largamente utilizzate nei programmi di disegno al calcolatore (CAD, computer aided design).

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

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

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

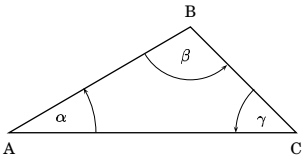
Se il valore comune di queste tre frazioni viene denotato con λ, abbiamo quindi

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

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 $0 < \alpha < 180°$.

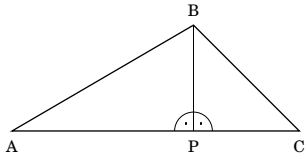
Con β e γ indichiamo gli altri due angoli come nella figura; spesso serve solo la grandezza assoluta degli angoli, allora si lasciano via le punte di freccia.

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

$$\sphericalangle(AC, AB) = \sphericalangle(AC', AB')$$
$$\sphericalangle(BA, BC) = \sphericalangle(B'A, B'C')$$
$$\sphericalangle(CB, CA) = \sphericalangle(C'B', C'A)$$

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.

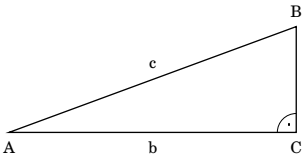
Ogni triangolo può essere considerato (talvolta anche in più modi - quando?) come unione di due triangoli rettangoli.



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 $\sphericalangle(BA, BC) = 90°$.



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

$$a^2 + b^2 = c^2.$$

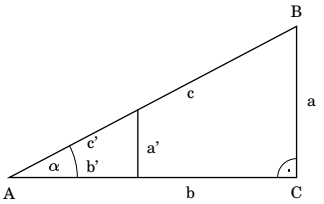
Dimostrazione: Pag. 9.

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:

$$c = \sqrt{a^2 + b^2}$$
$$a = \sqrt{c^2 - b^2}$$
$$b = \sqrt{c^2 - a^2}$$

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

$$\frac{a'}{c'} = \frac{a}{c}$$
$$\frac{b'}{a'} = \frac{b}{a}$$
$$\frac{c'}{b'} = \frac{c}{b}$$

Questi rapporti sono perciò funzioni dell'angolo α che vengono dette funzioni trigonometriche e denotate come segue:

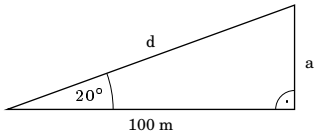
$$\sin\alpha := \frac{a}{c} \ldots \text{seno di } \alpha$$
$$\cos\alpha := \frac{b}{c} \ldots \text{coseno di } \alpha$$
$$\tan\alpha := \frac{a}{b} \ldots \text{tangente di } \alpha$$
$$\cot\alpha := \frac{b}{a} \ldots \text{cotangente di } \alpha$$

Dalle definizioni seguono le relazioni

$$a = c\sin\alpha = b\tan\alpha$$
$$b = c\cos\alpha = a\cot\alpha$$
$$c = \frac{a}{\sin\alpha} = \frac{b}{\cos\alpha}$$

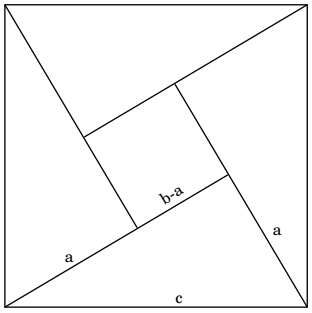
Esercizio 1: Calcolare $\sin 45°$, $\cos 45°$, $\tan 45°$, $\cot 45°$.

Esercizio 2: 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:



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

$$c^2 = (b-a)^2 + 4\frac{ab}{2},$$

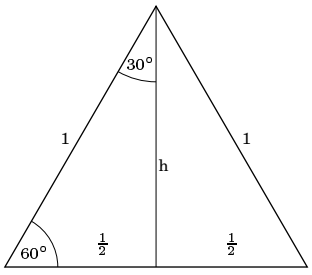
cioè

$$c^2 = b^2 - 2ab + a^2 + 2ab = b^2 + a^2.$$

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°.

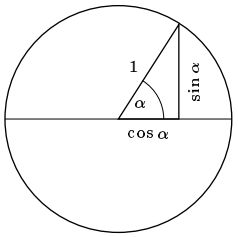


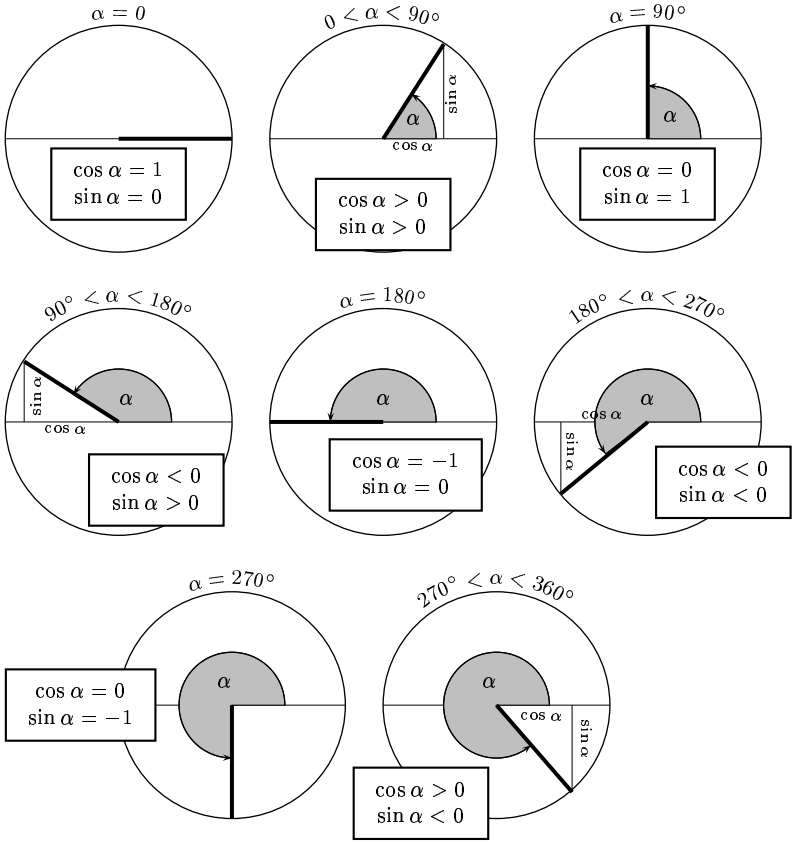
Dalla figura otteniamo

$$\begin{aligned} h &= \sqrt{1-\tfrac{1}{4}} = \frac{\sqrt{3}}{2} \\ \sin 60^\circ &= \cos 30^\circ = \frac{\sqrt{3}}{2} \\ \sin 30^\circ &= \cos 60^\circ = \frac{1}{2} \\ \tan 60^\circ &= 2h = \sqrt{3} \\ \tan 30^\circ &= \frac{1}{2h} = \frac{\sqrt{3}}{3} \end{aligned}$$

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 alla condizione $0 < \alpha < 90^\circ$ come finora. Definiamo prima $\sin\alpha$ e $\cos\alpha$ per ogni α con $0 \le \alpha \le 360^\circ$ come nelle seguenti figure:





Definiamo poi ogni volta

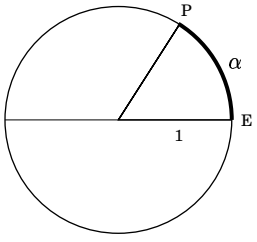
$$\tan\alpha := \frac{\sin\alpha}{\cos\alpha} \qquad \cot\alpha := \frac{\cos\alpha}{\sin\alpha}$$

quando $\cos\alpha \neq 0$ risp. $\sin\alpha \neq 0$. Si vede subito che questa definizione coincide con quella data a pag. 8, quando $0 < \alpha < 90^\circ$.

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 $\alpha = n \cdot 360^\circ + \alpha_0$ con $0 \le \alpha_0 < 360^\circ$ e possiamo definire $\cos\alpha := \cos\alpha_0$, $\sin\alpha := \sin\alpha_0$, $\tan\alpha := \tan\alpha_0$, $\cot\alpha := \cot\alpha_0$.

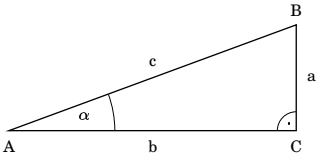
In matematica si identifica l'angolo con la lunghezza dell'arco descritto sulla circonferenza tra i punti E e P della figura a lato, aggiungendo però multipli del perimetro della circonferenza se l'angolo è immaginato ottenuto dopo essere girato più volte attorno al centro. Se il centro del cerchio è l'origine $(0,0)$ del piano, possiamo assumere che $E = (1,0)$. Siccome il perimetro della circonferenza di raggio 1 è 2π, si ha $360^\circ = 2\pi$.



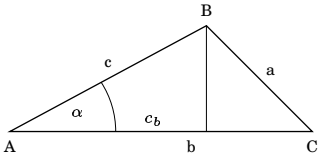
È chiaro che un angolo di g° è uguale a $\dfrac{g}{360}2\pi$, in altre parole $g^\circ = \dfrac{2\pi g}{360}$, e viceversa $\alpha = \alpha\dfrac{360^\circ}{2\pi}$ per ogni $\alpha \in \mathbb{R}$. Infatti $1 = \dfrac{360^\circ}{2\pi} \sim 57.29577951^\circ$.

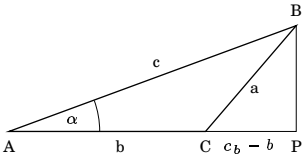
Il teorema del coseno

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:

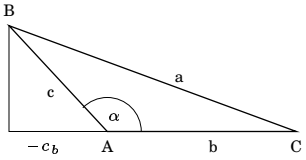


In questo caso $c_b = b$.





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

$$c^2 = a^2 + b^2.$$

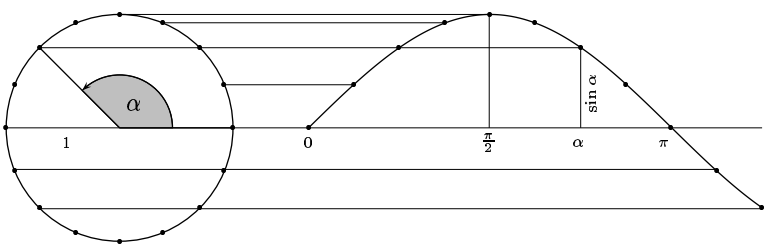
Dimostrazione: Dalla figura in alto a destra a pag. 8 si vede che il triangolo è rettangolo con ipotenusa c se e solo se $b_a = 0$ (oppure, equivalentemente, $a_b = 0$). L'enunciato segue dal teorema precedente.

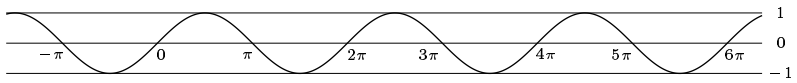
Teorema del coseno:

$$a^2 = b^2 + c^2 - 2bc\cos\alpha$$

Dimostrazione: $c_b = c\cos\alpha$ in tutti e quattro i casi del precedente teorema (cfr. le definizioni degli angoli sul cerchio a pag. 9).

Il grafico della funzione seno





La periodicità di *sin* e *cos*

Dalle definizioni date a pag. 9 segue che

$$\cos(\alpha + 360°) = \cos\alpha$$
$$\sin(\alpha + 360°) = \sin\alpha$$

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 9. 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: Non richiesta. Una dimostrazione geometrica si trova nei libri scolastici, una dimostrazione analitica forse verrà data nei corsi di Analisi.

Esercizio: $\cos\alpha = \sin(\alpha + \frac{\pi}{2})$.

Verificare l'enunciato prima nelle illustrazioni a pag. 9 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 $|\sin\alpha| \le 1$ e $|\cos\alpha| \le 1$. Infatti, se dall'analisi complessa anticipiamo le formule

$$\cos z = \frac{e^{iz} + e^{-iz}}{2}$$
$$\sin z = \frac{e^{iz} - e^{-iz}}{2i}$$

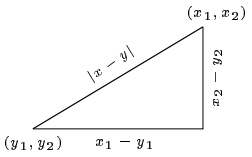
valide per ogni numero complesso z, vediamo che ad esempio $\cos ai = \frac{e^{-a} + e^{a}}{2}$, quindi per a reale e tendente ad infinito (in questo caso e^{-a} tende a 0) $\cos ai$ si comporta come $\frac{e^a}{2}$ e tende quindi fortemente ad infinito.

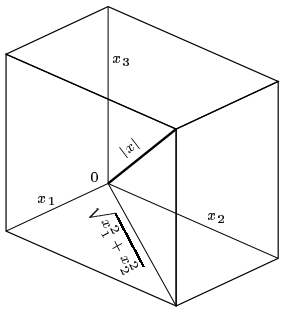
Distanze in $\mathbb{R}^n$

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 $\mathbb{R}^n$. Allora

$$|x - y|^2 = \sum_{k=1}^{n} (x_k - y_k)^2 =$$
$$= \sum_{k=1}^{n} x_k^2 + \sum_{k=1}^{n} y_k^2 - 2 \sum_{k=1}^{n} x_k y_k =$$
$$= |x|^2 + |y|^2 - 2 \sum_{k=1}^{n} x_k y_k.$$

L'espressione $(x, y) := \sum_{k=1}^{n} x_k y_k$ si chiama il **prodotto scalare** dei vettori x ed y. Esso è di fondamentale importanza per tutta la geometria.

Dall'ultima serie di equazioni 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

$$|x - y|^2 = |x|^2 + |y|^2 - 2|x||y| \cos \alpha,$$

da cui

$$(x, y) = |x||y| \cos \alpha.$$

Fare un disegno!

Ortogonalità

La formula fondamentale

$$(x, y) = |x||y| \cos \alpha$$

rimane valida anche se x e y sono uno un multiplo dell'altro, ad esempio $y = tx$ per $t \in \mathbb{R}$, però entrambi $\neq 0$ (ciò implica $t \neq 0$). In questo caso infatti il triangolo determinato da x, y e 0 è degenerato, ma è naturale assegnare all'angolo tra x e y il valore 0 (per cui $\cos \alpha = 1$) se $t > 0$ e invece il valore $180°$ (cosicché $\cos \alpha = -1$) se $t < 0$.

Inoltre $(x, y) = (x, tx) = t(x, x) = t|x|^2$ e $|x||y| = |x||tx| = |t||x||x| = |t||x|^2$. Dimostrare queste relazioni e concludere da soli, stando attenti ai segni.

Quindi se i due vettori sono diversi da zero (ciò implica che anche $|x| \neq 0$ e $|y| \neq 0$), allora essi sono ortogonali (cioè $\alpha = 90°$ oppure $\alpha = 270°$) se e solo se $\cos \alpha = 0$, cioè se e solo se $(x, y) = 0$.

Siccome infine $(x, 0) = 0$ per ogni x, è naturale includere anche il vettore 0 tra i vettori ortogonali ad x. Raccogliendo tutto possiamo perciò dire:

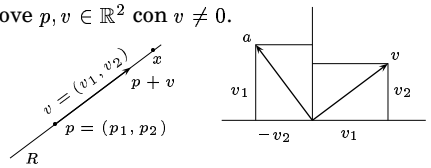
Due vettori x ed y di $\mathbb{R}^n$ sono ortogonali se e solo se $(x, y) = 0$.

Rette nel piano

Una retta R nel piano reale $\mathbb{R}^2$ possiede una rappresentazione parametrica

$$R = \{p + tv \mid t \in \mathbb{R}\},$$

dove $p, v \in \mathbb{R}^2$ con $v \neq 0$.



Il vettore $a = (a_1, a_2) := (-v_2, v_1)$ che si ottiene ruotando v di $90°$ è anch'esso diverso da zero. Un punto $x = (x_1, x_2)$ appartiene alla retta se e solo se il vettore $x - p$ è parallelo a v, cioè se e solo se $x - p$ è ortogonale ad a. Quindi i punti della retta sono esattamente i punti che soddisfano l'equazione

$$a_1(x_1 - p_1) + a_2(x_2 - p_2) = 0, \qquad (*)$$

che può essere scritta anche nella forma

$$a_1 x_1 + a_2 x_2 = a_1 p_1 + a_2 p_2.$$

Siccome però a_1 e a_2 non sono entrambi zero, per ogni $c \in \mathbb{R}$ si trovano facilmente p_1 e p_2 tali che $a_1 p_1 + a_2 p_2 = c$. Ciò mostra che ogni equazione della forma $a_1 x_1 + a_2 x_2 = c$ può essere portata nella forma (*) e descrive perciò una retta ortogonale ad a e quindi parallela a v. La retta $a_1 x_1 + a_2 x_2 = 0$ è tra queste rette tutte parallele ad R quella che passa per l'origine.

Troviamo ad esempio una rappresentazione parametrica per la retta $3x + 5y = 18$. Per $y = 0$ troviamo $x = 6$, quindi $p = (6, 0)$ è un punto della retta che deve essere parallela a $v = (5, -3)$.

Se p e q sono due punti distinti di $\mathbb{R}^2$, una rappresentazione parametrica e l'equazione della retta passante per questi due punti si trovano ponendo $v := q - p$.

ALGORITMI E STRUTTURE DI DATI

Corso di laurea in matematica Anno accademico 2001/2002 Numero 3 ◇ 8 Ottobre 2001

I linguaggi di programmazione

Un linguaggio di programmazione è un linguaggio che permette la realizzazione di algoritmi su un calcolatore. Il calcolatore esegue istruzioni numeriche (diverse a seconda del processore) che insieme formano il linguaggio macchina del calcolatore. Tipicamente queste istruzioni comprendono operazioni in memoria, istruzioni di flusso (test, salti, subroutines, terminazione), definizioni di costanti, accesso a funzioni del sistema operativo, funzioni di input e output.

I numeri che compongono il linguaggio macchina possono essere inseriti direttamente in memoria. Un programma scritto in un altro linguaggio di programmazione deve invece essere convertito in linguaggio macchina; ciò può avvenire prima dell'esecuzione del programma tramite un compilatore che trasforma il programma scritto nel linguaggio sorgente in codice macchina oppure tramite un interprete che effettua una tale trasformazione durante l'esecuzione del programma e solo in parte per quei pezzi del programma che devono essere in quel momento eseguiti. Siccome il codice preparato da un compilatore è già pronto mentre le operazioni dell'interprete devono essere ripetute durante ogni esecuzione, è chiaro che i linguaggi compilati (assembler, C) sono più veloci di quelli interpretati (Perl, Lisp, Apl, Basic).

La velocità del hardware moderno rende meno importanti queste differenze. Spesso, come nel Perl, il linguaggio utilizza moduli (scritti ad esempio in C) già compilati e quindi la traduzione riguarda solo una parte del programma, oppure è possibile, come in Java (e di nascosto anche in Perl), una compilazione in due tempi, che traduce il programma sorgente prima in codice indipendente dalla macchina (linguaggio per una macchina virtuale o bytecode) che su un calcolatore specifico viene poi eseguito da un interprete.

Nonostante che algoritmi e programmi dovrebbero essere separati il più possibile, la sintassi e soprattutto i tipi di dati previsti o definibili di uno specifico linguaggio inducono a paradigmi di programmazione diversi. Molto spesso è possibile, una volta appresi i meccanismi, imitare le costruzioni di un linguaggio anche in altri; per questa ragione è molto utile conoscere linguaggi diversi per avere un bagaglio di tecniche applicabili in molti casi indipendentemente dal linguaggio utilizzato.

Tre problemi semplici

Daremo adesso una prima panoramica dei più diffusi linguaggi di programmazione; nell'ultimo numero del corso verranno presentati il Lisp, il Prolog, il Forth e alcuni linguaggi del passato (Algol, PL/1, Apl, Cobol). Il C è indispensabile e il linguaggio più noto, il Perl il più ricco e il più utile, il Lisp il più profondo, il Java il linguaggio più di moda (e uno di quelli più richiesti dal mercato). Per illustrare il loro uso utilizzeremo i seguenti tre problemi:

1. Calcolo del prodotto scalare di due vettori.
2. Calcolo dei numeri di Fibonacci (pag. 13).
3. Retta passante per due punti distinti p e q nel piano. Risolveremo quest'ultimo compito con l'algoritmo indicato a pag. 11:

(1) Input di $p = (p_1, p_2)$ e $q = (q_1, q_2)$.
(2) $v = (v_1, v_2) = q - p$.
(3) $a = (a_1, a_2) = (-v_2, v_1)$.
(4) $c = a_1 p_1 + a_2 p_2$.
(5) Output dell'equazione nella forma $a_1 x + a_2 y = c$, dove a_1, a_2 e c sono i valori calcolati nei punti (3) e (4).

In questo modo si può naturalmente solo esemplificare la sintassi e non diventano visibili i punti forti o deboli dei linguaggi (strutture di dati, modularità, possibilità di una programmazione funzionale).

Questa settimana

12 I linguaggi di programmazione
Tre problemi semplici
Procedure e funzioni

13 I numeri di Fibonacci
Il sistema di primo ordine
Numeri esadecimali

14 Diagrammi di flusso
Il linguaggio macchina
I linguaggi assembler

15 Basic
Fortran
Pascal

16 C
C++

17 Java
Python
Perl

Procedure e funzioni

Una *procedura* in un programma è una parte del programma che può essere chiamata esplicitamente per eseguire determinate operazioni. In un linguaggio macchina una procedura o *subroutine* è un pezzo di codice, di cui è noto l'indirizzo iniziale e da cui si torna indietro quando nell'esecuzione del codice vengono incontrate istruzioni di uscita. Una procedura può dipendere da parametri (detti anche argomenti); una procedura che restituisce (in qualche senso intuitivo) un risultato si chiama *funzione* e corrisponde nell'utilizzo alle funzioni o applicazioni della matematica. Una differenza importante è, nei linguaggi non puramente funzionali, che procedure e funzioni spesse eseguono delle operazioni i cui effetti non sono sempre rilevabili dal risultato; in tal caso la dimostrazione della correttezza di un algoritmo è più complicata di una normale dimostrazione matematica. Per questo oggi si cerca di sostituire il più possibile gli algoritmi procedurali tradizionali con tecniche che utilizzano solo funzioni e operazioni matematiche con funzioni (composizione di funzioni, formazione di coppie).

Una procedura o funzione in un programma si chiama *ricorsiva* se chiama se stessa. Il concetto di ricorsività è molto importante in matematica, informatica, logica e forse in molti comuni ragionamenti umani. Calcoliamo il fattoriale $n! := 1 \cdot 2 \cdot 3 \cdots n$ di un numero intero n tramite una funzione ricorsiva in Perl:

```
sub fatt {my $n=shift;
  if ($n> 1) {$n*fatt($n-1)}
  else {1}}
```

I numeri di Fibonacci

La successione dei numeri di Fibonacci è definita dalla ricorrenza $F_0 = F_1 = 1, F_n = F_{n-1} + F_{n-2}$ per $n \geq 2$. Quindi si inizia con due volte 1, poi ogni termine è la somma dei due precedenti: $1, 1, 2, 3, 5, 8, 13, 21, \ldots$. Un programma iterativo in Perl per calcolare l'n-esimo numero di Fibonacci:

```
sub fib1 {my $n=shift; my ($a,$b,$k);
  return 1 if $n< =1;
  for ($a=$b=1,$k=2;$k< =$n;$k++)
  {($a,$b)=($a+$b,$a)} $a}
```

Possiamo visualizzare i numeri di Fibonacci da F_0 a F_{20} e da F_{50} a F_{60} con la seguente funzione:

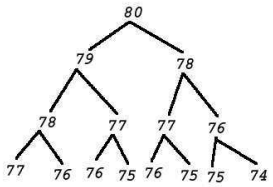
```
sub visfibonacci
  {for (0..20,50..60) {printf("%3d %-12.0f\n",$_,fib1($_))}}
```

La risposta è fulminea.

La definizione stessa dei numeri di Fibonacci è di natura ricorsiva, e quindi sembra naturale usare invece una funzione ricorsiva:

```
sub fib2 {my $n=shift;
  return 1 if $n< =1;
  fib2($n-1)+fib2($n-2)}
```

Se però adesso nella funzione **fibonacci** sostituiamo *fib1* con *fib2*, ci accorgiamo che il programma si blocca dopo la serie dei primi 20 numeri di Fibonacci, cioè che anche i velocissimi Pentium non sembrano in grado di calcolare F_{50}. Infatti qui incontriamo il fenomeno di una ricorsione con *sovrapposizione dei rami*, cioè una ricorsione in cui le operazioni chiamate ricorsivamente vengono eseguite molte volte. Questo fenomeno è frequente nella ricorsione doppia e diventa chiaro se osserviamo l'illustrazione a lato che mostra lo schema secondo il quale avviene ad esempio il calcolo di F_{80}. Si vede che F_{78} viene calcolato due volte, F_{77} tre volte, F_{76} cinque volte, ecc. (si ha l'impressione che riappaia la successione di Fibonacci e infatti è così, quindi un numero esorbitante di ripetizioni impedisce di completare la ricorsione). È noto che F_n è approssimativamente (con un errore minore di 0.5) uguale a $\frac{1}{\sqrt{5}}(\frac{1+\sqrt{5}}{2})^{n+1}$ e quindi si vede che questo algoritmo è di complessità esponenziale.



Il metodo del sistema di primo ordine

Più avanti in analisi si imparerà che un'equazione differenziale di secondo ordine può essere ricondotta a un sistema di due equazioni di primo ordine. In modo simile possiamo trasformare la ricorrenza di Fibonacci in una ricorrenza di primo ordine in due dimensioni. Ponendo $x_n := F_n, y_n := F_{n-1}$ otteniamo il sistema

$$x_{n+1} = x_n + y_n$$
$$y_{n+1} = x_n$$

per $n \geq 0$ con le condizioni iniziali $x_0 = 1, y_0 = 0$ (si vede subito che ponendo $F_{-1} = 0$ la relazione $F_{n+1} = F_n + F_{n-1}$ vale per $n \geq 1$). Per applicare questo algoritmo dobbiamo però in ogni passo generare due valori, avremmo quindi bisogno di una funzione che restituisce due valori numerici. Ciò in Perl, dove una funzione può restituire come risultato una lista, è molto facile:

```
sub fib3 {my $n=shift; my ($x,$y);
  return (1,0) if $n==0;
  ($x,$y)=fib3($n-1); ($x+$y,$x)}
```

Per la visualizzazione dobbiamo ancora modificare la funzione **fibonacci** nel modo seguente:

```
sub visfibonacci {my ($x,$y);
  for (0..20,50..60)
  {($x,$y)=fib3($_);
  printf("%3d %-12.0f\n",$_,$x)}}
```

Numeri esadecimali

Nei linguaggi macchina e assembler 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
```

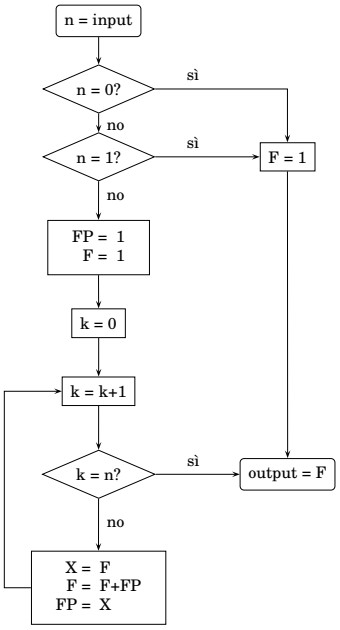
In questo modo adesso possiamo scrivere $2789 = (AE5)_{16}$. In genere si possono usare anche indifferentemente le corrispondenti lettere minuscole. Si noti che `(F)`$_{16}$ `= 15` assume nel sistema esadecimale lo stesso ruolo come il `9` nel sistema decimale. Quindi `(FF)`$_{16}$ `= 255` $= 16^2 - 1 = 2^8 - 1$. Un numero naturale n con $0 \leq n \leq 255$ 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.

Diagrammi di flusso

Algoritmi semplici possono essere espressi mediante diagrammi di flusso. Illustriamo questo metodo con un algoritmo per i numeri di Fibonacci.



La tecnica dei diagrammi di flussi oggi viene usata pochissimo; è lenta e poco efficiente e non adatta alla rappresentazione di strutture di dati complesse. Può essere utile (nei primi giorni) al programmatore principiante per apprendere in un modo visivo alcuni meccanismi della programmazione procedurale.

Talvolta, soprattutto nei libri sugli algoritmi, si usano pseudolinguaggi, cioè linguaggi che non sono veri linguaggi di programmazione e nemmeno formalmente definiti in tutti i dettagli, ma le cui istruzioni e costruzioni hanno un significato facilmente intuibile. In un tale pseudolinguaggio il nostro diagramma di flusso potrebbe essere così tradotto:

```
          n=input
          if n=0 then goto uno
          if n=1 then goto uno
          FP=1
          F=1
          k=0
ciclo:    k=k+1
          if k=n then goto fine
          X=F
          F=F+FP
          FP=X
          goto ciclo
uno:      F=1
fine:     output=F
```

La rappresentazione di un algoritmo mediante un pseudolinguaggio è spesso adeguata quando si vuole studiare la complessità dell'algoritmo, perché in genere le istruzioni del pseudolinguaggio corrispondono in un modo trasparente alle operazioni effettivamente eseguite dal processore. D'altra parte però questo tipo di rappresentazione induce facilmente il programmatore a una concezione procedurale tradizionale degli algoritmi.

Si noti che nel diagramma di flusso e nel pseudolinguaggio abbiamo usato il segno di uguaglianza sia per le assegnazioni (ad esempio `k=k+1` ovviamente non può esprimere un'uguaglianza ma significa che il valore di `k` viene modificato e posto uguale a uno in più di quanto era prima) sia nei test di uguaglianza. Per distinguere i due utilizzi molti linguaggi (tra cui il C, il Perl e il Java) usano un doppio segno di uguaglianza (`==`) nei test di uguaglianza.

Il linguaggio macchina

Proviamo a scrivere nel linguaggio macchina del 6502 l'algoritmo per la moltiplicazione con 10 di un byte n. L'aritmetica nel 6502 avviene modulo 256, quindi, affinché il risultato sia corretto, deve valere $0 \le n \le 25$.

Utilizziamo la moltiplicazione russa, che verrà spiegata più avanti, per questo caso particolare: Formiamo $p := 2x$, poi raddoppiamo n due volte (cosicché n adesso ha 8 volte il suo valore iniziale) e aggiungiamo p.

Per n usiamo l'indirizzo esadecimale `F0`, per la variabile p l'indirizzo `F1`. Nel 6502 tutte le operazioni aritmetiche avvengono nell'accumulatore; dobbiamo quindi trasferire il contenuto di `F0` nell'accumulatore mediante l'istruzione `A5 F0`. Le istruzioni del 6502 consistono di al massimo 3 bytes, di cui il primo indica l'operazione, i rimanenti gli argomenti.

Disabilitiamo con `D8` la modalità decimale, poi con `0A` moltiplichiamo il contenuto a dell'accumulatore per due (in verità questa istruzione sposta i bits nella rappresentazione binaria di a di una posizione a sinistra e pone l'ultimo bit uguale a 0). Memorizziamo il nuovo valore in `F1` con `85 F1`, poi effettuiamo due raddoppi con `0A 0A`. Mettiamo il riporto uguale a zero con `18` (esadecimale) e addizioniamo all'accumulatore il contenuto di `F1` con `65 F1`; infine trasferiamo il valore dell'accumulatore nella locazione di memoria `F0` con `85 F0`. Il programma completo in linguaggio macchina a questo punto è

```
A5 F0 D8 0A 85 F1 0A 0A 18 65 F1 85 F0.
```

I linguaggi assembler

Il linguaggio macchina del processore 6502, utilizzato da alcuni dei più famosi personal computer di tutti i tempi (Apple II, Commodore, Atari) all'inizio degli anni '80, era elegante ed efficiente e molto istruttivo, perché era molto facile lavorare direttamente in memoria. Quei computer, piuttosto lenti se programmati in Basic, si rivelavano vivacissimi se programmati in linguaggio macchina (o in assembler). Il processore ha ancora oggi i suoi appassionati e una pagina Web (*www.6502.org/*) offre esempi di codice, descrizioni delle istruzioni e links ad altre pagine Web.

Come abbiamo visto, è possibile inserire direttamente un programma in linguaggio macchina in memoria. La difficoltà non consiste tanto nel memorizzare i codici (almeno per il 6502 che ha relativamente poche istruzioni), perché è facile ricordarseli dopo pochi giorni di pratica, ma piuttosto nella manutenzione del programma (modifiche, documentazione). Se ad esempio si volesse inserire un nuovo pezzo di programma in un certo punto, bisogna non solo spostare una parte del programma (ciò è possibile mediante appositi comandi dal terminale), ma è necessario anche aggiustare tutti gli indirizzi dei salti.

Per questa ragione sono stati inventati i linguaggi assembler. Così si chiamano linguaggi le cui istruzioni fondamentali corrispondono esattamente alle istruzioni in linguaggio macchina, ma sono espresse in una forma non numerica più facile da ricordare, e inoltre permettono l'utilizzo di nomi simbolici per gli indirizzi. Gli assembler più recenti (ad esempio per i processori Intel) sono molto complessi e forniscono moltissime funzioni.

Il programma in linguaggio macchina in basso a sinistra viene tradotto così in assembler:

```
            ; questo è un commento
CLD         ; clear decimal
LDA  $F0    ; load accumulator
ASL         ; arithmetic shift left
STA  $F1    ; store accumulator
ASL
ASL
CLC         ; clear carry
ADC  $F1    ; add with carry
STA  $F0
```

Utilizzando nomi simbolici per gli indirizzi, potremmo scrivere:

```
    CLD
    LDA  N
    ASL
    STA  P
    ASL
    ASL
    CLC
    ADC  P
    STA  N
```

Basic

Il Basic è stato stato uno dei primi e più popolari linguaggi di programmazione per PC. Concettualmente esprime una programmazione orientata ai diagrammi di flusso ed ha un po' gli stessi svantaggi. Anche dal punto di vista della sintassi è molto limitato: i nomi delle variabili possono avere solo due lettere, non ci sono variabili locali e non ci sono veri sottoprogrammi o funzioni. Ci sono molti dialetti e quindi il linguaggio è anche poco portatile. Alcuni dei dialetti superano in parte le limitazioni del Basic classico ma sono molto complicati e più difficili da imparare dei linguaggi professionali come C/C++, Perl e Java. Risolviamo i nostri tre compiti in Basic classico.

Prodotto scalare

```
 10   dim a(3)
 20   dim b(3)
 30   for i=0 to 3
 40   a(i)=i+1
 50   next i
 60   for i=0 to 3
 70   b(i)=(i+1)*(i+1)
 80   next i
100   p=0
110   for i=0 to 3
120   p=p+a(i)*b(i)
130   next i
140   print p
```

Numeri di Fibonacci

```
 10   print "n = ";
 12   input n
 20   if n=0 then 200
 30   if n=1 then 200
 40   fp=1
 50   f=1
 60   k=0
 70   k=k+1
 80   if k=n then 210
 90   x=f
100   f=f+fp
110   fp=x
120   goto 70
200   f=1
210   print f
```

Retta passante per p e q

```
 10   print "p1 = "
 12   input p1
 20   print "p2 = "
 22   input p2
 30   print "q1 = "
 32   input q1
 40   print "q2 = "
 42   input q2
 50   v1=q1-p1:  v2=q2-p2
 60   a1=-v2:  a2=v1
 70   c=a1*p1+a2*p2
 80   print a1; "x + "; a2;
 82   print "y = "; c
```

Il punto e virgola nei comandi di stampa non indica, come in altri linguaggi, la fine di un comando, ma un output senza passaggio a una nuova riga.

Fortran

Il Fortran è insieme al Lisp uno dei più vecchi linguaggi di programmazione ancora in uso. Anch'esso induce a una programmazione strettamente procedurale e ha molte limitazioni; è largamente diffuso però in ambiente ingegneristico e numerico nonostante i molti difetti, soprattutto perché esistono vaste (e preziose) librerie di software. Se il programma sorgente viene scritto in un file **alfa.f**, sotto Linux il comando

f77 -ffree-form alfa.f -o alfa

crea un file eseguibile **alfa**.

Prodotto scalare

```
real a(4), b(4) integer i,p
do 10 i=1,4
c Basic 0..3, Fortran 1..4
a(i)=i
10 continue
do 20 i=1,4
b(i)=i*i
20 continue
p=0
do 30 i=1,4
p=p+a(i)*b(i)
30 continue
write(*,*) p
end
```

Retta passante per p e q

```
real p1,p2,q1,q2,v1,v2,a1,a2,c
write(*,20) 'p1,p2,q1,q2:'
read(*,10) p1,p2,q1,q2
v1=q1-p1
v2=q2-p2
a1=-v2
a2=v1
c=a1*p1+a2*p2
write(*,30) a1,'x + ',a2,'y = ',c
10 format(f9.2)
20 format(a)
30 format(f9.2,a4,f9.2,a4,f9.2)
end
```

Pascal

Il Pascal è stato sviluppato negli anni '70 da Niklaus Wirth. È un linguaggio procedurale in cui però molta importanza è attribuita ai tipi di dati, anche complessi, e alla programmazione strutturale. Wirth, professore di informatica a Zurigo, è anche autore di un famoso libro sugli algoritmi e strutture di dati.

Il Pascal è un po' meno flessibile e meno potente del C e quindi meno popolare tra i programmatori; è invece spesso insegnato nei corsi universitari. Un buon compilatore Pascal libero che funziona anche con Linux si trova ad esempio in *gd.tuwien.ac.at/languages/pascal/fpc/www/*. Il file *fpc-1.0.4-1.i386.rpm* può essere installato con

rpm -Uvh fpc-1.0.4-1.i386.rpm

(ed eliminato con **rpm -e fpc**) e permette di invocare **man fpc** per le istruzioni sull'uso del compilatore. Per un programma semplice contenuto nel file **alfa.pas** il comando **fpc alfa** crea l'eseguibile **alfa** e un file oggetto **alfa.o** (che può essere rimosso). Dallo stesso sito si può prelevare anche la documentazione completa che viene installata in **/usr/doc**.

```
program scalare;
type vettore4=array[1..4] of real;
var i: integer; a,b: vettore4;
  p: real;
function scal (x,y: vettore4): real;
var p:  real; i:  integer;
begin
p:=0;
for i:=1 to 4 do p:=p+x[i]*y[i];
scalare:=p;
end;
begin
for i:=1 to 4 do a[i]:=i;
for i:=1 to 4 do b[i]:=i*i;
p:=scal(a,b);
writeln(p:6:2); // formato f6.2
end.
```

Si vede la meticolosa dichiarazione delle variabili e la forte strutturazione imposta dal programma già in un esempio così semplice. Si noti anche che, a differenza da altri linguaggi, l'istruzione di assegnazione usa il simbolo `:=` invece del semplice `=`, imitando la notazione che in matematica si usa per indicare l'uguaglianza per definizione (pag. 3). Come in C l'`i`-esima componente di un vettore `a` viene indicata con `a[i]`.

Delphi è un ambiente di sviluppo abbastanza diffuso per la programmazione orientata agli oggetti in Pascal sotto Windows.

C

Un programma in C o C++ in genere viene scritto in più files, che conviene raccogliere nella stessa directory. Avrà anche bisogno di files che fanno parte dell'ambiente di programmazione o di una nostra raccolta di funzioni. Tutto insieme si chiama un progetto. I files del progetto devono essere compilati e collegati (*linked*) per ottenere un file eseguibile (detto spesso applicazione). Il programma in C/C++ costituisce il codice sorgente (*source code*) di cui la parte principale è contenuta in files che portano l'estensione **.c**, mentre un'altra parte, soprattutto le dichiarazioni generali, è contenuta in files che portano l'estensione **.h** (da *header*, intestazione).

Il C può essere considerato come un linguaggio macchina universale, le cui operazioni hanno effetti diretti in memoria, anche se la locazione concreta degli indirizzi non è nota al programmatore. Il compilatore deve quindi sapere quanto spazio deve riservare alle variabili (e solo a questo serve la dichiarazione del tipo delle variabili) e di quanti e di quale tipo sono gli argomenti delle funzioni.

I comandi di compilazione sotto Unix possono essere battuti dalla shell, ma ciò deve avvenire ogni volta che il codice sorgente è stato modificato e quindi consuma molto tempo e sarebbe difficile evitare errori. In compilatori commerciali, soprattutto su altri sistemi, queste operazioni vengono spesso eseguite attraverso un'interfaccia grafica; sotto Unix normalmente questi comandi vengono raccolti in un cosiddetto *makefile*, che viene successivamente eseguito mediante il comando **make**.

Tra le dichiarazioni del C e la specifica del tipo di una variabile in Basic o Perl esiste una sostanziale differenza. In C le dichiarazioni hanno essenzialmente il solo scopo di indicare la quantità di memoria che una variabile occupa, ma non vincolano il programmatore a usare quella parte della memoria in un modo determinato. Il codice

```
double x;
sprintf(&x,"Alberto");
puts(&x);
```

provocherà, su molti compilatori, un avvertimento (*warning*) perché l'uso dello spazio occupato dalla variabile reale *x* per contenere una stringa è insolito, ma non genera un errore. Se con

```
double x;
sprintf((char*)&x,"Alberto");
puts((char*)&x);
```

si informa il compilatore delle proprie intenzioni, non protesterà più. In Basic e in Perl invece i tipi servono per definire le operazioni da eseguire su quelle variabili, mentre la gestione della memoria è compito dell'interprete.

Presentiamo adesso un programma completo in C che contiene tre funzioni che corrispondono ai problemi posti a pag. 12.

```
// alfa.c
# include <stdio>
# include <stdlib>

void fibonacci(),retta(),scalare();

double input();

int main();
////////////////////
int main()
{scalare(); fibonacci(); retta();
exit(0);}

void fibonacci()
{char input[40]; int n,k; double a,b,c;
printf("Inserisci n: ");
fgets(input,38,stdin);
n=atoi(input);
if (n< =1) {a=1; goto fine;}
for (a=b=1,k=2;k< =n;k++)
{c=a; a+=b; b=c;}
fine: printf("%.0f\n",a);}

void retta()
{double p1,p2,q1,q2,v1,v2,a1,a2,c;
p1=input("p1"); p2=input("p2");
q1=input("q1"); q2=input("q2");
v1=q1-p1; v2=q2-p2; a1=-v2; a2=v1;
c=a1*p1+a2*p2;
printf("%.2fx + %.2fy = %.2f\n",
  a1,a2,c);}

void scalare()
{double a[4],b[4],p; int i;
for (i=0;i< 4;i++)
{a[i]=i+1; b[i]=(i+1)*(i+1);}
for (p=0,i=0;i< 4;i++) p+=a[i]*b[i];
printf("%9.2f\n",p);}

double input (char *A)
{char input[40];
printf("Inserisci %s: ",A);
fgets(input,38,stdin);
return atof(input);}
```

Il commento nella prima riga indica il nome del file - ciò è utile nella stampa. In questo caso semplice possiamo compilare direttamente dalla shell con i comandi

```
gcc -c alfa.c
gcc -o alfa alfa.o
```

Il primo comando crea il file oggetto **alfa.o**, il secondo il programma eseguibile **alfa**.

C++

Nato e cresciuto insieme a Unix e perfezionato dai migliori programmatori del mondo, il C è probabilmente il linguaggio più su misura per il programmatore. Come Unix invita a una programmazione che utilizza molte piccole funzioni che il programmatore esperto riesce a scrivere rapidamente e che possono essere raccolte in un modo estremamente efficiente. Il programma a sinistra apparentemente è il più complicato di quelli visti finora per i nostri tre problemi; tra l'altro abbiamo scritto una apposita funzione (*input*) per l'immissione di un numero dalla tastiera. Uno dei vantaggi del C è che queste funzioni ausiliarie possono essere scritte e gestite con grande facilità. Il C, pur rimanendo fedele al lavoro diretto in memoria, permette la creazione di dati molto complessi.

Molti programmatori in C riescono quindi a realizzare nei loro programmi i concetti della programmazione orientata agli oggetti. Il C++, sviluppato più tardi del C, mette più chiaramente in evidenza questi concetti, tra cui il più importante è quello di classe. Una **classe** nel C++ possiede componenti che possono essere sia dati che funzioni (*metodi*). Se *alfa* è una classe, ogni elemento di tipo *alfa* si chiama un **oggetto** della classe. Così

```
class vettore {public: double x,y,z;
  double lun()
  {return sqrt(x*x+y*y+z*z);}};
```

definisce una classe i cui oggetti rappresentano vettori tridimensionali e contengono, oltre alle tre componenti, anche la funzione che calcola la lunghezza del vettore. Le componenti vengono usate come nel seguente esempio:

```
void prova ()
{vettore v;
v.x=v.y=2; v.z=1;
printf("%.3f\n",v.lun());}
```

L'indicazione *public:* (non dimenticare il doppio punto) fa in modo che le componenti che seguono sono visibili anche al di fuori della classe; con *private:* invece si ottiene il contrario. *private:* e *public:* possono apparire nella stessa classe; l'impostazione di default è *private:*. Una tipica classe per una libreria grafica potrebbe essere

```
class rettangolo {public: double x,y,dx,dy;
  void disegna(),sposta(double,double);};
```

È difficile decidere se preferire il C o il C++. I due linguaggi sono altamente compatibili (nel senso che molto spesso i programmi scritti in C possono essere direttamente utilizzati per il C++), il C++ è in un certo senso più confortevole, il C più adatto alla programmazione di sistema sotto Unix.

Java

Java è un linguaggio di programmazione orientato agli oggetti che eredita alcune sue caratteristiche dal C++.

Uno degli aspetti più importanti della programmazione orientata agli oggetti è la *comunicazione*: In un certo senso gli oggetti si informano tra di loro sulla propria posizione e sul proprio stato, sulle attività e funzioni. Questa comunicazione nel programma si riflette poi in una comunicazione tra le persone coinvolte nel progetto, per questo la programmazione orientata agli oggetti è particolarmente adatta al lavoro di gruppo. Essa è estroversa, mentre la programmazione ricorsiva e creativa è, almeno in parte, più introversa.

Bisogna dire che talvolta (ovviamente anche per ragioni commerciali) l'importanza di nuovi paradigmi di programmazione, in particolare della programmazione orientata agli oggetti, e di strutture innovative nei linguaggi, viene spesso esagerata. Un buon programmatore in C riesce facilmente a creare quasi tutte queste strutture e il tempo perso a imparare il linguaggio di moda potrebbe essere usato per migliorare la propria biblioteca di funzioni.

A differenza dal C++ nel Java funzioni possono essere definite soltanto come componenti di una classe e, come d'uso nella programmazione orientata agli oggetti, vengono allora dette *metodi* della classe. La classe **String** possiede ad esempio un metodo **length** che restituisce la lunghezza della stringa per cui viene chiamato:

```
String nome="Maria Stuarda";
n=nome.length();
```

Le funzioni matematiche sono per la maggior parte metodi della classe **Math** e vengono chiamate ad esempio così:

```
y=Math.sin(x); y=Math.log(x);
y=Math.sqrt(x); y=Math.exp(x);
```

Segue un programma completo per i numeri di Fibonacci; come si vede, le operazioni di input e output sono, a causa dei molti controlli e della rigidità dei tipi di dati, piuttosto difficoltose.

```
import java.io.IOException;

class algoritmi
{public static void main
   (String parametri[])
{fibonacci();}

public static void fibonacci()
{int n=0,k,m,a,b,c;
byte input[]=new byte[40];
String stringa;
System.out.print("Inserisci n: ");
try {m=System.in.read(input,0,38);
stringa=new String(input,0,m-1);
n=Integer.parseInt(stringa);}
catch(IOException e){}
if (n< =1) a=1; else
{for (a=b=1,k=2;k< =n;k++)
{c=a; a+=b; b=c;}}
System.out.println(a);}}
```

Questo programma va scritto in un file **algoritmi.java** (il nome del file deve corrispondere al nome della classe) da cui si ottiene, con **javac algoritmi.java** il file **algoritmi.class** che contiene il bytecode (pag. 12) che può essere eseguito con **java algoritmi**.

Molti dei paradigmi di comunicazione contenuti in Java sono sotto Unix divise tra il sistema operativo e il linguaggio C. Il programmatore in C può liberamente utilizzare i meccanismi avanzati del sistema operativo e quindi non necessita che essi siano contenuti nel linguaggio.

Java offre invece, come molti altri linguaggi di programmazione, una gestione automatica della memoria. Ciò, almeno in certe circostanze, può essere indubbiamente un punto in favore di Java.

I concetti della programmazione orientata agli oggetti in Java sono sicuramente più perfezionati rispetto al C/C++, sono però anche più rigidi e lasciano meno libertà al programmatore. La facilità con cui nel C si possono gestire grandi raccolte di programmi con numerosissimi files è unica e permette al programmatore ben organizzato una programmazione modulare snella ed efficiente.

Unix possiede una potentissima libreria grafica (*Xlib*), anch'essa direttamente utilizzabile in C. Le notevoli capacità grafiche offerte anche da Java hanno però il vantaggio di essere indipendenti dalla piattaforma e di permettere quindi lo sviluppo di programmi ad interfaccia grafica utilizzabili sotto più sistemi operativi.

Python

Molto leggibile negli esempi più semplici, usa l'indentazione per la strutturazione. Ha molte funzioni per le liste, arriva a una programmazione funzionale non completa, la gestione dei molti moduli non è sempre trasparente.

Come il Perl permette l'assegnazione contemporanea anche negli scambi: *(a,b)=(b,a)* invece di *c=a; a=b; b=c;*.

Nonostante l'apparente semplicità in genere non suscita l'entusiasmo degli studenti ed è probabilmente uno di quei linguaggi (ma VisualBasic è peggio) dove chi inizia non impara a programmare seriamente, perché troppi meccanismi rimangono nascosti.

```
#! /usr/bin/python
# alfa
import string

def fibonacci():
   n=string.atoi(raw_input("Inserisci n: "))
   if n< =1: f=1
   else:
    a=b=1
    for k in range(2,n+1): a,b=a+b,a
   print a

def retta():
   p1=string.atof(raw_input("Inserisci p1: "))
   p2=string.atof(raw_input("Inserisci p2: "))
   q1=string.atof(raw_input("Inserisci q1: "))
   q2=string.atof(raw_input("Inserisci q2: "))
   v1=q1-p1; v2=q2-p2; a1=-v2; a2=v1
   c=a1*p1+a2*p2
   print "%6.2fx + %6.2fy = %6.2f" %(a1,a2,c)

def scalare():
   a=(1,2,3,4); b=(1,4,9,16)
   p=0
   for i in range(0,4): p=p+a[i]*b[i]
   print p

fibonacci()
retta()
scalare()
```

Perl

Al Perl sarà dedicato il prossimo numero, per cui ci limitiamo qui a risolvere i nostri tre problemi, ancora in un unico file direttamente eseguibile dalla shell di Unix.

```
#! /usr/bin/perl -w
# alfa (nome del file)

fibonacci(); retta(); scalare();

# A differenza dal Python, in Perl
# le funzioni possono essere anche
# chiamate prima di essere definite.

sub fibonacci {my $n=input("Inserisci n: ");
   my ($a,$b,$k); if ($n< =1) {$a=1; goto fine}
   for ($a=$b=1,$k=2;$k< =$n;$k++)
   {($a,$b)=($a+$b,$a)}
   print "$a\n"}

sub input {print shift; my $a=< stdin> ;
   chop $a; $a}

sub retta {my $punti=input("Inserisci p1 p2 q1 q2: ");
   my ($p1,$p2,$q1,$q2)=split(/\s+/,$punti);
   my ($a1,$a2,$v1,$v2,$c); $v1=$q1-$p1; $v2=$q2-$p2;
   $a1=-$v2; $a2=$v1; $c=$a1*$p1+$a2*$p2;
   printf("%.2fx + %.2fy = %.2f\n",$a1,$a2,$c)}

sub scalare {my @a=(1,2,3,4); my @b=(1,4,9,16);
   my $p; for $k (0..3) {$p+=$a[$k]*$b[$k]}
   print "$p\n"}
```

ALGORITMI E STRUTTURE DI DATI

Corso di laurea in matematica | Anno accademico 2001/2002 | Numero 4 ◇ 15 Ottobre 2001

Programmare in Perl

Il Perl è il linguaggio preferito dagli amministratori di sistema e una felice combinazione di concezioni della linguistica e di abili tecniche di programmazione; è usato nello sviluppo di software per l'Internet e in molte applicazioni scientifiche semplici o avanzate.

Il vantaggio a prima vista più evidente del Perl sul C è che non sono necessarie dichiarazioni per le variabili e che variabili di tipo diverso possono essere liberamente miste, ad esempio come componenti di una lista. Esistono alcune differenze più profonde: nel Perl una funzione può essere valore di una funzione, e il nome di una variabile (compreso il nome di una funzione) può essere usato come stringa. Queste caratteristiche sono molto potenti e fanno del Perl un linguaggio adatto alla programmazione funzionale e all'intelligenza artificiale.

Variabili nel Perl

Il Perl non richiede una dichiarazione per le variabili che distingue invece dall'uso dei simboli *$*, *@* e *%* con cui i nomi delle variabili iniziano. Il Perl conosce essenzialmente tre tipi di variabili: *scalari* (riconoscibili dal *$* iniziale), *liste* (o *vettori* di scalari, riconoscibili dal *@* iniziale) e *vettori associativi* (*hashes*, che iniziano con *%*) di scalari. Iniziano invece senza simboli speciali i nomi delle funzioni e dei riferimenti a files (*filehandles*) (variabili improprie). Quindi in Perl *$alfa*, *@alfa* e *%alfa* sono tre variabili diverse, indipendenti tra di loro. Esempio:

```
#! /usr/bin/perl -w

$a=7; @a=(8,$a,"Ciao");
%a=("Galli",27,"Motta",26);
print "$a\n"; print '$a\n';
for (@a) {print "$_ "}
print "\n", $a{"Motta"}, "\n";
```

con output

```
7
$a\n8 7 Ciao
26
```

Nella prima riga riconosciamo la direttiva tipica degli script di shell (pagina 22) che in questo caso significa che lo script viene eseguito dall'interprete */usr/bin/perl* con l'opzione *-w* (da *warning*, avvertimento) per chiedere di essere avvertiti se il programma contiene parti sospette.

Stringhe sono incluse tra virgolette oppure tra apostrofi; se sono incluse tra virgolette, le variabili scalari e i simboli per i caratteri speciali (ad es. *\n*) che appaiono nella stringa vengono sostituite dal loro valore, non invece se sono racchiuse tra apostrofi.

Il punto e virgola alla fine di un'istruzione può mancare se l'istruzione è seguita da una parentesi graffa chiusa.

A differenza dal C nel Perl ci sono due forme diverse per il *for*. In questo primo caso la variabile speciale *$_* percorre tutti gli elementi della lista *@a*; le parentesi graffe attorno a *{print "$_ "}* sono necessarie nel Perl, nonostante che si tratti di una sola istruzione. Si vede anche che il *print*, come altre funzioni del Perl, in situazioni semplici non richiede le parentesi tonde attorno all'argomento. Bisogna però stare attenti anche con *print* perché ad esempio con *print (3-1)*7* si ottiene l'output *2*, perché viene prima eseguita l'espressione *print(3-1)*, seguita da un'inutile moltiplicazione per *7*. Quindi qui bisogna scrivere *print((3-1)*7)*.

In questo corso parleremo poco delle variabili hash; nell'esempio si vede che *$a{"Motta"}* è il valore di *%a* nella voce *"Motta"*. Si nota con un po' di sorpresa forse che *$a{"Motta"}* non inizia con *%* ma con *$*; la ragione è che il valore della componente è uno scalare dal quale è del tutto indipendente la variabile *$a*.

Questa settimana

18 Programmare in Perl
Variabili nel Perl
Input dalla tastiera

19 Liste
La funzione grep del Perl
Alcuni operatori per liste
Contesto scalare e contesto listale

20 Files e operatore <>
Funzioni del Perl
Moduli
Il modulo files

21 Vero e falso
Operatori logici del Perl
Operatori di confronto
Istruzioni di controllo
map

22 Comandi Emacs
Scrivere programmi con Emacs
Semplici comandi Unix

Input dalla tastiera

Esaminiamo un semplice programma che assumiamo che sia contenuto nel file **alfa**. Dopo il comando *alfa* dalla shell (dobbiamo ricordarci di rendere il file eseguibile con *chmod +x alfa*) ci viene chiesto il nome che possiamo inserire dalla tastiera; il programma ci saluta utilizzando il nome specificato.

```
#! /usr/bin/perl -w
# alfa
use strict 'subs';

print "Come ti chiami? ";
$nome=< stdin> ; chop($nome);
print "Ciao, $nome!\n";
```

Se una riga contiene un # (che però non deve far parte di una stringa), il resto della riga (compreso il #) viene ignorato dall'interprete, con l'eccezione della direttiva *#!* (*shebang*, probabilmente da *shell bang*; *shebang* significa "cosa", "roba", ma anche "capanna") che viene vista prima dalla shell e le indica quale interprete (Perl, Shell, Python, ...) deve eseguire lo script.

L'istruzione *use strict 'subs';* controlla se il programma non contiene stringhe non contenute tra virgolette o apostrofi (*bar words*); in pratica avverte soprattutto quando si è dimenticato il simbolo *$* all'inizio del nome di una variabile scalare.

<stdin> legge una riga dallo standard input, compreso il carattere di invio finale che viene tolto con *chop*, una funzione che elimina l'ultimo carattere di una stringa.

Liste

Una variabile che denota una lista ha un nome che, come sappiamo, inizia con @. I componenti della lista devono essere scalari. Non esistono quindi liste di liste e simili strutture superiori nel Perl (a differenza ad esempio dal Lisp) che comunque possono, con un po' di fatica, essere simulate utilizzando *puntatori* (che in Perl si chiamano *riferimenti*), che tratteremo quando parleremo della programmazione funzionale.

Perciò, e questo è caratteristico per il Perl, la lista *(0,1,(2,3,4),5)* è semplicemente un modo più complicato di scrivere la lista *(0,1,2,3,4,5)*, e dopo

```
@a=(0,1,2); @b=(3,4,5);
@c=(@a,@b);
```

@c è uguale a *(0,1,2,3,4,5)*, ha quindi 6 elementi e non due. Perciò la seguente funzione può essere utilizzata per stampare le somme delle coppie successive di una lista.

```
sub sdue {my ($x,$y);
  while (($x,$y,@_)=@_) {print $x+$y,"\n"}}
```

Perché termina il ciclo del *while* (pag. 21) in questo esempio? A un certo punto la lista @_ rimasta sarà vuota (se all'inizio consisteva di un numero pari di argomenti), quindi l'espressione all'interno del *while* equivarrà a *($x,$y,@_)=()* (in Perl *()* denota la lista vuota), e quindi anche la parte sinistra in essa è vuota; la lista vuota in Perl però ha il valore booleano *falso*.

Il *k*-esimo elemento di una lista *@a* (cominciando a contare da 0) viene denotato con *$a[k]*. *@a[k]* invece ha un significato diverso ed è uguale alla lista il cui unico elemento è *$a[k]*. Infatti le parentesi quadre possono essere utilizzate per denotare segmenti di una lista: *@a[2..4]* denota la lista i cui elementi sono *$a[2]*, *$a[3]* e *$a[4]*, in *@a[2..4,6]* l'elemento *$a[6]* viene aggiunto alla fine del segmento, in *@a[6,2..4]* invece all'inizio.

(2..5) è la lista *(2,3,4,5)*, mentre *(2..5,0,1..3)* è uguale a *(2,3,4,5,0,1,2,3)*.

La funzione grep del Perl

La funzione *grep* viene usata come filtro per estrarre da una lista quelle componenti per cui un'espressione (nel formato che scegliamo il primo argomento di *grep*) è vera. La variabile speciale *$_* può essere usata in questa espressione e assume ogni volta il valore dell'elemento della lista che viene esaminato. Esempi:

```
sub pari {grep {$_%2==0} @_}
sub negativi {grep {$_< 0} @_}
@a=pari(0,1,2,3,4,5,6,7,8);
for (@a) {print "$_ "}
print "\n"; # output 0 2 4 6 8
@a=negativi(0,2,-4,3,-7,-10,9);
for (@a) {print "$_ "}
print "\n"; # output -4 -7 -10
@a=("Ferrara","Firenze","Roma","Foggia");
@a=grep {!/^F/} @a;
for (@a) {print "$_ "}
print "\n"; # output Roma
```

Il cappuccio ^ denota l'inizio della riga, e quindi */^F/* è vera se la riga inizia con *F*; un punto esclamativo anteposto significa negazione.

Alcuni operatori per liste

L'istruzione *push(@a,@b)* aggiunge la lista *@b* alla fine della lista *@a*; lo stesso effetto si ottiene con *@a=(@a,@b)* che è però più lenta e probabilmente in molti casi implica che *@b* viene attaccata a una nuova copia di *@a*. La funzione restituisce come valore la nuova lunghezza di *@a*.

Per attaccare *@b* all'inizio di *@a* si usa invece *unshift(@a,@b)*; anche qui si potrebbe usare *@a=(@b,@a)* che è però anche qui meno efficiente. Anche questa funzione restituisce la nuova lunghezza di *@a*.

shift(@a) risp. *pop(@a)* tolgono il primo risp. l'ultimo elemento dalla lista *@a* e restituiscono questo elemento come valore. All'interno di una funzione si può omettere l'argomento; *shift* e *pop* operano allora sulla lista @_ degli argomenti.

Una funzione più generale per la modifica di una lista è *splice*; l'istruzione *splice(@a,pos,elim,@b)* elimina, a partire dalla posizione *pos* un numero *elim* di elementi e li sostituisce con la lista *@b*. Esempi:

```
@a=(0,1,2,3,4,5,6,7,8);
splice(@a,0,3,"a","b","c");
print "@a\n"; # output a b c 3 4 5 6 7 8
splice(@a,4,3,"x","y");
print "@a\n"; # output a b c 3 x y 7 8
splice(@a,1,6);
print "@a\n"; # output a 8
```

reverse(@a) restituisce una copia invertita della lista *@a* (che non viene modificata dall'istruzione).

$#a è l'ultimo indice valido della lista *@a* e quindi uguale alla sua lunghezza meno uno.

Contesto scalare e contesto listale

In Perl avviene una specie di conversione automatica di liste in scalari e viceversa; se una variabile viene usata come scalare, si dice anche che viene usata in contesto scalare, e se viene usata come lista, si dice che viene usata in contesto listale. In verità è un argomento un po' intricato, perché si scopre che in contesto scalare le liste definite mediante una variabile si comportano diversamente da liste scritte direttamenta nella forma $(a_0, a_1, \ldots, a_n)$. Infatti in questa forma, in contesto scalare, la virgola ha un significato simile a quello dell'operatore virgola del C: se i componenti a_i sono espressioni che contengono istruzioni, queste vengono eseguite; il risultato (in contesto scalare) di tutta la lista è il valore dell'ultima componente.

Il valore scalare di una lista descritta da una variabile è invece la sua lunghezza.

Uno scalare in contesto listale diventa uguale alla lista il cui unico elemento è quello scalare. Esempi:

```
@a=7; # raro
print "$a[0]\n"; # output 7
@a=(8,2,4,7);
$a=@a; print "$a\n"; # output 4
$a=(8,2,4,7);
print "$a\n"; # output 7
@a=(3,4,9,1,5);
while (@a> 2) {print shift @a} # output 349
print "\n";
```

Un esempio dell'uso dell'operatore virgola:

```
$a=4;
$a=($a=2*$a,$a--,$a+=3);
print "$a\n"; # output 10
```

Files e operatore <>

stdin, *stdout* e *stderr* sono i *filehandles* (un tipo improprio di variabile che corrisponde alle variabili del tipo *FILE** del C) che denotano standard input, standard output e standard error. Se *File* è un filehandle, *<File>* è il risultato della lettura di una riga dal file corrispondente a *File*. Esso contiene anche il carattere di invio alla fine di ogni riga.

Può accadere che l'ultima riga di un file non termini in un carattere invio, quindi se usiamo *chop* per togliere l'ultimo carattere, possiamo perdere un carattere. Nell'input da tastiera l'invio c'è sempre, quindi possiamo usare *chop*, come abbiamo visto a pagina 18; altrimenti si può usare la funzione *chomp* che toglie l'ultimo carattere da una stringa solo se è il carattere invio.

Per aprire un file si può usare *open* come nel seguente esempio (lettura di un file e stampa sullo schermo):

```
open(File,"lettera");
while (< File> ) {print $_}
close(File);
```

oppure

```
$/=undef;
open(File,"lettera");
print < File> ;
close(File);
```

Il separatore di fineriga (una stringa) è il valore della variabile speciale *$/* e può essere impostato dal programmatore; di default è uguale a "\n". Se lo rendiamo indefinito con *$/=undef* possiamo leggere tutto il file in un blocco solo come nel secondo esempio.

Per aprire il file *beta* in scrittura si può usare *open(File,"> beta")* oppure, nelle più recenti versioni del Perl, *open(File,"> ","beta")*. La seconda versione può essere applicata anche a files il cui nome inizia con > (una cattiva idea comunque per le evidenti inferenze con il simbolo di redirezione > della shell). Similmente con *open(File,"> > beta")* si apre un file per aggiungere un testo.

Il filehandle diventa allora il primo argomento di *print*, il testo da scrivere sul file è il secondo argomento, come nell'esempio che segue e nelle funzioni *files::scrivi* e *files::aggiungi*.

```
open(File,"> beta");
print File "Ciao, Franco.\n";
close(File);
```

Abbiamo già osservato che la variabile che abbiamo chiamato *File* negli usi precedenti di *open* è impropria; una conseguenza è che questa variabile non può essere usata come variabile interna (mediante *my*) o locale di una funzione, in altre parole non può essere usata in funzioni annidate. Per questo usiamo i moduli *FileHandle* e *DirHandle* del Perl che permettono di utilizzare variabili scalari per riferirsi a un filehandle, come nel nostro modulo *files* che viene descritto a lato.

Funzioni del Perl

Una funzione del Perl ha il formato seguente

```
sub f {...}
```

dove al posto dei puntini stanno le istruzioni della funzione. Gli argomenti della funzione sono contenuti nella lista @_ a cui si riferisce in questo caso l'operatore *shift* che estrae da una lista il primo elemento. Le variabili interne della funzione vengono dichiarate tramite *my* oppure con *local* (che però ha un significato leggermente diverso da quello che uno si aspetta). La funzione può restituire un risultato mediante un *return*, altrimenti come risultato vale l'ultimo valore calcolato prima di uscire dalla funzione. Alcuni esempi tipici che illustrano soprattutto l'uso degli argomenti:

```
sub raddoppia {my $a=shift; $a+$a}
sub somma2 {my ($a,$b)=@_; $a+$b}
sub somma {my $s=0;
  for (@_) {$s+=$_} $s}
print raddoppia(4)," ";
print somma2(6,9)," ";
print somma(0,1,2,3,4),"\n";
```

con output *8 15 10*.

Alcuni operatori abbreviati che vengono usati in C e Perl:

```
$a+=$b ... $a=$a+$b
$a–=$b ... $a=$a–$b
$a*=$b ... $a=$a*$b
$a/=$b ... $a=$a/$b
$a++   ... $a=$a+1
$a–    ... $a=$a–1
```

Moduli

Le raccolte di funzioni in Perl si chiamano *moduli*; è molto semplice crearle. Assumiamo che vogliamo creare un modulo *matematica*; allora le funzioni di questo modulo vanno scritte in un file **matematica.pm** (quindi il nome del file è uguale al nome del modulo a cui viene aggiunta l'estensione **.pm** che sta per *Perl module*). Prima delle istruzioni e funzioni adesso deve venire la dichiarazione *package matematica;*.

Il modulo può contenere anche istruzioni al di fuori delle sue funzioni; per rendere trasparenti i programmi queste istruzioni dovrebbero solo riguardare le variabili proprie del modulo.

Nell'utilizzo il modulo restituisce un valore che è uguale al valore dell'ultima istruzione in esso contenuto; se non ci sono altre istruzioni, essa può anche consistere di un *1;* all'inizio del file (che però non deve essere invalidata da un'altra istruzione che restituisce un valore falso).

Dopo di ciò altri moduli o il programma principale possono usare il modulo matematica con l'inclusione *use matematica;*; una funzione *f* di *matematica* deve essere chiamata con *matematica::f*.

Se alcuni moduli che si vogliono usare si trovano in cartelle α, β, γ che non sono tra quelle nelle quali il Perl cerca di default, si indica ciò con *use lib 'α', 'β', 'γ';*.

Il modulo files

```
1; # files.pm
use DirHandle; use FileHandle;

package files;

sub aggiungi {my ($a,$b)=@_; my $file=new FileHandle;
open($file,"> > $a"); print $file $b; close($file)}

sub leggi {local $/=undef; my $a; my $file=new FileHandle;
if (open($file,shift)) {$a=< $file> ; close($file); $a} else {"}}

sub scrivi {my ($a,$b)=@_; my $file=new FileHandle;
open($file,"> $a"); print $file $b; close($file)}

sub catalogo {my $dir=new DirHandle;
opendir($dir,shift); my @a=grep {!/^\./} readdir($dir);
closedir($dir); @a}
```

In *catalogo* il significato di *opendir* e *closedir* è chiaro; *readdir* restituisce il catalogo della cartella associata con il *dirhandle $dir*, da cui, con un *grep* (pagina 19) il cui primo argomento è un espressione regolare, vengono estratti tutti quei nomi che non iniziano con un punto (cioè che non sono files o cartelle nascosti). Esempi d'uso:

```
use files;

print files::leggi("lettera");

for (files::catalogo('.')) {print "$_\n"}

$catalogo=join("\n",files::catalogo('/'));
files::scrivi("root",$catalogo);
```

Abbiamo usato la funzione *join* per unire con caratteri di nuova riga gli elementi della lista ottenuta con *files::catalogo* in un'unica stringa.

Vero e falso

La verità di un'espressione in Perl viene sempra valutata in contesto scalare. Gli unici valori scalari falsi sono la stringa vuota *“”* e il numero 0. La lista vuota in questo contesto assume il valore 0 ed è quindi anch'essa falsa. Lo stesso vale però per *(0)* e ogni lista scritta in forma esplicita il cui ultimo elemento è 0.

Attenzione: In Perl il numero 0 e la stringa *“0”* vengono identificati (si distinguono solo nell'uso), quindi anche la stringa *“0”* è falsa, benché non vuota. Le stringhe *“00”* e *“0.0”* sono invece vere.

Operatori logici del Perl

Per la congiunzione logica (AND) viene usato l'operatore *&&*, per la disgiunzione (OR) l'operatore | | . Come in molti altri linguaggi di programmazione questi operatori non sono simmetrici; infatti, se A è falso, in A&&B il valore del secondo operando B non viene più calcolato, e lo stesso vale per A | | B se A è vero.

In particolare *if (A&&B)* $\{\alpha\}$ è equivalente a *if (A)* $\{$*if (B)* $\{\alpha;\}\}$ e *if (A| | B)* $\{\alpha;\}$ è equivalente a *if (A)* $\{\alpha\}$ *else* $\{$*if (B)* $\{\alpha\}\}$.

Il punto esclamativo viene usato per la negazione logica; anche *not* può essere usato a questo scopo.

and e *or* hanno una priorità minore di *&&* e | | ; tutti e quattro gli operatori restituiscono l'ultimo resultato calcolato, con qualche piccola sorpresa:

```
$a = 1 and 2 and 3; print “$a\n”;
  # output: 1 (sorpresa!)
$a = (1 and 2 and 3); print “$a\n”; # output: 3
$a = 1 && 2 && 3; print “$a\n”; # output: 3
$a = 0 or “” or 4; print “$a\n”;
  # output: 0 (sorpresa)
$a = (0 or “” or 4); print “$a\n”; # output: 0
$a = 0 || “” || 4; print “$a\n”; # output: 4
$a = 1 and 0 and 4; print “$a\n”;
  # output: 1 (sorpresa)
$a = (1 and 0 and 4); print “$a\n”; # output: 0
$a = 1 && 0 && 4; print “$a\n”; # output: 0
$a = 5 && 7 && “”; print “$a=\n”;
  # Nessun output!
```

Operatori di confronto

Il Perl distingue operatori di confronto tra stringhe e tra numeri. Per il confronto tra numeri si usano gli operatori ==, *!=*, < , > , < =, > =, per le stringhe invece *eq*, *ne*, *lt*, *le*, *gt* e *ge*. Si osservi che, mentre le stringhe *“1.3”*, *“1.30”* e *“13/10”* sono tutte distinte, le assegnazioni *$a=1.3*, *$b=1.30* e *$c=13/10* definiscono le tre variabili come numeri che hanno la stessa rappresentazione come stringhe, come si vede dai seguenti esempi:

```
$a=1.3; $b=1.30; $c=13/10;
print “ok 1\n” if $a==$b; # output: ok 1
print “ok 2\n” if $a==$c; # output: ok 2
print “ok 3\n” if $a eq $c; # output: ok 3
print “ne 4” if “1.3” ne “1.30”; # output: ne 4
```

Istruzioni di controllo

Nelle alternative di un *if* si possono usare sia *else* che *elsif* (come abbreviazione di *else* $\{$*if* ...$\}$):

```
sub sgn {my $a=shift; if ($a< 0) {-1}
  elsif ($a> 0) {1} else {0}}
```

if può anche seguire un'istruzione o un blocco *do*:

α *if* A oppure *do* $\{\alpha; \beta; \gamma\}$ *if* A.

Il *goto* si usa nel modo solito (cfr. pag. 14).

In Perl esistono due forme alquanto diverse del *for*, da un lato l'analogo del *for* del C con una sintassi praticamente uguale, dall'altro il *for* che viene utilizzato per percorrere una lista.

Il *for* classico ha la seguente forma:

for(α;A;β) $\{\gamma;\}$

equivalente a

α;
ciclo: if (A) $\{\gamma; \beta$; *goto ciclo;*$\}$

α, β e γ sono successioni di istruzioni separate da virgole; l'ordine in cui le istruzioni in ogni successione vengono eseguite non è prevedibile. Ciascuno dei tre campi può anche essere vuoto.

while(A) è equivalente a *for(;A;)* e *until(A)* equivalente a *for(;not A;)*.

Da un *for* (o *while* o *until*) si esce con *last* (o con *goto*), mentre *next* fa in modo che si torni ad eseguire il prossimo passaggio del ciclo, dopo esecuzione di β. Quindi

for (;A;β*)* $\{\gamma_1$*; if (B) break;* γ_2*;*$\}$

è equivalente a

ciclo: if (not A) $\{$*goto fuori*$\}$
  γ_1*; if (B)* $\{$*goto fuori*$\}$
  γ_2*;* β*;*
  goto ciclo;
fuori:

mentre

for (;;β*)* $\{\gamma_1$*; if (B)* $\{$*continue*$\}$ $\gamma_2\}$

è equivalente a

ciclo: γ_1*; if (!B)* $\{\gamma_2\}$ β*; goto ciclo;*

Il *last* e il *next* possono essere seguiti da un'etichetta, ad esempio *last alfa* significa che si esce dal ciclo *alfa*, mentre con *next alfa* viene eseguito il prossimo passaggio dello stesso ciclo. Esempio:

```
alfa: for $a (3..7) {for $b (0..20)
  {$x=$a*$b; next if $x%2 or $x< 20;
  print “$x ”; last alfa if $x> 60}}
# output: 24 30 36 42 48 54 60 20 24 28 32 ...
```

Esistono anche le costruzioni *do* $\{...\}$ *while* A e *do* $\{...\}$ *until* A in cui le istruzioni nel blocco vengono eseguite sempre almeno una volta.

In *for $k (@a)* la variabile *$k* (locale per il *for*) percorre la lista *@a*. Se manca *$k* come in *for (@a)*, viene utilizzata la variabile speciale *@_*.

```
for (0..10) {last if $_> 5; print $_}
  # output: 012345

for ($k=0;$k< =10;$k++)
  {next if $k< 5; print $k}
  # output: 5678910

sub max {my $max=shift;
  for (@_) {$max=$_ if $_> $max} $max}
sub min {my $min=shift;
  for (@_) {$min=$_ if $_< $min} $min}
```

map

map è una funzione importante del Perl e di tutti i linguaggi funzionali molto utile nelle costruzioni matematiche. Con *map* da una lista $(a_1, \ldots, a_n)$ si ottiene la lista $f(a_1, \ldots, a_n)$, se f è una funzione a valori scalari anch'essa argomento di *map*.

```
sub quadrato {my $a=shift; $a*$a}
@a=map {quadrato($_)} (0..8);
print “@a\n”;
  # output: 0 1 4 9 16 25 36 49 64
```

Siano dati numeri reali $a_1, \ldots, a_n$. La loro *media aritmetica* è definita come $\frac{a_1+\cdots+a_n}{n}$, la *media geometrica* come $\sqrt[n]{a_1 a_2 \cdots a_n}$, la *media armonica* come $\frac{n}{\frac{1}{a_1}+\cdots+\frac{1}{a_n}}$. Nel secondo caso i numeri devono essere positivi, nel caso della media armonica devono essere tutti diversi da zero. Osserviamo che

$$\log \sqrt[n]{a_1 a_2 \cdots a_n} = \frac{\log a_1 + \cdots + \log a_n}{n},$$

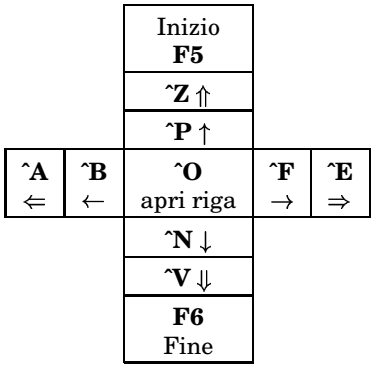
per cui vediamo che il logaritmo della media geometrica è uguale alla media aritmetica dei logaritmi dei numeri a_j. Quindi

$$\sqrt[n]{a_1 a_2 \cdots a_n} = e^{\frac{\log a_1 + \cdots + \log a_n}{n}}.$$

La media armonica è invece uguale al reciproco della media aritmetica dei reciproci degli a_j.

Queste considerazioni matematiche permettono un elegante applicazione di *map* per calcolare la media geometrica e la media armonica dalla media geometrica (esercizi 19 e 20).

Comandi Emacs



elenco buffer aperti	**F1**	incolla	**^Y**
uscire	**F4** (**^XC**)	cancella carattere	**^D**
salvare	**F8** (**^XS**)	cancella ←	**^H**
comando	**F9**	cancella resto riga	**^K**
sostituzione	**F10**	cancella riga	**^AK**
apri file	**TAB** (**^XF**)	cancella da qui	**^TR**
tabulatore	**^Q TAB**	scambiare due righe	**^XT**
inserisci file	**^TI**	cerca →	**^S**
togli buffer	**^TK**	cerca ←	**^R**
tutta la finestra	**F7**	termina comando	**^G**
cambia mezzafinestra	**^CV**	undo	**^TU**
apropos espressione	**^TH A**	maiuscole	**^TM**
apropos tasto	**^TH K**	minuscole	**^TN**
apropos variabile	**^TX**	compila	**Canc**
descrivi comando	**^TW**	esegui alfa	**Fine**
descrivi variabile	**^TV**	goto	↖

Scrivere programmi con Emacs

L'interazione dell'utente con il kernel di Unix avviene mediante la **shell**, un *interprete di comandi* per un suo proprio linguaggio di programmazione. I programmi per la shell rientrano in una categoria molto più generale di programmi, detti *script*, che consistono di un file di testo (che a sua volta può chiamare altri files), la cui prima riga inizia con **#!** a cui segue un comando di script, che può chiamare una delle shell, ma anche il comando di un linguaggio di programmazione molto più evoluto come il **Perl**, comprese le opzioni, ad esempio **#! Perl -w**.

Emacs è l'editor ideale per scrivere i nostri programmi in Perl. Nella nostra interfaccia grafica Emacs può essere invocato tramite *Alt-e*, dalla consolle con **emacs**. Abitarsi ad usare la tastiera e fare a meno del menu (di cui serve solo la funzione *Select and Paste* sotto *Edit*).

Premendo il tasto ↖ sulla riga di comando di Emacs appare *goto:* e si può indicare il numero della riga a cui si vuole andare. Questo è comodo nella programmazione, perché i messaggi d'errore spesso contengono la riga in cui il compilatore ritiene probabile che si trovi l'errore.

Con il tasto *Canc* si ottiene la compilazione del programma, se il file su cui si sta lavorando sotto Emacs fa parte di un progetto. I messaggi di compilazione avvengono su una nuova pagina di Emacs. Con il tasto *Fine* viene invece eseguito il programma **alfa**, se la directory contiene un file eseguibile di questo nome.

Semplici comandi Unix

Login, logout, utente

Ctrl-Alt-Canc	riavvio
poweroff	spegnere
startx	entrare in XX
Ctrl-Alt-Del	uscire da X
passwd	cambio password
exit	chiudere shell

Gestione di files e cartelle

cd	cambiare directory
ls	catalogo
ls -l	catalogo lungo
less	leggere un file
man	istruzioni
c	less o cd
rm	cancellare un file
rm -rf	canc. una directory
mv	spostare o rinominare
mkdir	creare una nuova dir.
cp	copiare

Inserimento dei comandi

^a	inizio riga
^e	fine riga
^f	un carattere avanti
^b	un carattere indietro
^k	cancellare resto riga
^d	cancella carattere
^h	cancella ←
↑	ultimo comando
↓	prossimo comando

Una seduta sotto Unix inizia con l'entrata nel sistema, il *login*. All'utente vengono chiesti il nome di login (l'*account*) e la *password*. Si esce con **exit** oppure, in modalità grafica (**X Window** senza *s* finale, abbreviato **X**) (in cui dalla consolle si entra con *startx*) prima con *Ctrl-Alt-Del*, poi **exit**. A questo punto si può fare un nuovo login. Per spegnere il computer invece di **exit** battere **poweroff**.

Il file system di Unix è gerarchico, dal punto di vista logico i files accessibili sono tutti contenuti nella directory *root* che è sempre designata con /. I livelli gerarchici sono indicati anch'essi con /, per esempio **/alfa** è il file (o la directory) **alfa** nella directory root, mentre **/alfa/beta** è il nome completo di un file **beta** contenuto nella directory **/alfa**. In questo caso **beta** si chiama anche il *nome relativo* del file.

Per entrare in una directory **alfa** si usa il comando **cd alfa**, dove **cd** è un'abbreviazione di *choose directory*. Ogni utente ha una sua directory di login, che può essere raggiunta battendo **cd** da solo. La cartella di lavoro dell'utente X (nome di login) viene anche indicata con **~X**. X stesso può ancora più brevemente denotarla con **~**. Quindi per l'utente X i comandi **cd ~X**, **cd ~** e **cd** hanno tutti lo stesso effetto.

Files il cui nome (relativo) inizia con **.** (detti *files nascosti*) non vengono visualizzati con un normale **ls** ma con **ls -a**. Eseguendo questo comando si vede che il catalogo inizia con due nomi, **.** e **..**. Il primo indica la cartella in cui ci si trova, il secondo la cartella immediatamente superiore, che quindi può essere raggiunta con **cd ..**.

Il comando **mv** (abbreviazione di *move*) ha due usi distinti. Può essere usato per spostare un file o una directory in un'altra directory, oppure per rinominare un file o una directory. Se l'ultimo argomento è una directory, viene eseguito uno spostamento.

ALGORITMI E STRUTTURE DI DATI

Corso di laurea in matematica | Anno accademico 2001/2002 | Numero 5 ◇ 22 Ottobre 2001

L'algoritmo euclideo

Questo algoritmo familiare a tutti e apparentemente a livello solo scolastico, è uno dei più importanti della matematica. Sorprendentemente ciò non vale solo per le sue generalizzazioni ad anelli di polinomi o anelli di numeri, ma è lo stesso algoritmo elementare che impariamo a scuola ad avere numerose applicazioni: in problemi pratici (ad esempio nella grafica al calcolatore), in molti campi avanzati della matematica (teoria dei numeri e analisi complessa), nell'informatica teorica. L'algoritmo euclideo si basa sulla seguente osservazione (lemma di Euclide):

Siano a, b, c, q, d numeri interi e $a = qb + c$. Allora

$$(d|a \text{ e } d|b) \iff (d|b \text{ e } d|c).$$

Quindi i comuni divisori di a e b sono esattamente i comuni divisori di b e c. In particolare le due coppie di numeri devono avere lo stesso massimo comune divisore: $mcd(a, b) = mcd(b, c)$.

Calcoliamo $d := mcd(7464, 3580)$:

$$\begin{aligned}
7464 &= 2\cdot 3580 + 304 &&\Longrightarrow d = mcd(3580, 304)\\
3580 &= 11\cdot 304 + 236 &&\Longrightarrow d = mcd(304, 236)\\
304 &= 1\cdot 236 + 68 &&\Longrightarrow d = mcd(236, 68)\\
236 &= 3\cdot 68 + 32 &&\Longrightarrow d = mcd(68, 32)\\
68 &= 2\cdot 32 + 4 &&\Longrightarrow d = mcd(32, 4)\\
32 &= 8\cdot 4 + 0 &&\Longrightarrow d = mcd(4, 0) = 4
\end{aligned}$$

Si vede che il massimo comune divisore è l'ultimo resto diverso da 0 nell'algoritmo euclideo. L'algoritmo in Perl è molto semplice (dobbiamo però prima convertire i numeri negativi in positivi):

```
sub mcd # $d=mcd($a,$b)
  {my ($a,$b)=@_;
  $a=-$a if $a<0; $b=-$b if $b<0;
  for (;$b;) {($a,$b)=($b,$a%$b)} $a}
```

Si noti nell'ultima riga l'utilizzo dell'assegnazione simultanea nel Perl già vista a pag. 13. Altrettanto semplice è la versione ricorsiva:

```
sub mcd # $d=mcd($a,$b)
  {my ($a,$b)=@_;
  $a=-$a if $a<0; $b=-$b if $b<0;
  return $a if $b==0; mcd($b,$a%$b)}
```

dove usiamo la relazione

$$mcd(a, b) = \begin{cases} a & se \quad b = 0 \\ mcd(b, a\%b) & se \quad b > 0 \end{cases}$$

Il massimo comune divisore

Tutti i numeri considerati $a, b, c, d, \ldots$ sono interi, cioè elementi di $\mathbb{Z}$. Usiamo l'abbreviazione $\mathbb{Z}d := \{nd \mid n \in \mathbb{Z}\}$.

Diciamo che a è un *multiplo* di d se $a \in \mathbb{Z}d$, cioè se esiste $n \in \mathbb{Z}$ tale che $a = nd$. In questo caso diciamo anche che d *divide* a o che d è un *divisore* di a e scriviamo $d|a$.

Dimostrazione del lemma di Euclide: Se $d|a$ e $d|b$, cioè $dx = a$ e $dy = b$ per qualche x, y, allora $c = a - qb = dx - qdy = d(x - qy)$ e vediamo che $d|c$.

E viceversa.

Definizione: Per $(a, b) \neq (0, 0)$ il *massimo comune divisore* di a e b, denotato con $\mathrm{mcd}(a, b)$, è il più grande $d \in \mathbb{N}$ che è un comune divisore di a e b, cioè tale che $d|a$ e $d|b$. Poniamo invece $\mathrm{mcd}(0, 0) := 0$. In questo modo $\mathrm{mcd}(a, b)$ è definito per ogni coppia (a, b) di numeri interi.

Perché esiste $\mathrm{mcd}(a, b)$? Per $(a, b) = (0, 0)$ è uguale a 0 per definizione. Assumiamo che $(a, b) \neq (0, 0)$. Adesso $1|a$ e $1|b$ e se $d|a$ e $d|b$ e ad esempio $a \neq 0$, allora $d \leq |a|$, per cui vediamo che esiste solo un numero finito (al massimo $|a|$) di divisori comuni ≥ 1, tra cui uno ed uno solo deve essere il più grande. $\mathrm{mcd}(a, b)$ è quindi univocamente determinato e uguale a 0 se e solo se $a = b = 0$. Si noti che $d|a \iff -d|a$, per cui possiamo senza perdita di informazioni assumere che $d \in \mathbb{N}$.

Questa settimana

23 L'algoritmo euclideo
Il massimo comune divisore
Nascondere le variabili con my

24 Divisione con resto
Sottogruppi di $\mathbb{Z}$
Primo incontro con gli ideali

25 La moltiplicazione russa
Trovare la rappresentazione binaria
La potenza russa
Tirocini all'ARDSU

26 Lo schema di Horner
Zeri di una funzione continua

Nascondere le variabili con my

Consideriamo le seguenti istruzioni:

```
$a=7;
sub quadrato {$a=shift; $a*$a}
print quadrato(10),"\n";
# output: 100
print "$a\n";
# output: 10
```

Vediamo che il valore della variabile esterna *$a* è stato modificato dalla chiamata della funzione *quadrato* che utilizza anch'essa la variabile *$a*. Probabilmente non avevamo questa intenzione e si è avuto questo effetto solo perché accidentalmente le due variabili avevano lo stesso nome. Per evitare queste collisioni dei nomi delle variabili il Perl usa la specifica **my**:

```
$a=7;
sub quadrato {my $a=shift; $a*$a}
print quadrato(10),"\n";
# output: 100
print "$a\n";
# output: 7
```

In questo modo la variabile all'interno di *quadrato* è diventata una variabile privata o locale di quella funzione; quando la funzione viene chiamata, alla *$a* interna viene assegnato un nuovo indirizzo in memoria, diverso da quello corrispondente alla variabile esterna. Consideriamo un altro esempio:

```
sub quadrato {$a=shift; $a*$a}
sub xpiu1alcubo {$a=shift;
  quadrato($a+1)*($a+1)}
print xpiu1alcubo(2);
# output: 36 invece di 27
```

Viene prima calcolato *quadrato(2+1)*, ponendo la variabile globale *$a* uguale all'argomento, cioè a 3, per cui il risultato finale è $9(3 + 1) = 36$.

Divisione con resto

Proposizione: Siano dati $a, b \in \mathbb{R}$ con $b \neq 0$. Allora esistono univocamente determinati $q \in \mathbb{Z}$ e $r \in \mathbb{R}$ tali che

$$a = qb + r \text{ e } 0 \leq r < |b|.$$

La penultima relazione implica che, se $a, b \in \mathbb{Z}$, anche $r \in \mathbb{Z}$ (e quindi $r \in \mathbb{N}$).

Dimostrazione: Sia $b > 0$. Allora

$$\mathbb{R} = \ldots \dot\cup [-2b, -b) \dot\cup [-b, 0) \dot\cup [0, b) \dot\cup [b, 2b) \dot\cup [2b, 3b) \dot\cup \ldots,$$

cioè

$$\mathbb{R} = \dot{\bigcup_{q \in \mathbb{Z}}} [qb, qb + b) = \dot{\bigcup_{q \in \mathbb{Z}}} [qb, qb + |b|)$$

Se invece $b < 0$, allora

$$\mathbb{R} = \ldots \dot\cup [2b, b) \dot\cup [b, 0) \dot\cup [0, -b) \dot\cup [-b, -2b) \dot\cup [-2b, -3b) \dot\cup \ldots,$$

cioè

$$\mathbb{R} = \dot{\bigcup_{q \in \mathbb{Z}}} [qb, qb - b) = \dot{\bigcup_{q \in \mathbb{Z}}} [qb, qb + |b|)$$

Quindi in entrambi i casi

$$\mathbb{R} = \dot{\bigcup_{q \in \mathbb{Z}}} [qb, qb + |b|)$$

Il punto sul segno di unione significa che si tratta di unioni disgiunte; questo implica che per ogni $a \in \mathbb{R}$ esiste esattamente un $q \in \mathbb{Z}$ per il quale $a \in [qb, qb + |b|)$.

E questo è esattamente l'enunciato della proposizione: trovato q, possiamo porre r uguale a $a - qb$.

r si chiama il *resto* nella divisione di a per b o il resto di a *modulo* b. Nella matematica si scrive spesso $r = a \mod b$. In Perl (e in C) il resto viene calcolato dall'espressione *a%b*, che dà però risultati corretti solo per $a \in \mathbb{N}$ e $b \in \mathbb{N} + 1$. Per questa ragione nelle funzioni *mcd* abbiamo prima convertito eventuali argomenti negativi in positivi.

Sottogruppi di $\mathbb{Z}$

Osservazione 1: Ogni insieme non vuoto A di numeri naturali possiede un minimo (cioè un elemento $d \in A$ con la proprietà che $d \leq a$ per ogni $a \in A$. Questa proprietà importante può essere dimostrata per induzione.

Teorema 1: Ogni sottogruppo H di $\mathbb{Z}$ è della forma $H = \mathbb{Z}d$ per un numero naturale d che è univocamente determinato. Infatti, se $H \neq \{0\}$, allora

$$d = \min\{a > 0 \mid a \in H\},$$

mentre naturalmente d deve essere uguale a 0 se $H = \{0\}$.

Dimostrazione: Sia $H \neq \{0\}$. Allora H possiede un elemento $a \neq 0$ e, se $a < 0$, anche $-a \in H$, e vediamo che possiamo sempre trovare un elemento $a > 0$ con $a \in H$.

Sia $A := \{a > 0 \mid a \in H\}$. Come abbiamo appena osservato, A non è vuoto, e dall'osservazione precedente il teorema segue che A possiede un minimo che chiamiamo d.

Dimostriamo che $H = \mathbb{Z}d$.

In primo luogo $d \in H$, e quindi $d, 2d = d + d, 3d = d + d + d, \ldots, -d, -2d = -(d + d), \cdots \in H$, perché, per ipotesi, H è un sottogruppo di $\mathbb{Z}$. In altre parole $\mathbb{Z}d \subset H$.

Dobbiamo ancora dimostrare che $H \subset \mathbb{Z}d$. Sia $a \in H$. Allora $a = qd + r$ con $0 \leq r < d$, per cui $r = a - qd \in H$. Ma $0 \leq r < d$, quindi per la minimalità di d vediamo che r deve essere uguale a 0. Ciò significa $a = qd \in \mathbb{Z}d$.

L'unicità di d verrà dimostrata nel prossimo articolo (oss. 4).

Primo incontro con gli ideali

Definizione 1: $(G, +)$ sia un gruppo abeliano e A e B due sottoinsiemi di G. Allora poniamo

$$A + B := \{a + b \mid a \in A, b \in B\}.$$

In $G = \mathbb{Z}$ quindi $\mathbb{Z}a + \mathbb{Z}b = \{na + mb \mid n, m \in \mathbb{Z}\}$.

Adesso consideriamo di nuovo solo numeri interi.

Se $d|a$, allora a è un multiplo di d. Questa terminologia familiare dal lingugaggio comune è più profonda di quanto possa sembrare e porta alla teoria degli *ideali*, uno dei concetti più importanti dell'algebra.

Osservazione 2: $a|b \iff \mathbb{Z}b \subset \mathbb{Z}a$.

Infatti a divide b se e solo se ogni multiplo di b è anche un multiplo di a.

Nonostante che si tratti di una quasi ovvia riformulazione del concetto di divisibilità, questa osservazione trasforma un problema aritmetico in un problema insiemistico.

Osservazione 3: $\mathbb{Z}a = \mathbb{Z}b \iff a = \pm b$.

Dimostrazione: $\mathbb{Z}a = \mathbb{Z}b$ per l'oss. 2 implica che a e b si dividono a vicenda, per cui $a = \pm b$ (esercizio 28).

Osservazione 4: Dall'osservazione 3 otteniamo anche l'unicità di d nel teorema che caratterizza i sottogruppi di $\mathbb{Z}$. Perche?

Osservazione 5: Dati a e b, esiste un unico numero naturale tale che $\mathbb{Z}a + \mathbb{Z}b = \mathbb{Z}d$.

Dimostrazione: $\mathbb{Z}a + \mathbb{Z}b$ è un sottogruppo di $\mathbb{Z}$ (esercizio 30). L'enunciato segue dal teorema 1.

Teorema 2: Dati a e b, sia d l'unico numero naturale per il quale $\mathbb{Z}a + \mathbb{Z}b = \mathbb{Z}d$. Allora d è il massimo comune divisore di a e b. Infatti d ha le seguenti proprietà:

(1) $d|a$ e $d|b$.
(2) Se f è un altro divisore comune comune di a e b (cioè se $f|a$ e $f|b$), allora $f|d$.

Dimostrazione: (1) $\mathbb{Z}a \subset \mathbb{Z}a + \mathbb{Z}b$, per cui dall'oss. 2 segue che $d|a$. Nello stesso modo si vede che $d|b$.

(2) Ancora per l'oss. 2 abbiamo $\mathbb{Z}a \subset \mathbb{Z}f$ e $\mathbb{Z}b \subset \mathbb{Z}f$, per cui $\mathbb{Z}d = \mathbb{Z}a + \mathbb{Z}b \subset \mathbb{Z}f + \mathbb{Z}f = \mathbb{Z}f$, e quindi $f|d$.

Siccome nel punto (2) necessariamente $f \leq d$ (esercizio 29), vediamo che d è veramente il massimo comune divisore di a e b.

Teorema 3: Sia $d = \mathrm{mcd}(a, b)$. Allora esistono numeri $n, m \in \mathbb{Z}$ tali che $d = na + mb$.

Dimostrazione: Ciò è una conseguenza immediata del teorema 2.

Definizione 2: Due numeri interi a e b si chiamano *relativamente primi* se $\mathrm{mcd}(a, b) = 1$. È chiaro che ciò accade se e solo se a e b non hanno comuni divisori tranne 1 e -1.

Teorema 4: $\mathrm{mcd}(a, b) = 1 \iff$ esistono $n, m \in \mathbb{Z}$ tali che $na + mb = 1$.

Dimostrazione: $\Longrightarrow$: Segue dal teorema 3.

$\Longleftarrow$: Sia $d := \mathrm{mcd}(a, b)$. Se $na + mb = 1$, allora $1 \in \mathbb{Z}a + \mathbb{Z}b = \mathbb{Z}d$, per cui $d|1$ e quindi $d = \pm 1$! Ma $d \geq 0$, quindi $d = 1$.

La moltiplicazione russa

Esiste un algoritmo leggendario del contadino russo per la moltiplicazione di due numeri, uno dei quali deve essere un intero positivo. Assumiamo che vogliamo calcolare $86x$, dove x è un numero reale (nello stesso modo abbiamo calcolato $10x$ a pag. 14):

$$86 \cdot x \longrightarrow 43 \cdot 2x \xrightarrow{\nearrow 2x} 42 \cdot 2x \longrightarrow 21 \cdot 4x \xrightarrow{\nearrow 4x} 20 \cdot 4x \longrightarrow$$
$$10 \cdot 8x \longrightarrow 5 \cdot 16x \xrightarrow{\nearrow 16x} 4 \cdot 16x \longrightarrow 2 \cdot 32x \longrightarrow$$
$$1 \cdot 64x \xrightarrow{\nearrow 64x} \longrightarrow 0 \cdot 128x \longrightarrow \bullet$$

Lo schema va interpretato così: p sarà il risultato della moltiplicazione; all'inizio poniamo $p = 0$. $86x = 43 \cdot 2x$, quindi possiamo sostituire x con $2x$ e dimezzare il primo fattore che così diventa dispari. Con il nuovo x abbiamo $43x = x + 42x$. Aggiungiamo x a p e procediamo con $42x = 21 \cdot 2x$ come nel primo passo: sostituiamo quindi x con $2x$ e dimezziamo il primo fattore. E così via: Quando arriviamo a un primo fattore dispari, sommiamo l'ultimo valore di x a p e diminuiamo il primo fattore di uno che così diventa pari. Quando il primo fattore è pari, sostituiamo x con $2x$ e dimezziamo il primo fattore. Ci fermiamo quando il primo fattore è uguale a 0.

Altrettanto semplice e importante è la formulazione puramente matematico-ricorsiva dell'algoritmo - scriviamo f per la funzione definita da $f(n, x) = nx$:

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

Diamo una versione ricorsiva e una versione iterativa in Perl per questo algoritmo:

```
sub mrussa # $p=mrussa($n,$x)
  {my ($n,$x)=@_;
  return $x+mrussa($n-1,$x) if $n%2;
  return mrussa($n/2,$x+$x) if $n>0; 0}
sub mrussa # $p=mrussa($n,$x)
  {my ($n,$x)=@_; my $p;
  for ($p=0;$n;) {if ($n%2) {$p+=$x; $n--}
  else {$x+=$x; $n/=2}} $p}
```

Come trovare la rappresentazione binaria

Non è difficile convincersi che ogni numero naturale $n > 0$ possiede una rappresentazione binaria, cioè della forma

$$n = a_k 2^k + a_{k-1} 2^{k-1} + \cdots + a_2 2^2 + a_1 2 + a_0 \qquad (*)$$

con coefficienti (o cifre) $a_i \in \{0, 1\}$ e $a_k = 1$ univocamente determinati. Sia $\text{rapp2}(n) = (a_k, \ldots, a_0)$ la lista i cui elementi sono queste cifre. Dalla rappresentazione (*) si deduce la seguente relazione ricorsiva, in cui utilizziamo il meccanismo della fusione di liste del Perl (cfr. pag. 19):

$$\text{rapp2}(n) = \begin{cases} (1) & \text{se} \quad n = 1 \\ (\text{rapp2}(\frac{n}{2}), 0) & \text{se} \quad n \text{ è pari} \\ (\text{rapp2}(\frac{n-1}{2}), 1) & \text{se} \quad n \text{ è dispari} \end{cases}$$

Questa relazione può essere tradotta immediatamente in un programma in Perl:

```
sub rapp2 # @cifre=rapp2($n)
  {my $n=shift; return (1) if $n==1;
  return (rapp2($n/2),0) if $n%2==0;
  (rapp2(($n-1)/2),1)}
```

La potenza russa

Per il calcolo di potenze con esponenti reali arbitrari si può usare la funzione **pow** del Perl: la terza radice si ottiene ad esempio con *pow(x,1/3)*. Come molte altre funzioni matematiche (ad esempio *sin* e *cos*) anche *pow* richiede l'inclusione *use POSIX;* all'inizio del file.

Per esponenti interi positivi si può usare invece un altro metodo del contadino russo. Assumiamo di voler elevare x alla 937-esima potenza.

$$x^{937} \xrightarrow{\nearrow x} x^{936} \longrightarrow (x^2)^{468} \longrightarrow (x^4)^{234} \longrightarrow (x^8)^{117} \xrightarrow{\nearrow x^8}$$
$$(x^8)^{116} \longrightarrow (x^{16})^{58} \longrightarrow (x^{32})^{29} \xrightarrow{\nearrow x^{32}} (x^{32})^{28} \longrightarrow$$
$$(x^{64})^{14} \longrightarrow (x^{128})^{7} \xrightarrow{\nearrow x^{128}} (x^{128})^{6} \longrightarrow (x^{256})^{3} \xrightarrow{\nearrow x^{256}}$$
$$(x^{256})^{2} \longrightarrow (x^{512})^{1} \xrightarrow{\nearrow x^{512}} (x^{512})^{0} \longrightarrow \bullet$$

Lo schema va interpretato così: $x^{937} = x \cdot x^{936}$. Ci ricordiamo il fattore x. $x^{936} = (x^2)^{468}$, quindi sostituendo x con x^2 dobbiamo solo fare la 468-esima potenza del nuovo x oppure la 234-esima se sostituiamo ancora x con il suo quadrato. Quando arriviamo a un esponente dispari, moltiplichiamo l'ultimo valore di x con il fattore fino a quel punto memorizzato e possiamo quindi diminuire l'esponente di uno, ottenendo di nuovo un esponente pari. E così via. In ogni passaggio dobbiamo solo o formare un quadrato e dimezzare l'esponente oppure moltiplicare il fattore con il valore attuale di x e diminuire l'esponente di uno.

Anche in questo caso invece del ragionamente iterativo si può riformulare il problema in modo matematico-ricorsivo - stavolta $f(x, n) = x^n$:

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

È facile tradurre queste idee in un programma ricorsivo o iterativo in Perl:

```
sub potenza # $p=potenza($x,$n)
  {my ($x,$n)=@_;
  return $x*potenza($x,$n-1) if $n%2;
  return potenza($x*$x,$n/2) if $n>0; 1}
sub potenza # $p=potenza($x,$n)
  {my ($x,$n)=@_; my $p;
  for ($p=1;$n;) {if ($n%2) {$p*=$x; $n--}
  else {$x*=$x; $n/=2}} $p}
```

Tirocini all'ARDSU

L'ufficio *Orientamento al Lavoro* dell'ARDSU di Ferrara è un buon punto di partenza per chi cerca contatti con il mondo del lavoro. Proprio in questi mesi ha avuto, da buone aziende a Ferrara o nella provincia di Ferrara, molte offerte di tesi di laurea e tirocini (per informatici e matematici) con possibilità di assunzioni in seguito.

L'ufficio inoltre organizza da anni seminari (ad esempio su *tecniche e strumenti per la ricerca attiva del lavoro*), corsi di informatica di base, corsi di lingua inglese e diversi progetti formativi. L'ARDSU collabora strettamente con l'università e con gli enti pubblici e ha contatti con molte aziende.

L'ufficio si trova a Ferrara in via Cairoli 32 (cortile interno). Telefonare prima alla signora Ornella Gandini, tel. 299812.

Lo schema di Horner

Sia dato un polinomio $f = a_0x^n + a_1x^{n-1} + \cdots + a_n \in A[x]$, dove A è un qualsiasi anello commutativo. Per $\alpha \in A$ vogliamo calcolare $f(\alpha)$.

Sia ad esempio $f = 3x^4 + 5x^3 + 6x^2 + 8x + 17$. Poniamo

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

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

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

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

$$g := b_0x^{n-1} + b_1x^{n-2} + \cdots + b_{n-1}.$$

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

$$\begin{array}{rcl} \alpha g = & \alpha b_0x^{n-1} + \alpha b_1x^{n-2} + \cdots + \alpha b_{n-1} = \\ = & (b_1 - a_1)x^{n-1} + (b_2 - a_2)x^{n-2} + \cdots + \\ & +(b_{n-1} - a_{n-1})x + b_n - a_n = \\ = & (b_1x^{n-1} + b_2x^{n-2} + \cdots + b_{n-1}x + b_n) - \\ & -(a_1x^{n-1} + a_2x^{n-2} + \ldots a_{n-1}x + a_n) = \\ = & x(g - b_0x^{n-1}) + b_n - (f - a_0x^n) = \\ = & xg - b_0x^n + b_n - f + a_0x^n = xg + b_n - f \end{array}$$

quindi

$$f = (x - \alpha)g + b_n,$$

e ciò implica

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

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

Questo algoritmo si chiama *schema di Horner* o *schema di Ruffini* ed è molto più veloce del calcolo separato delle potenze di α (tranne nel caso che il polinomio consista di una sola o di pochissime potenze, in cui si userà invece l'algoritmo del contadino russo). Quando serve solo il valore $f(\alpha)$, in un programma in Perl si può usare la stessa variabile per tutti i b_k:

```
sub horner # y=horner($x,@a)
  {my ($x,@a)=@_; my $b=shift @a;
  for (@a) {$b=$b*$x+$_} $b}
```

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

$$(1,0,0,1,1,0,1,1,1)_2 = \text{horner}(2,1,0,0,1,1,0,1,1,1)$$

$$(A,F,7,3,0,5,E)_{16} = \text{horner}(16,10,15,7,3,0,5,14).$$

Zeri di una funzione continua

Siano $a < b$ e $f : [a,b] \longrightarrow \mathbb{R}$ una funzione continua tale che $f(a) < 0$ e $f(b) > 0$. In analisi si impara che allora la funzione f deve contenere uno zero nell'intervallo (a,b). Da questo fatto deriva un buon metodo elementare e facile da ricordare per la ricerca delle radici di una funzione continua, che in un pseudolinguaggio che prevede procedure ricorsive può essere formulato nel modo seguente:

```
if b − a < ε then return (a, b)
x = (a + b)/2
if f(x) == 0 then return (x, x)
if f(x) > 0 then cerca in (a, x) # ricorsione
else cerca in (x, b) # ricorsione
```

$\varepsilon > 0$ è qui la precisione richiesta nell'approssimazione al valore x della radice; cioè ci fermiamo quando abbiamo trovato un intervallo di lunghezza $< \varepsilon$ al cui interno si deve trovare uno zero della funzione. È chiaro che questo algoritmo teoricamente deve terminare. In pratica però potrebbe non essere così. Infatti, se ε è minore della precisione della macchina, a un certo punto si avrà che il valore effettivamente calcolato come approssimazione di $x = \frac{a+b}{2}$ è uguale a b, e quindi, se al passo (4) dobbiamo sostituire b con x, rimaniamo sempre nella situazione (a,b) e avremo un ciclo infinito.

Assumiamo ad esempio che $a = 3.18$ e $b = 3.19$ e che la macchina arrotondi a due cifre decimali. Allora $a + b = 6.37$ e $x = \frac{a+b}{2} = 3.185$ che viene arrotondato a $3.19 = b$. Se noi avessimo impostato $\varepsilon = 0.001$, il programma possibilmente non termina.

Si può però sfruttare questo fenomeno a nostro favore: l'imprecisione della macchina fa in modo che prima o poi arriviamo a $x = a$ o $x = b$, e in quel momento ci fermiamo, potendo così applicare un criterio di interruzione indipendente dalla macchina.

In Perl possiamo formulare l'algoritmo nel modo seguente - bisogna prima verificare che veramente $f(a) < 0$ e $f(b) > 0$ e sostituire f con $-f$ quando accade il contrario:

```
sub zero # ($a,$b)=zero($f,$a,$b)
  {my ($f,$a,$b)=@_;
  my $x=($a+$b)/2;
  return ($a,$b) if $x==$a or $x==$b;
  my $fx=&$f($x); return ($x,$x) if $fx==0;
  return zero($f,$a,$x) if $fx>0;
  zero($f,$x,$b)}
```

$f(x)$ deve essere calcolato due volte, per questa ragione nella terzultima riga del programma abbiamo introdotto la variabile ausiliaria *$fx* a cui viene assegnato il valore $f(x)$.

Dobbiamo spiegare qui la sintassi usata nel Perl per argomenti che sono funzioni. L'argomento *$f* che rappresenta la funzione è uno scalare (cfr. pag. 18) di un tipo particolare: un *puntatore* (o *riferimento*) alla funzione. Per chiamare la funzione a cui punta questo riferimento bisogna utilizzare *&$f*. Se abbiamo definito una funzione *f* mediante *sub f*, il puntatore ad essa è *\&f*.

Vediamo queste nozioni nel seguente esempio in cui calcoliamo una (delle al massimo quattro) radici del polinomio $f = x^4 + x - 7$. Verifichiamo prima che $f(0) = -7 < 0$ e $f(2) = 16 + 2 - 7 = 11 > 0$. Adesso possiamo scrivere

```
($a,$b)=zero(\&f,0,2);
print "$a <= x <= $b\n";
# output: 1.52935936477247 <= x <= 1.52935936477247
```

ALGORITMI E STRUTTURE DI DATI

Corso di laurea in matematica | Anno accademico 2001/2002 | Numero 6 ◇ 29 Ottobre 2001

Un po' di algebra esterna

V sia uno spazio vettoriale (su un corpo K qualsiasi che in seguito sarà uguale ad $\mathbb{R}$, ma non ha importanza) e $v_1, \dots, v_m \in V$ (con $m \geq 0$). Usiamo la seguente abbreviazione:

$v_1 \wedge \cdots \wedge v_m = 0$ *se* $v_1, \dots, v_m$ *sono linearmente dipendenti (su K).*

$v_1 \wedge \cdots \wedge v_m \neq 0$ *se* $v_1, \dots, v_m$ *sono linearmente indipendenti (su K).*

*Questa notazione appartiene all'**algebra esterna**, un metodo molto potente che permette di amministrare in modo algebrico e trasparente tutte le formule che riguardano determinanti, sottodeterminanti, prodotto vettoriale ($v \times w$), dipendenza lineare ecc.*

***Teorema 1:** Siano $u = (u_1, \dots, u_n)$ e $v = (v_1, \dots, v_n)$ due vettori di $\mathbb{R}^n$. Allora i seguenti enunciati sono equivalenti:*

(1) $u \wedge v = 0$.

(2) $\begin{vmatrix} u_i & v_i \\ u_j & v_j \end{vmatrix} = 0$ *per ogni* i, j *con* $1 \leq i, j \leq n$.

(3) $\begin{vmatrix} u_i & v_i \\ u_j & v_j \end{vmatrix} = 0$ *per ogni* i, j *con* $1 \leq i < j \leq n$.

Dimostrazione: È chiaro che (2) e (3) sono equivalenti, perché per $i = j$ questi determinanti sono in ogni caso nulli, mentre, se scambiamo i e j, cambiano solo di segno.

(1) $\Longrightarrow$ (2): Se u e v sono linearmente dipendenti, allora uno dei due, ad esempio v, è un multiplo dell'altro: $v = \lambda u$ con $\lambda \in \mathbb{R}$. Allora

$$\begin{vmatrix} u_i & v_i \\ u_j & v_j \end{vmatrix} = \begin{vmatrix} u_i & \lambda u_i \\ u_j & \lambda u_j \end{vmatrix} = \lambda u_i u_j - \lambda u_i u_j = 0.$$

(2) $\Longrightarrow$ (1): Se $u = 0$, tutti gli enunciati sono banalmente veri. Sia quindi $u \neq 0$, ad esempio $u_1 \neq 0$. Per ipotesi, per ogni j vale

$$\begin{vmatrix} u_1 & v_1 \\ u_j & v_j \end{vmatrix} = 0$$

cioè $u_1 v_j - u_j v_1 = 0$ e quindi $v_j = \dfrac{v_1}{u_1} u_j$ per ogni j.

Cio significa che $v = \lambda u$ con $\lambda = \dfrac{v_1}{u_1}$.

***Teorema 2:** Siano $u = (u_1, \dots, u_n)$, $v = (v_1, \dots, v_n)$ e $w = (w_1, \dots, w_n)$ tre vettori di $\mathbb{R}^n$. Allora i seguenti enunciati sono equivalenti:*

(1) $u \wedge v \wedge w = 0$.

(2) $\begin{vmatrix} u_i & v_i & w_i \\ u_j & v_j & w_j \\ u_k & v_k & w_k \end{vmatrix} = 0$ *per ogni* i, j, k *con* $1 \leq i, j, k \leq n$.

(3) $\begin{vmatrix} u_i & v_i & w_i \\ u_j & v_j & w_j \\ u_k & v_k & w_k \end{vmatrix} = 0$ *per ogni* i, j, k *con* $1 \leq i < j < k \leq n$.

Dimostrazione: Pagina 29.

***Definizione:** Siano $v = (v_1, v_2, v_3)$, $w = (w_1, w_2, w_3)$ due vettori di $\mathbb{R}^3$. Allora il vettore*

$$v \times w := \left(\begin{vmatrix} v_2 & w_2 \\ v_3 & w_3 \end{vmatrix}, -\begin{vmatrix} v_1 & w_1 \\ v_3 & w_3 \end{vmatrix}, \begin{vmatrix} v_1 & w_1 \\ v_2 & w_2 \end{vmatrix} \right)$$

*si chiama il **prodotto vettoriale** dei vettori v e w. Dal teorema 1 vediamo che v e w sono linearmente dipendenti se e solo se $v \times w = 0$.*

Questa settimana

27 Un po' di algebra esterna
Intersezione di rette nel piano

28 Determinanti
Multilinearità del determinante

29 Dimostrazione del teorema 27/2
Due casi speciali
Proiezione di un punto su una retta
Punto e retta nel piano

Intersezione di rette nel piano

Usiamo le abbreviazioni, per $p, v \in \mathbb{R}^n$,

$$\mathbb{R}v := \{tv \mid t \in \mathbb{R}\}$$

$$p + \mathbb{R}v := \{p + tv \mid t \in \mathbb{R}\}$$

Gli insiemi della prima forma per cui $v \neq 0$ sono quindi esattamente le rette in $\mathbb{R}^n$ passanti per l'origine, mentre ogni retta può essere scritta nella seconda forma con $v \neq 0$. Per p si può usare un punto qualsiasi della retta.

Due rette $R = p + \mathbb{R}v$ ed $S = q + \mathbb{R}w$ sono parallele se e solo se i vettori v e w sono paralleli, cioè se e solo se esiste $\lambda \in \mathbb{R}$ (necessariamente $\neq 0$, perché sia v che w devono essere $\neq 0$) tale che $w = \lambda v$, cioè se e solo se $v \wedge w = 0$. In questo caso le due rette coincidono se e solo se $q \in R$, cioè se e solo se $q - p$ è un multiplo di v e allora $R = S = R \cap S$. Altrimenti $R \cap S = \emptyset$.

Se invece le rette non sono parallele, i vettori $v = (v_1, v_2)$ e $w = (w_1, w_2)$ sono linearmente indipendenti e per trovare l'intersezione $R \cap S$ consideriamo l'equazione vettoriale $p + tv = q + sw$ nelle incognite t ed s, costituita da due equazioni scalari

$$\begin{aligned} v_1 t - w_1 s &= q_1 - p_1 \\ v_2 t - w_2 s &= q_2 - p_2 \end{aligned}$$

il cui determinante $\begin{vmatrix} v_1 & w_1 \\ v_2 & w_2 \end{vmatrix}$ è diverso da zero (teorema 1 o prop. 29/1) e che quindi possiede esattamente una soluzione (t, s) da cui otteniamo l'unico punto $p + tv$ dell'intersezione delle due rette.

Quindi due rette non parallele nel piano $\mathbb{R}^2$ si intersecano esattamente in un punto.

L'intersezione di due rette date tramite equazioni $a_1 x_1 + a_2 x_2 = c$ e $b_1 x_1 + b_2 x_2 = d$ può essere calcolata anche risolvendo direttamente il sistema lineare

$$\begin{aligned} a_1 x_1 + a_2 x_2 &= c \\ b_1 x_1 + b_2 x_2 &= d \end{aligned}$$

nelle incognite x_1 e x_2.

Determinanti

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

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 di cui avremo bisogno in seguito. Tutte queste regole valgono anche per determinanti $n \times n$, se riformulate in modo naturale.

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

Dimostrazione: Immediata.

Lemma 2: 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 3: Se si scambiano due righe o due colonne in una matrice 3×3, il determinante si moltiplica per -1.

Dimostrazione: Ciò, per il lemma 1, è evidente per lo scambio della seconda e della terza colonna e, per il lemma 2, 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 4: 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.

Multilinearità del determinante

Siano dati n vettori

$$a_1 = (a_1^1, \dots, a_1^n), \dots, a_n = (a_n^1, \dots, a_n^n)$$

di $\mathbb{R}^n$. Allora la matrice

$$A := \begin{pmatrix} a_1^1 & \dots & a_n^1 \\ & \dots & \\ a_1^n & \dots & a_n^n \end{pmatrix}$$

è quadratica e possiamo formare il suo determinante

$$|A| := \begin{vmatrix} a_1^1 & \dots & a_n^1 \\ & \dots & \\ a_1^n & \dots & a_n^n \end{vmatrix}$$

che denotiamo anche con $\det(a_1, \dots, a_n)$.

Si noti che nella matrice abbiamo, come d'uso nel calcolo tensoriale (un calcolo multilineare sistematico molto importante nella geometria differenziale e nella fisica matematica), scritto gli indici di riga in alto, così come spesso, e con buone ragioni, anche al di fuori di una matrice gli indici dei componenti di un vettore v di $\mathbb{R}^n$ vengono posti in alto: $v = (v^1, \dots, v^n)$. È quasi sempre chiaro dal contesto se si tratta di indici o di esponenti di potenze.

Proposizione 1: Il determinante è una funzione *multilineare* delle colonne (e delle righe) di una matrice.

Ciò significa che, se anche $b = (b^1, \dots, b^n)$ è un vettore di $\mathbb{R}^n$, allora, per $\lambda, \mu \in \mathbb{R}$,

$$\det(\lambda a_1 + \mu b, a_2, \dots, a_n) = \\ = \lambda \det(a_1, a_2, \dots, a_n) + \mu \det(b, a_2, \dots, a_n)$$

$$\det(a_1, \dots, a_{n-1}, \lambda a_n + \mu b) = \\ = \lambda \det(a_1, \dots, a_{n-1}, a_n) + \mu \det(a_1, \dots, a_{n-1}, b)$$

Dimostrazione: Ciò, per $n = 2$, è evidente (verificare da soli) e, per l'ultima colonna, segue poi per induzione, come si vede dalle formule di espansione date. Dai lemmi di scambio (che valgono per ogni n, anche se li abbiamo dimostrato solo per $n = 2, 3$) ciò implica che il determinante è lineare in ogni colonna (e quindi anche in ogni riga, perché, come si vedrà nel corso di Geometria, il determinante della matrice trasposta di A, cioè della matrice che cui colonne sono le righe di A, è uguale al determinante di A).

Proposizione 2: Se in un determinante a una colonna (o riga) aggiungiamo un multiplo di una delle altre colonne (o righe), il determinante non cambia:

$$\det(\dots, a_i + \lambda a_k, \dots) = \det(\dots, a_i, \dots)$$

per $i \neq k$.

Dimostrazione: Assumiamo $i < k$. Allora

$$\det(\dots, a_i + \lambda a_k, \dots, a_k, \dots) = \\ = \det(\dots, a_i, \dots, a_k, \dots) + \\ + \lambda \det(\dots, a_k, \dots, a_k, \dots) = \\ = \det(\dots, a_i, \dots, a_k, \dots)$$

usando la multilinearità e il lemma 4.

Dimostrazione del teorema 27/2

Anche questa volta è chiaro che (2) e (3) sono equivalenti, perché se due dei tre indici coincidono, il corrispondente determinante si annulla per il lemma 28/4, e se invece gli indici sono tutti distinti, li possiamo scambiare per ottenere $i < j < k$ con il determinante che cambia solo di segno.

(1) $\Longrightarrow$ (2): Sia $u \wedge v \wedge w = 0$, ad esempio $w = \lambda u + \mu v$ con $\lambda, \mu \in \mathbb{R}$. Allora

$$\begin{vmatrix} u_i & v_i & w_i \\ u_j & v_j & w_j \\ u_k & v_k & w_k \end{vmatrix} = \begin{vmatrix} u_i & v_i & \lambda u_i + \mu v_i \\ u_j & v_j & \lambda u_j + \mu v_j \\ u_k & v_k & \lambda u_k + \mu v_k \end{vmatrix} =$$

$$= \lambda \begin{vmatrix} u_i & v_i & u_i \\ u_j & v_j & u_j \\ u_k & v_k & u_k \end{vmatrix} + \mu \begin{vmatrix} u_i & v_i & v_i \\ u_j & v_j & v_j \\ u_k & v_k & v_k \end{vmatrix} = 0$$

come segue dal lemma 28/4.

(2) $\Longrightarrow$ (1): Tutti i determinanti al punto (2) siano nulli. Dobbiamo dimostrare che $u \wedge v \wedge w = 0$. Ciò è sicuramente vero se $u \wedge v = 0$. Possiamo quindi assumere che $u \wedge v \neq 0$. Dal teorema 27/1 segue che allora ad esempio $\begin{vmatrix} u_1 & v_1 \\ u_2 & v_2 \end{vmatrix} \neq 0$, mentre per ipotesi per ogni k con $1 \leq k \leq n$ abbiamo

$$\begin{vmatrix} u_k & v_k & w_k \\ u_1 & v_1 & w_1 \\ u_2 & v_2 & w_2 \end{vmatrix} = 0.$$

Sviluppando questo determinante troviamo

$$u_k \begin{vmatrix} v_1 & w_1 \\ v_2 & w_2 \end{vmatrix} - v_k \begin{vmatrix} u_1 & w_1 \\ u_2 & w_2 \end{vmatrix} + w_k \begin{vmatrix} u_1 & v_1 \\ u_2 & v_2 \end{vmatrix} = 0$$

e ciò, valendo per ogni indice k, significa che

$$\begin{vmatrix} v_1 & w_1 \\ v_2 & w_2 \end{vmatrix} u - \begin{vmatrix} u_1 & w_1 \\ u_2 & w_2 \end{vmatrix} v + \begin{vmatrix} u_1 & v_1 \\ u_2 & v_2 \end{vmatrix} w = 0$$

una relazione lineare tra i vettori u, v e w in cui almeno il terzo coefficiente è diverso da zero. I tre vettori sono quindi linearmente dipendenti.

Due casi speciali

Dai teoremi 27/1 e 27/2 otteniamo come casi particolari le seguenti proposizioni.

Proposizione 1: Siano $u = (u_1, u_2)$ e $v = (v_1, v_2)$ due vettori del piano $\mathbb{R}^2$. Allora i seguenti enunciati sono equivalenti:

(1) $u \wedge v = 0$.

(2) $\begin{vmatrix} u_1 & v_1 \\ u_2 & v_2 \end{vmatrix} = 0$

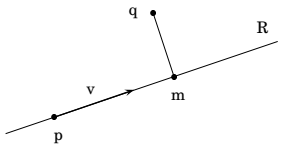
Proposizione 2: Siano $u = (u_1, u_2, u_3)$, $v = (v_1, v_2, v_3)$ e $w = (w_1, w_2, w_3)$ tre vettori di $\mathbb{R}^3$. Allora i seguenti enunciati sono equivalenti:

(1) $u \wedge v \wedge w = 0$.

(2) $\begin{vmatrix} u_1 & v_1 & w_1 \\ u_2 & v_2 & w_2 \\ u_3 & v_3 & w_3 \end{vmatrix} = 0.$

Proiezione di un punto su una retta

Siano dati una retta $R = p + \mathbb{R}v$ (con $v \neq 0$) e un punto q di $\mathbb{R}^n$. Vogliamo calcolare la proiezione ortogonale m di q su R. Il punto m deve essere in primo luogo un punto della retta e quindi della forma $m = p + tv$, inoltre il vettore $q - m$ deve essere ortogonale a v, cioè $(q - m, v) = 0$, ossia $(q, v) = (m, v)$. Ciò implica



$$(q, v) = (m, v) = (p + tv, v) = (p, v) + t(v, v) = (p, v) + t|v|^2,$$

ossia

$$t = \frac{(q, v) - (p, v)}{|v|^2} = \frac{(q - p, v)}{|v|^2}.$$

L'unica soluzione è quindi

$$m = p + \frac{(q - p, v)}{|v|^2} v.$$

La distanza di q dalla retta è uguale alla lunghezza del vettore $m - q$.

Punto e retta nel piano

Nel piano le formule per la proiezione di un punto su una retta possono essere formulate in maniera più esplicita. Come a pagina 11 siano $v = (v_1, v_2)$ e $a = (a_1, a_2)$ con $a_1 = -v_2, a_2 = v_1$. La retta può essere rappresentata in forma parametrica come $R = p + \mathbb{R}v$ oppure tramite l'equazione

$$a_1 x_1 + a_2 x_2 = c \text{ oppure } (a, x) = (a, p)$$

con $c = a_1 p_1 + a_2 p_2 = (a, p)$. I vettori a e v sono ortogonali tra di loro e hanno la stessa lunghezza. Possiamo calcolare

$$m = p + \frac{(q - p, v)}{|v|^2} v$$

come nel caso generale di $\mathbb{R}^n$; nel piano vediamo però che m deve essere anche della forma

$$m = q + sa$$

per qualche $s \in \mathbb{R}$ il cui valore può essere trovato utilizzando l'equazione $(a, m) = c$ che m come punto della retta deve soddisfare. Quindi

$$c = (a, m) = (a, q) + s(a, a) = (a, q) + s|a|^2,$$

per cui

$$s = \frac{c - (a, q)}{|a|^2} = \frac{(a, p) - (a, q)}{|a|^2} = \frac{(a, p - q)}{|a|^2}$$

e quindi

$$m = q + \frac{c - (a, q)}{|a|^2} a$$

Adesso otteniamo facilmente la distanza di q dalla retta, infatti questa distanza coincide con la lunghezza del vettore $q - m$:

$$|q - m| = \frac{|(a, q) - c|}{|a|} = \frac{|a_1 q_1 + a_2 q_2 - c|}{\sqrt{a_1^2 + a_2^2}}$$

ALGORITMI E STRUTTURE DI DATI

Corso di laurea in matematica | Anno accademico 2001/2002 | Numero 7 ◇ 5 Novembre 2001

La retta in uno spazio vettoriale

Una retta R in uno spazio vettoriale reale V possiede una rappresentazione parametrica

$$R = p + \mathbb{R}v$$

con $p, v \in V$ e $v \neq 0$. È chiaro che, sostituendo v con un vettore w, si ottiene la stessa retta se e solo se $w = \lambda v$ con $\lambda \in \mathbb{R} \setminus 0$.

Sostituendo p con un punto q, si ottiene la stessa retta se e solo se $q - p = tv$ con $t \in \mathbb{R}$. In altre parole, ogni punto $q \in R$ può essere usato al posto di p.

Le rette $R = p + \mathbb{R}v$ ed $S = q + \mathbb{R}w$ sono parallele se e solo se $w = \lambda v$ con $\lambda \in \mathbb{R} \setminus 0$. In questo caso esse coincidono se e solo se $q \in R$, cioè se e solo se $q = p + tv$ per qualche t, cioè se e solo se $q - p$ è un multiplo di v, come già osservato. In questo caso $R = S = R \cap S$. Altrimenti $R \cap S = \emptyset$ perché $p + tv = q + s\lambda v$ implica $q = p + (t - s\lambda)v \in R$.

Due rette in $\mathbb{R}^3$

Lemma: Siano dati tre vettori a, b, c in uno spazio vettoriale qualunque. Allora:

(1) Se c è combinazione lineare di a e b, allora $a \wedge b \wedge c = 0$.
(2) Se $a \wedge b \neq 0$ (cioè se a e b sono linearmente indipendenti) e $a \wedge b \wedge c = 0$, allora c è combinazione lineare di a e b.

Dimostrazione: (1) Chiaro.

(2) $a \wedge b \wedge c = 0$ implica che esiste una relazione $\lambda a + \mu b + \nu c = 0$ con coefficienti $\lambda, \mu, \nu \in \mathbb{R}$ non tutti nulli. Se $\nu = 0$, allora $(\lambda, \mu) \neq (0, 0)$ e $\lambda a + \mu b = 0$ (perché in quel caso anche $\nu c = 0$), in contraddizione all'indipendenza lineare di a e b. Quindi $\nu \neq 0$. Allora però possiamo scrivere $c = -\frac{\lambda}{\nu}a - \frac{\mu}{\nu}b$ e vediamo che c è una combinazione lineare di a e b.

Siano date due rette $R = p + \mathbb{R}v$ ed $S = q + \mathbb{R}w$ (con $v, w \neq 0$) nello spazio $\mathbb{R}^3$, non necessariamente parallele.

Per trovare l'intersezione $R \cap S$ dobbiamo risolvere l'equazione vettoriale $p + tv = q + sw$ nelle incognite scalari t ed s, equivalente all'equazione $tv - sw = q - p$, dalla quale si vede che $R \cap S \neq \emptyset$ se e solo se il vettore $q - p$ è combinazione lineare di v e w.

(1) v e w siano linearmente dipendenti: In questo caso $q - p$ è combinazione linerare di v e w se e solo se è un multiplo di v, e siccome $p \in R$, ciò accade se e solo $q \in R$.

Quindi $R \cap S = R = S$ oppure $R \cap S = \emptyset$ come già visto prima.

(2) v e w siano linearmente indipendenti: In questo caso, per il lemma, $q - p$ è combinazione lineare di v e w se e solo se $v \wedge w \wedge (q - p) = 0$ e quindi, per la prop. 29/2, se e solo se

$$\begin{vmatrix} v_1 & w_1 & q_1 - p_1 \\ v_2 & w_2 & q_2 - p_2 \\ v_3 & w_3 & q_3 - p_3 \end{vmatrix} = 0$$

(sempre in questa ipotesi però, che v e w siano linearmente indipendenti).

Assumiamo che ciò accada e che $p + tv = q + sw$ sia un punto dell'intersezione $R \cap S$. Possono essercene altri?

Per una coppia di numeri reali t', s' sono allora equivalenti:

$$\begin{aligned} p + t'v &= q + s'w \\ q + sw - tv + t'v &= q + s'w \\ (t' - t)v &= (s' - s)w \end{aligned}$$

Ma v e w sono linearmente indipendenti, quindi $(t' - t)v = (s' - s)w$ implica $t' = t$ e $s' = s$. Quindi le due rette si intersecano in un solo punto. Il seguente teorema riassume ciò che abbiamo finora dimostrato.

Teorema: In $\mathbb{R}^3$ siano date due rette $R = p + \mathbb{R}v$ ed $S = q + \mathbb{R}w$.

(1) Se i due vettori v e w sono linearmente indipendenti, allora le rette R e S si intersecano se e solo se

$$\begin{vmatrix} v_1 & w_1 & q_1 - p_1 \\ v_2 & w_2 & q_2 - p_2 \\ v_3 & w_3 & q_3 - p_3 \end{vmatrix} = 0$$

e in questo caso si intersecano in un solo punto.

(2) Se i due vettori v e w sono linearmente dipendenti, allora le due rette sono parallele e si intersecano se e solo se $q - p$ è un multiplo di v e in quest'ultimo caso le due rette coincidono.

Questa settimana

30 La retta in uno spazio vettoriale
Due rette in $\mathbb{R}^3$
Il prodotto vettoriale

31 Identità di Graßmann e di Jacobi
Area di un parallelogramma
Significato geometrico di $v \times w$

32 Piani nello spazio
Proiezione di un punto su un piano

33 Il piano passante per tre punti
Il volume
Orientamento

Il prodotto vettoriale

Abbiamo definito a pagina 27 il prodotto vettoriale $v \times w$ di due vettori $v = (v_1, v_2, v_3)$ e $w = (w_1, w_2, w_3)$ di $\mathbb{R}^3$. Esso permette di rappresentare il prodotto esterno $v \wedge w$ di due vettori di $\mathbb{R}^3$ come un vettore dello stesso $\mathbb{R}^3$. Ciò non è possibile in altre dimensioni perché il prodotto esterno di due vettori di $\mathbb{R}^n$ è un vettore di uno spazio vettoriale reale di dimensione $\binom{n}{2}$ e solo per $n = 3$ si ha $\binom{n}{2} = n$.

Scrivendo il prodotto vettoriale come vettore colonna e calcolando esplicitamente i coefficienti abbiamo

$$v \times w = \begin{pmatrix} v_2 w_3 - v_3 w_2 \\ v_3 w_1 - v_1 w_3 \\ v_1 w_2 - v_2 w_1 \end{pmatrix}$$

Si noti il modo ciclico in cui si susseguono gli indici.

Proposizione 1: $u = (u_1, u_2, u_3)$ sia un terzo vettore di $\mathbb{R}^3$. Allora

$$\det(u, v, w) = (u, v \times w)$$

dove l'espressione a destra è il prodotto scalare dei vettori u e $v \times w$ che abbiamo definito a pagina 11.

Dimostrazione: Il prodotto scalare è uguale a

$$u_1 \begin{vmatrix} v_2 & w_2 \\ v_3 & w_3 \end{vmatrix} - u_2 \begin{vmatrix} v_1 & w_1 \\ v_3 & w_3 \end{vmatrix} + u_3 \begin{vmatrix} v_1 & w_1 \\ v_2 & w_2 \end{vmatrix}$$

Però questo è proprio il determinante $\det(u, v, w)$ secondo la formula di espansione data a pagina 28.

Proposizione 2: Il vettore $v \times w$ è ortogonale sia a v che a w.

Dimostrazione: Verifichiamo ad es. che $(v, v \times w) = 0$. Per la prop. 1 abbiamo $(v, v \times w) = \det(v, v, w)$. Questo determinante però si annulla, perché contiene due righe uguali (lemma 28/4).

Identità di Graßmann e di Jacobi

Siano $u, v, w \in \mathbb{R}^3$.

Identità di Graßmann:

$$u \times (v \times w) = (u, w)v - (u, v)w$$
$$(u \times v) \times w = (u, w)v - (v, w)u$$

Dimostrazione: Il modo più indolore per verificare queste identità è il calcolo diretto. Scriviamo i vettori come colonne.

$$u \times (v \times w) = \begin{pmatrix} u_1 \\ u_2 \\ u_3 \end{pmatrix} \times \begin{pmatrix} v_2w_3 - v_3w_2 \\ v_3w_1 - v_1w_3 \\ v_1w_2 - v_2w_1 \end{pmatrix} =$$

$$= \begin{pmatrix} \begin{vmatrix} u_2 & v_3w_1 - v_1w_3 \\ u_3 & v_1w_2 - v_2w_1 \end{vmatrix} \\ -\begin{vmatrix} u_1 & v_2w_3 - v_3w_2 \\ u_3 & v_1w_2 - v_2w_1 \end{vmatrix} \\ \begin{vmatrix} u_1 & v_2w_3 - v_3w_2 \\ u_2 & v_3w_1 - v_1w_3 \end{vmatrix} \end{pmatrix} =$$

$$= \begin{pmatrix} u_2v_1w_2 - u_2v_2w_1 - u_3v_3w_1 + u_3v_1w_3 \\ u_3v_2w_3 - u_3v_3w_2 - u_1v_1w_2 + u_1v_2w_1 \\ u_1v_3w_1 - u_1v_1w_3 - u_2v_2w_3 + u_2v_3w_2 \end{pmatrix} =$$

$$= \begin{pmatrix} v_1(u_2w_2 + u_3w_3) - w_1(u_2v_2 + u_3v_3) \\ v_2(u_3w_3 + u_1w_1) - w_2(u_3v_3 + u_1v_1) \\ v_3(u_1w_1 + u_2w_2) - w_3(u_1v_1 + u_2v_2) \end{pmatrix} =$$

$$= \begin{pmatrix} v_1(u_2w_2 + u_3w_3 + u_1w_1) - w_1(u_2v_2 + u_3v_3 + u_1v_1) \\ v_2(u_3w_3 + u_1w_1 + u_2w_2) - w_2(u_3v_3 + u_1v_1 + u_2v_2) \\ v_3(u_1w_1 + u_2w_2 + u_3w_3) - w_3(u_1v_1 + u_2v_2 + u_3v_3) \end{pmatrix} =$$

$$= (u, w)v - (u, v)w$$

Questa è la prima identità di Graßmann. Nella seconda usiamo l'antisimmetria del prodotto vettoriale (esercizio 50):

$$(u \times v) \times w = -w \times (u \times v) = -((w, v)u - (w, u)v)$$
$$= (u, w)v - (v, w)u$$

Lemma: $(u \times v, w) = (u, v \times w) = \det(u, v, w)$.

Dimostrazione: Esercizio 51.

Proposizione: $|v \times w|^2 = |v|^2|w|^2 - (v, w)^2$

Dimostrazione: $|v \times w|^2 = (v \times w, v \times w) = (v, w \times (v \times w)) = (v, (w, w)v - (w, v)w) = |w|^2(v, v) - (w, v)(v, w) = |v|^2|w|^2 - (v, w)^2$.

Identità di Jacobi: $u \times (v \times w) + v \times (w \times u) + w \times (u \times v) = 0$.

Dimostrazione: Esercizio 52.

Osservazione: Il prodotto vettoriale, considerato come operazione binaria su $\mathbb{R}^3$, è bilineare, antisimmetrico e non associativo, perché dalle identità di Graßmann si vede che in genere

$$u \times (v \times w) \neq (u \times v) \times w.$$

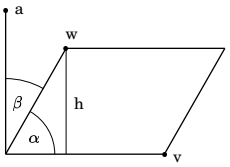
Le identità di Jacobi sono un'importante alternativa alla legge associativa in un anello: $(\mathbb{R}^3, +, \times)$ è un'algebra di Lie.

Le idee di Carl Gustav Jacob Jacobi (1804-1851), Hermann Graßmann (1809-1977) e Marius Sophus Lie (1842-1899) sono ancora oggi importanti in molti rami della matematica.

Area di un parallelogramma

Consideriamo due vettori linearmente indipendenti v e w in $\mathbb{R}^n$. Insieme all'origine essi determinano un parallelogramma la cui area, con le notazione nella figura, è uguale a $|v|h = |v||w||\sin\alpha|$, dove abbiamo usato il valore assoluto del seno per non essere obbligati a un particolare orientamento dell'angolo α.

Assumiamo adesso che $v, w \in \mathbb{R}^2$. In questo caso l'area del parallelogramma può essere espressa come valore assoluto del determinante dei due vettori, come adesso vediamo. Questo risultato può essere ottenuto in modo elegante se introduciamo di nuovo il vettore magico $a = (a_1, a_2) = (-v_2, v_1)$ che si ottiene girando v per 90 gradi in senso antiorario.



In primo luogo $|\sin\alpha| = |\cos\beta|$, per cui

$$h = |w||\sin\alpha| = |w||\cos\beta| = \frac{|w||(a, w)|}{|w||a|} =$$
$$= \frac{|(a, w)|}{|a|} = \frac{|(a, w)|}{|v|} = \frac{|\det(v, w)|}{|v|}$$

dove abbiamo usato la formula fondamentale $(a, x) = \det(v, x)$ valida per ogni $x \in \mathbb{R}^2$ (esercizio 45) e il fatto che $|a| = |v|$.

L'area del parallelogramma è uguale a $h|v|$, quindi uguale a

$$|\det(v, w)| = |v_1w_2 - v_2w_1|.$$

L'area di un parallelogramma nel piano e perciò uguale al valore assoluto del determinante che si ottiene dai due vettori. Se questi sono linearmente dipendenti, l'area è evidentemente nulla così come il determinante, e vediamo che la formula vale per una coppia qualsiasi di vettori del piano. Una formula analoga vale in $\mathbb{R}^n$ come dimostreremo per $n = 3$.

Significato geometrico di $v \times w$

Due vettori linearmente indipendenti v e w in $\mathbb{R}^3$ formano, insieme all'origine, un triangolo non degenere in cui denotiamo con α l'angolo nell'origine.

Per la proposizione a lato e le formule a pagina 11 abbiamo

$$|v \times w|^2 = |v|^2|w|^2 - (v, w)^2 =$$
$$= |v|^2|w|^2 - |v|^2|w|^2\cos^2\alpha =$$
$$= |v|^2|w|^2(1 - \cos^2\alpha) =$$
$$= |v|^2|w|^2\sin^2\alpha$$

per cui

$$|v \times w| = |v||w||\sin\alpha|.$$

$v \times w$ è quindi un vettore ortogonale a v e w di lunghezza uguale all'area del parallelogramma racchiuso da v e w ed è orientato in modo che i vettori v, w e $v \times w$ formino un sistema destrogiro come vedremo a pagina 33.

Piani nello spazio

Un piano P in uno spazio vettoriale reale V possiede una rappresentazione

$$P = p + \mathbb{R}v + \mathbb{R}w$$

con $p, v, w \in V$ e dove v e w sono linearmente indipendenti. p appartiene a P e può essere sostituito da qualsiasi altro punto del piano.

Un punto $x \in V$ appartiene al piano P se e solo se $x - p$ è combinazione lineare di v e w, quindi, essendo v e w linearmente indipendenti, se e solo se

$$(x - p) \wedge v \wedge w = 0$$

come segue dal nostro lemma a pagina 30.

Assumiamo adesso che $V = \mathbb{R}^3$. In questo caso per la prop. 29/2 la condizione $(x - p) \wedge v \wedge w = 0$ è equivalente a

$$\det(x - p, v, w) = 0$$

Dalla proposizione 30/1 sappiamo però che

$$\det(x - p, v, w) = (x - p, v \times w)$$

e quindi x appartiene al piano se e solo se $(x - p, v \times w) = 0$. Ciò dal lato geometrico significa che x appartiene a P se e solo se $x - p$ è ortogonale al prodotto vettoriale $v \times w$, mentre dal lato analitico ci fornisce anche un'equazione che descrive il piano:

$$(x, v \times w) = (p, v \times w)$$

Se rappresentiamo i vettori tramite i loro componenti,

$$\begin{aligned} x &= (x_1, x_2, x_3) \\ p &= (p_1, p_2, p_3) \\ v &= (v_1, v_2, v_3) \\ w &= (w_1, w_2, w_3) \end{aligned}$$

questa equazione è un'equazione scalare nelle tre incognite x_1, x_2, x_3:

$$a_1 x_1 + a_2 x_2 + a_3 x_3 = c$$

con

$$\begin{aligned} a_1 &= v_2 w_3 - v_3 w_2 \\ a_2 &= v_3 w_1 - v_1 w_3 \\ a_3 &= v_1 w_2 - v_2 w_1 \\ c &= (p, v \times w) \end{aligned}$$

che possiamo scrivere nella forma

$$(a, x) = (a, p)$$

in analogia con quanto abbiamo visto a pagina 29 per la retta nel piano, con $a := (a_1, a_2, a_3)$.

Sia viceversa data un'equazione della forma

$$a_1 x_1 + a_2 x_2 + a_3 x_3 = c$$

con $a := (a_1, a_2, a_3) \neq (0, 0, 0)$. Anche stavolta è facile trovare un punto $p := (p_1, p_2, p_3)$ tale che $c = a_1 p_1 + a_2 p_2 + a_3 p_3$, e allora l'equazione può essere scritta nella forma

$$(a, x) = (a, p)$$

ossia

$$(a, x - p) = 0$$

L'insieme delle soluzioni di questa equazione consiste quindi di tutti i punti x per cui il vettore $x - p$ è ortogonale al vettore a. Ciò intuitivamente dimostra già che l'insieme delle soluzioni forma un piano, per il quale comunque possiamo facilmente trovare una rappresentazione parametrica:

Sia ad esempio $a_3 \neq 0$, allora l'equazione $a_1 x_1 + a_2 x_2 + a_3 x_3 = c$ è equivalente (cfr. pag. 5) a

$$\frac{a_1}{a_3} x_1 + \frac{a_2}{a_3} x_2 + x_3 = \frac{c}{a_3}$$

cioè a

$$x_3 = \frac{c}{a_3} - \frac{a_1}{a_3} x_1 - \frac{a_2}{a_3} x_2$$

Per ogni scelta di x_1 e x_2 otteniamo un valore di x_3 tale che $x := (x_1, x_2, x_3)$ sia una soluzione dell'equazione e viceversa, quindi le soluzioni sono esattamente i punti rappresentabili nella forma

$$\begin{aligned} x_1 &= t \\ x_2 &= s \\ x_3 &= \frac{c}{a_3} - \frac{a_1}{a_3} t - \frac{a_2}{a_3} s \end{aligned}$$

per $t, s \in \mathbb{R}$ e quindi, se poniamo

$$\begin{aligned} p &:= (0, 0, \frac{c}{a_3}) \\ v &:= (1, 0, -\frac{a_1}{a_3}) \\ w &:= (0, 1, -\frac{a_2}{a_3}) \end{aligned}$$

le soluzioni costituiscono esattamente il piano

$$P = p + \mathbb{R}v + \mathbb{R}w.$$

Proiezione di un punto su un piano

Siano dati un piano P e un punto q di $\mathbb{R}^3$. La retta sia descritta dall'equazione

$$(a, x) = c$$

Vogliamo calcolare la proiezione ortogonale m di q su P. Il vettore $q - m$ deve essere ortogonale al piano e quindi parallelo al vettore a, per cui

$$m = q + sa$$

per qualche valore $s \in \mathbb{R}$ che otteniamo dall'equazione $(a, m) = c$ che m come punto del piano deve soddisfare. Quindi

$$\begin{aligned} c = (a, m) &= (a, q + sa) = \\ &= (a, q) + s(a, a) = (a, q) + s|a|^2 \end{aligned}$$

per cui

$$s = \frac{c - (a, q)}{|a|^2}$$

e quindi

$$m = q + \frac{c - (a, q)}{|a|^2} a$$

Si noti la completa analogia con la formula derivata a pagina 29 per la proiezione di un punto su una retta nel piano. Cos'è che hanno in comune le due situazioni?

La ragione è che entrambe le volte calcoliamo la proiezione ortogonale di un punto su un insieme descritto da un'equazione della forma

$$(a, x) = c$$

Infatti, assumiamo che siano dati un vettore $a = (a_1, \ldots, a_n) \neq 0$ di $\mathbb{R}^n$ e un'equazione $(a, x) = c$ per qualche $c \in \mathbb{R}$.

Nel corso di Geometria si imparerà che una tale equazione descrive un *iperpiano* di $\mathbb{R}^n$, cioè un sottospazio affine E di dimensione $n - 1$. Ciò implica che anche in questo caso, se scegliamo un punto arbitrario p dell'iperpiano (cioè un punto p tale che $(a, p) = c$), E consista esattamente di quei punti x per chi il vettore $x - p$ è ortogonale ad a e che viceversa un vettore b è ortogonale all'iperpiano se e solo se è parallelo ad a (qui entra l'ipotesi che la dimensione di E sia $n-1$). A questo punto il calcolo e le formule coincidono con quanto ottenuto per $\mathbb{R}^2$ e $\mathbb{R}^3$: La proiezione ortogonale m di un punto q su E è data da

$$m = q + \frac{c - (a, q)}{|a|^2} a$$

la distanza di q dall'iperpiano, cioè la lunghezza di $q - m$, da

$$\begin{aligned} |q - m| &= \frac{|(a, q) - c|}{|a|} = \\ &= \frac{|a_1 q_1 + \cdots + a_n q_n - c|}{\sqrt{a_1^2 + \cdots + a_n^2}} \end{aligned}$$

Il piano passante per tre punti

p, q ed r siano tre punti di uno spazio vettoriale reale V. I vettori $q-p$ ed $r-p$ sono linearmente indipendenti se e solo se i punti p, q ed r non stanno sulla stessa retta (esercizio 53). In questo caso essi generano un piano

$$P = p + \mathbb{R}(q-p) + \mathbb{R}(r-p)$$

a cui appartengono p, $q = p + (q-p)$ ed $r = p + (r-p)$.

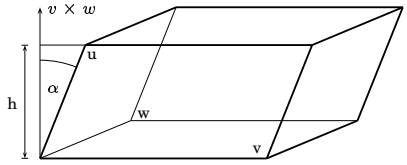
Il volume

Un *parallelepipedo*, analogo in dimensioni superiori del parallelogramma, centrato nell'origine di $\mathbb{R}^n$ è determinato da n vettori $v_1, \dots, v_n$ e può essere descritto analiticamente come l'insieme di tutte le combinazioni lineari

$$\lambda_1 v_1 + \cdots + \lambda_n v_n$$

in cui $0 \le \lambda_i \le 1$ per ogni $i = 1, \dots, n$. Il suo volume è uguale al valore assoluto del determinante della matrice le cui colonne sono i vettori $v_1, \dots, v_n$ come già dimostrato a pagina 31 per il caso $n = 2$.

Dimostriamo la formula per $n = 3$. Il parallelepipedo sia generato dai vettori u, v e w. Se questi sono linearmente dipendenti, il parallelepipedo è degenere (tutto contenuto in un piano) e il suo volume 3-dimensionale è uguale a zero così come $\det(u, v, w)$. Altrimenti abbiamo una situazione come nella figura.



È chiaro che il volume del parallelepipedo è uguale all'area del parallelogramma di base generato da v e w moltiplicato per l'altezza h. Abbiamo visto a pagina 31 che l'area del parallelogramma è uguale a $|v \times w|$, mentre l'altezza è uguale a

$$h = |u||\cos\alpha| = \frac{|u||(u, v \times w)|}{|u||v \times w|} = \\ = \frac{|(u, v \times w)|}{|v \times w|}$$

per cui

$$h|v \times w| = |(u, v \times w)| = |\det(u, v, w)|.$$

Abbiamo usato la formula che lega prodotto scalare e coseno a pagina 11 e la proposizione 30/1 che esprime il prodotto scalare tra un vettore e un prodotto vettoriale come determinante.

Orientamento

Denotiamo con $\mathbb{R}^n_m$ l'insieme delle matrici

$$A := \begin{pmatrix} a^1_1 & \dots & a^1_m \\ & \dots & \\ a^n_1 & \dots & a^n_m \end{pmatrix}$$

con n righe ed m colonne a coefficienti reali, i cui vettori colonna (che sono vettori di $\mathbb{R}^n$) verranno denotati con $a_1, \dots, a_m$ (oppure spesso con $e_1, \dots, e_n$, quando per $n = m$ formano una base). Abbiamo scritto gli indici di riga in alto come già a pagina 28.

Per $n = m$ la matrice diventa quadratia e possiamo considerare il suo determinante

$$\det A = \det(a_1, \dots, a_n).$$

In analogia con quanto visto nelle proposizioni 29/1 e 29/2, questo determinante è diverso da zero se e solo i vettori $a_1, \dots, a_n$ sono linearmente indipendenti e formano quindi una base di $\mathbb{R}^n$. Ciò verrà dimostrato nel corso di Geometria.

Il determinante suddivide quindi le matrici $A \in \mathbb{R}^n_n$ in due classi: quelle matrici il cui determinante è $\neq 0$ e le cui colonne formano quindi una base di $\mathbb{R}^n_n$, e le matrici il cui determinante è uguale a zero e le cui colonne sono linearmente dipendenti.

Questa suddivisione sussiste per ogni campo di scalari al posto di $\mathbb{R}$. In $\mathbb{R}$ però possiamo distinguere le basi ancora più finemente utilizzando l'ordine sulla retta reale. Infatti un numero reale $\neq 0$ o è > 0 oppure è < 0. Il determinante può essere perciò usato per suddividere ulteriormente le matrici con determinante $\neq 0$ in quelle che hanno determinante > 0 e quelle che hanno determinante < 0.

Mentre la lineare indipendenza dei vettori colonna di una matrice non dipende dall'ordine in cui questi vettori compaiono nella matrice, perché se cambiamo l'ordine delle colonne cambiamo solo al massimo il segno del determinante, l'essere il determinante > 0 o < 0 dipende dall'ordine in cui le colonne della matrice sono elencate e se quindi queste colonne le consideriamo come componenti di una base di $\mathbb{R}^n$, dobbiamo parlare di *basi ordinate*. Perciò, quando diciamo che $e_1, \dots, e_n$ è una base ordinata di $\mathbb{R}^n$, intendiamo che fissiamo anche l'ordine in cui i vettori $e_1, \dots, e_n$ sono elencati.

Definizione: Una base ordinata $e_1, \dots, e_n$ di $\mathbb{R}^n$ si dice *positivamente orientata* se $\det(e_1, \dots, e_n) > 0$ e *negativamente orientata* quando invece $\det(e_1, \dots, e_n) < 0$.

Per quanto visto, ogni base ordinata è o positivamente orientata oppure negativamente orientata. Inoltre, se in una base ordinata scambiamo due dei suoi elementi, otteniamo una base di orientamento opposto.

Consideriamo adesso due vettori v e w di $\mathbb{R}^3$ insieme al loro prodotto vettoriale.

Proposizione: v e w siano due vettori linearmente indipendenti di $\mathbb{R}^3$. Allora i vettori $v, w, v \times w$ sono linearmente indipendenti e formano una base ordinata positivamente orientata.

Dimostrazione: Abbiamo già osservato a pagina 27 che $v \times w \neq 0$ se e solo se, come nella nostra ipotesi, v e w sono linearmente indipendenti. Perciò

$$\det(v, w, v \times w) = (v \times w, v \times w) = \\ = |v \times w|^2 > 0$$

come si vede applicando il lemma a pagina 31 e la proposizione 30/1.

Tentiamo adesso di dare, almeno intuitivamente, un'interpretazione geometrica dell'orientamento di una base ordinata e_1, e_2, e_3 di $\mathbb{R}^3$, limitandoci al caso che e_1 si trovi sul lato positivo dell'asse x ed e_3 sul lato positivo dell'asse z, mentre e_2 si trovi nel piano x, y.

Consideriamo prima il caso che e_2 si trovi sul lato positivo dell'asse y. In questo caso $e_1 = (\lambda, 0, 0)$, $e_2 = (0, \mu, 0)$ ed $e_3 = (0, 0, \nu)$ con $\lambda, \mu, \nu > 0$ e $\det(e_1, e_2, e_3) =$

$$= \begin{vmatrix} \lambda & 0 & 0 \\ 0 & \mu & 0 \\ 0 & 0 & \nu \end{vmatrix} = \lambda\mu\nu > 0$$

Quindi in questo caso la base ordinata e_1, e_2, e_3 è positivamente ordinata. Facciamo adesso ruotoare e_2 nel piano x, y ponendo

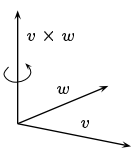
$$e_2 = e_2(t) = (\mu\cos t, \mu\sin t, 0)$$

per $0 \le t < 360°$. Allora per il determinante abbiamo $\det(e_1, e_2(t), e_3) =$

$$= \begin{vmatrix} \lambda & \mu\cos t & 0 \\ 0 & \mu\sin t & 0 \\ 0 & 0 & \nu \end{vmatrix} = \lambda\mu\nu\sin t$$

e vediamo che la base ordinata e_1, e_2, e_3 rimane positivamente orientata per $0 < t < 180°$ ed è invece negativamente orientata per $180° < t < 360°$. Per $t = 0$ oppure $t = 180°$ il vettore e_2 è parallelo ad e_1 e quindi non abbiamo più una base.

Questa considerazione, purché incompleta, descrive comunque la situazione nel caso di una base della forma $v, w, v \times w$ che, come abbiamo visto, è positivamente orientata e che, mediante una rotazione di $\mathbb{R}^3$, può sempre essere portata nella posizione appena descritta - si imparerà nel corso di Geometria che una rotazione lascia invariante il determinante di una base.



Forth e PostScript

*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. Un piccolo e ben funzionante interprete è **pfe**.*

*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.

Sul sito di Adobe (www.adobe.com/) si trovano manuali e guide alla programmazione in PostScript.

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**.

Questa settimana

34 Forth e PostScript
Lo stack
Programmare in Forth

35 Usare ghostscript
Il comando run di PostScript
Usare def
Diagrammi di flusso per lo stack
Lo stack dei dizionari

36 Argomenti di una macro
show e selectfont
if e ifelse
Cerchi con PostScript

37 Coordinate polari nel piano
Coordinate cilindriche
Coordinate polari nello spazio
Rotazioni nel piano

Programmare in Forth

Qualche passo dal libro *Stack computers* 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."

In ambiente Unix l'interattività tradizionale in Forth può essere sostituita con un'ancora più comoda organizzazione del programma su più files che come programma *script* (cfr. pag. 22), soprattutto se combinato con comandi di Emacs, diventa a sua volta praticamente interattivo.

Usare ghostscript

Sotto Unix si può battere **ghostscript** oppure semplicemente **gs** oppure meglio ad esempio **gs -g400x300** per avere una finestra di 400 x 300 pixel; sotto Windows cliccare sull'icona. Sotto Unix dovrebbe apparire una finestra per la grafica mentre sulla shell è attivo l'interprete che aspetta comandi. Proviamo prima a impostare alcuni comandi a mano (battere invio alla fine di ogni riga) facendo in modo che la finestra grafica rimanga visibile mentre battiamo i comandi:

```
0.05 setlinewidth
33.3 33.3 scale
0 0 moveto 5 4 lineto stroke
/rosso {1 0 0 setrgbcolor} def
rosso 5 4 moveto 8 2 lineto stroke
/nero {0 0 0 setrgbcolor} def
nero 8 2 moveto 0 0 lineto stroke
/giallo {1 1 0 setrgbcolor} def
giallo 7 5 moveto 4 5 3 0 360 arc fill
nero 7 5 moveto 4 5 3 0 360 arc stroke
rosso 6.5 5 moveto 4 5 2.5 0 360 arc fill
nero 6.5 5 moveto 4 5 2.5 0 360 arc stroke
/blu {0 0 1 setrgbcolor} def
blu 6 5 moveto 4 5 2 0 360 fill
blu 6 5 moveto 4 5 2 0 360 arc fill
nero 6 5 moveto 4 5 2 0 360 arc stroke
giallo 5.5 5 moveto 4 5 1.5 0 360 arc fill
nero 5.5 5 moveto 4 5 1.5 0 360 arc stroke
/verde {0 1 0 setrgbcolor} def
verde 4 5 moveto 8 6 lineto 7 7 lineto 4 5 lineto fill
nero 4 5 moveto 8 6 lineto 7 7 lineto 4 5 lineto stroke
quit
```

Per vedere una nuova pagina si usa **showpage**; questo comando è anche necessario se il file è destinato alla stampa.

Il comando run di PostScript

Invece di battere i comandi dalla tastiera li possiamo anche inserire in un file. Creiamo ad esempio una cartella **/home/sis/ps** e in essa un file **alfa** in cui trascriviamo i comandi dell'articolo precedente, tralasciando però il **quit** finale che serve solo per uscire dall'interprete e che non ci permetterebbe di osservare l'immagine.

A questo punto dopo aver aperto l'interprete possiamo battere **(alfa) run** se ci troviamo nella stessa cartella del file, altrimenti dobbiamo indicare il nome completo del file, quindi **(/home/sis/alfa) run**. Le parentesi tonde in PostScript servono per racchiudere una stringa e prendono quindi il posto delle virgolette in molti altri linguaggi. Sotto Windows si può usare ad esempio **(c:/sis/alfa) run** (con barre semplici) oppure **(c:\\sis\\alfa) run** (con backslash raddoppiati).

Il comando **run** può essere utilizzato anche all'interno di un file per chiedere l'esecuzione di altri files, tipicamente di qualche nostra raccolta di abbreviazioni.

Usare def

Le abbreviazioni vengono definite secondo la sintassi

/abbreviazione significato **def**

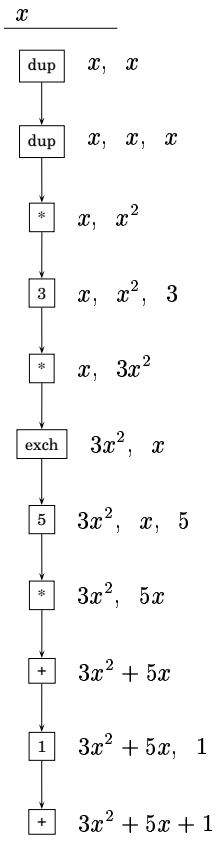
Se il significato è un operatore eseguibile bisogna racchiudere le operazioni tra parentesi graffe come abbiamo fatto sopra per i colori o a pagina 34 per *add* e *mul*, per impedire che vengano eseguite già la prima volta che l'interprete le incontra, cioè nel momento in cui legge l'abbreviazione.

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

```
/e 2.71828182845904523536 def
/pi 3.14159265358979323846 def
```

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, simili ai diagrammi di flusso per altri linguaggi visti a pagina 14, dove però adesso indichiamo ogni volta gli elementi più a destra dello stack vicino ad ogni istruzione. Utilizziamo due nuove istruzioni: **dup**, che duplica l'ultimo elemento dello stack (vedremo subito perché), ed **exch** che scambia gli elementi più a destra dello stack. Assumiamo inoltre di avere definito gli operatori **+** e * invece di **mul** e **add** come a pagina 34. All'inizio l'ultimo elemento dello stack è x.



La macroistruzione che corrisponde a questo diagramma di flusso è

*/f {dup dup * 3 * exch 5 * + 1 +} def*

Lo stack dei dizionari

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

```
4 dict begin /x 1 def /y 2 def /z 3 def /w (Rossi) def
...
end
```

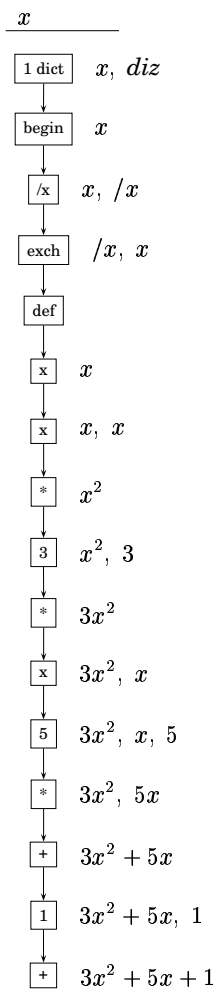
viene creato un dizionario di almeno 4 voci (il cui numero viene comunque automaticamente aumentato se vengono definite più voci). Tra *begin* e *end* tutte le abbreviazioni si riferiscono a questo dizionario se in esso si trova una tale voce (altrimenti il significato viene cercato nel prossimo dizionario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

Argomenti di una macro in PostScript

Consideriamo ancora la funzione $f(x) = 3x^2 + 5x + 1$. La macroistruzione che abbiamo trovato a pagina 35, 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 * 3 * x 5 * + 1 + end} def*

Esaminiamo anche questa espressione mediante un diagramma di flusso.



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 *x*. Questi due elementi devono essere messi nell'ordine giusto mediante un *exch* per poter applicare il *def* che inserisce la nuova abbreviazione per *x* 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.

show e selectfont

Abbiamo già osservato a pagina 35 che il PostScript usa le parentesi tonde per raccogliere le stringhe. La visualizzazione avviene mediante il comando **show** che però deve essere preceduto dall'indicazione del punto dove la stringa deve apparire e del font. Il font viene definito come nel seguente esempio:

*/times-20 {/**Times-Roman** 20 scala **div selectfont**} def*

dopo aver definito la scala ad esempio con **/scala 33.3 def**. Adesso possiamo visualizzare una stringa:

times-20 1 8 moveto (Il cerchio si chiude) **show**

gsave 8 8 moveto (e) (d) (u) (i) (h) (c) ( ) (i) (s) ( )
(o) (i) (h) (c) (r) (e) (c) ( ) (l) (I) ( ) (*)
22 {show 360 23 **div neg rotate**} **repeat grestore**

Il comando **gsave** viene utilizzato per salvare le impostazioni grafiche prima di effettuare cambiamenti, ad esempio prima di una rotazione, mentre **grestore** ripristina il vecchio stato grafico – anche qui si usa uno stack!. **a b div** è i quoziente $\frac{a}{b}$, **a neg** è $-a$, mentre **t rotate** ruota il sistema di coordinate per t gradi in senso antiorario. **10** {**operazione**} **repeat** ripete un'operazione 10 volte. Provare con ghostscript!

---

if e ifelse

Illustriamo l'uso di *if* e *ifelse* con due esempi.

a m resto calcola il resto di a modulo m anche per $m < 0$ utilizzando la funzione **mod** del PostScript che dà il resto corretto invece solo per $m > 0$, **n fatt** è il fattoriale di n.

```
/resto {2 dict begin /m exch def /a exch def
  m 0 lt {/a a neg def} if a m mod end} def

/fatt {1 dict begin /n exch def
  n 0 eq {1} {n 1 - fatt n *} ifelse end} def
```

Per trasformare un numero in una stringa si può usare

*/stringa-numerica {20 **string cvs**} def*

Esempi da provare:

2 6 moveto 117 40 resto stringa-numerica show
3 6 moveto 8 fatt stringa-numerica show

Cerchi con PostScript

/cerchio {3 dict begin /r exch def /y exch def /x exch def
*gsave **newpath** x y r 0 360 **arc** stroke grestore end} def*

/cerchiopieno {3 dict begin /r exch def /y exch def /x exch def
*gsave newpath x y r 0 360 arc **fill** grestore end} def*

/rosetta {5 dict begin /f exch def /n exch def /y exch def
*/x exch def /alfa 360 n div def gsave x y **translate***
n {f alfa rotate} repeat grestore end} def

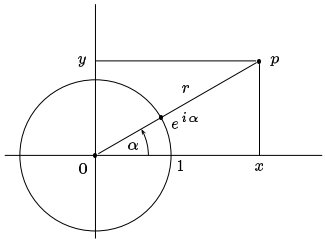
Provare adesso (dopo le solite impostazioni):

verde 3 3 2.3 cerchiopieno nero 3 3 2.3 cerchio
3 3 7 {giallo 1.5 0 0.5 cerchiopieno
nero 1.5 0 0.5 cerchio} rosetta
rosso 3 3 0.8 cerchiopieno nero 3 3 0.8 cerchio

Per cancellare una pagina si usa **erasepage**.

Coordinate polari nel piano

Sia $p = (x, y)$ un punto del piano reale.



Si vede che, se $p \neq (0,0)$, allora

$$\boxed{\begin{aligned} x &= r\cos\alpha \\ y &= r\sin\alpha \end{aligned}} \qquad (*)$$

dove $r = \sqrt{x^2+y^2}$, mentre l'angolo α è univocamente determinato se chiediamo $0 \leq \alpha < 2\pi$.

Nel caso $p = (0,0)$ la rappresentazione (*) rimane valida con $r = 0$ e qualsiasi α, la biiettività della (*) viene quindi meno nel punto $p = (0,0)$.

Scriviamo adesso $e^{i\alpha} := (\cos\alpha, \sin\alpha)$ come abbiamo già fatto nel disegno; allora la relazione (*) può anche essere scritta nella forma

$$p = re^{i\alpha}$$

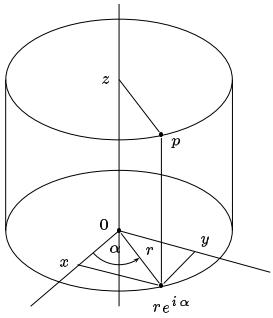
Questo prodotto di r con $e^{i\alpha}$ può essere interpretato come prodotto dello scalare reale r con il vettore $e^{i\alpha}$ di $\mathbb{R}^2$ ed è allo stesso tempo il prodotto dei numeri complessi r ed $e^{i\alpha}$ come vedremo nel prossimo numero.

Coordinate cilindriche nello spazio

Un punto $p = (x, y, z)$ dello spazio può essere rappresentato nella forma

$$\boxed{\begin{aligned} x &= r\cos\alpha \\ y &= r\sin\alpha \\ z &= z \end{aligned}}$$

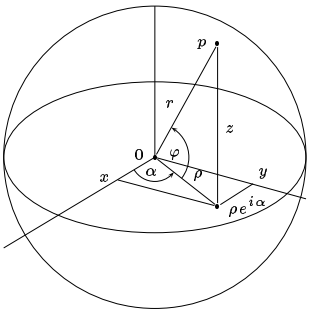
come si vede dalla figura.



La rappresentazione è univoca per $(x, y) \neq (0,0)$, quindi per tutti i punti che non si trovano sull'asse z.

Coordinate polari (o sferiche) nello spazio

Un punto $p = (x, y, z)$ dello spazio tridimensionale può anche essere rappresentato come nella figura seguente:



Avendo $\rho = r\cos\varphi$, si vede che

$$\boxed{\begin{aligned} x &= r\cos\alpha\cos\varphi \\ y &= r\sin\alpha\cos\varphi \\ z &= r\sin\varphi \end{aligned}}$$

con

$$\begin{aligned} &r \geq 0 \\ &0 \leq \alpha < 2\pi \\ &-\frac{\pi}{2} \leq \varphi \leq \frac{\pi}{2} \end{aligned}$$

Questa rappresentazione è quella che si usa nelle coordinate geografiche di un punto della terra o della sfera celeste:

$\alpha =$ longitudine, $\varphi =$ latitudine.

Anche in questo caso la corrispondenza non è biiettiva, perché non solo per $p = (0,0,0)$ la rappresentazione è valida per $r = 0$ e valori arbitrari di α e φ, ma anche per ogni altro punto $\neq (0,0,0)$ dell'asse z bisogna porre $\varphi = 90°$ e quindi $\cos\varphi = 0$ e $\sin\varphi = 1$ se $z > 0$ oppure $\varphi = -90°$ e quindi $\cos\varphi = 0$ e $\sin\varphi = -1$ se $z < 0$, e allora ogni α va bene. Quindi su tutta l'asse z le coordinate polari non sono univocamente determinate.

Spesso al posto di φ si usa

$$\theta := 90° - \varphi$$

quindi $\cos\varphi = \sin\theta, \sin\varphi = \cos\theta$.

***

Molte funzioni della matematica e della fisica presentano *simmetrie*. A una funzione $f = f(x, y, z)$ definita su $\mathbb{R}^3$ (per semplicità, ma spesso bisognerà studiare bene il più adatto dominio di definizione) possiamo associare la funzione $g = g(r, \alpha, \varphi)$ definita da

$$g(r,\alpha,\varphi) := f(r\cos\alpha\cos\varphi, r\sin\alpha\cos\varphi, r\sin\varphi)$$

che in caso di simmetrie può avere una forma analitica molto più semplice della f.

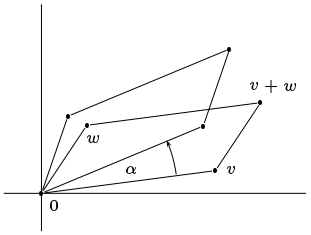
$f(x, y, z) = x^2 + y^2 + z^2$ ad esempio diventa così $g(r, \alpha, \varphi) = r^2$, una funzione di una sola variabile notevolmente più semplice. Altre volte una funzione dipende solo dalla direzione e quindi non da r; in questo caso g è una funzione di sole due variabili e anche questa è una semplificazione. Nello stesso modo si usano le coordinate cilindriche e le coordinate polari piane.

Rotazioni nel piano

Consideriamo l'applicazione f_α da $\mathbb{R}^2$ in $\mathbb{R}^2$ che consiste nel ruotare un punto $v = (v_1, v_2)$ per l'angolo fissato α in senso antiorario. È chiaro che $f_\alpha(\lambda v) = \lambda f_\alpha(v)$ per ogni $\lambda \in \mathbb{R}$ e dal disegno si vede che anche

$$f_\alpha(v + w) = f_\alpha(v) + f_\alpha(w)$$

per $v, w \in \mathbb{R}^2$. Una rotazione è quindi un'applicazione lineare.



Sia e_1, e_2 la base canonica di $\mathbb{R}^2$. Allora $v = v_1e_1 + v_2e_2$, perciò $f_\alpha(v) = v_1 f_\alpha(e_1) + v_2 f_\alpha(e_2)$. Ma

$$f_\alpha(e_1) = \begin{pmatrix} \cos\alpha \\ \sin\alpha \end{pmatrix}$$

$$f_\alpha(e_2) = \begin{pmatrix} -\sin\alpha \\ \cos\alpha \end{pmatrix}$$

– vettore magico di $f_\alpha(e_1)$! Quindi

$$f_\alpha(v) = v_1 \begin{pmatrix} \cos\alpha \\ \sin\alpha \end{pmatrix} + v_2 \begin{pmatrix} -\sin\alpha \\ \cos\alpha \end{pmatrix} = \begin{pmatrix} v_1\cos\alpha - v_2\sin\alpha \\ v_1\sin\alpha + v_2\cos\alpha \end{pmatrix}$$

Se per una matrice

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \in \mathbb{R}^2_2$$

definiamo

$$Av = \begin{pmatrix} av_1 + bv_2 \\ cv_1 + dv_2 \end{pmatrix}$$

vediamo che possiamo prendere

$$A = \begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix}.$$

Notiamo anche che le colonne di A sono proprio $f_\alpha(e_1)$ e $f_\alpha(e_2)$.

ALGORITMI E STRUTTURE DI DATI

Corso di laurea in matematica | Anno accademico 2001/2002 | Numero 9 ◇ 19 Novembre 2001

Programmare in C

Un programma in C/C++ in genere viene scritto in più files, che conviene raccogliere nella stessa directory. Avrà anche bisogno di files che fanno parte dell'ambiente di programmazione o di una nostra raccolta di funzioni esterna al progetto e che si trovano in altre directory. Tutto insieme si chiama un progetto. I files del progetto devono essere compilati e collegati (linked) per ottenere un file eseguibile (detto spesso applicazione). Il programma in C/C++ costituisce il codice sorgente (source code) di cui la parte principale è contenuta in files che portano l'estensione ***.c****, mentre un'altra parte, soprattutto le dichiarazioni generali, è contenuta in files che portano l'estensione* ***.h*** *(da header, intestazione).*

Cos'è una dichiarazione? Il C può essere considerato come un linguaggio macchina universale, le cui operazioni hanno effetti diretti in memoria, anche se la locazione effettiva degli indirizzi non è nota al programmatore. Il compilatore deve quindi sapere quanto spazio deve riservare alle variabili (e a questo serve la dichiarazione del tipo delle variabili) e di quanti e di quale tipo sono gli argomenti e i risultati delle funzioni. Ogni file sorgente (con estensione ***.c****) viene compilato separatamente in un file oggetto (con estensione* ***.o****), cioè un file in linguaggio macchina. Se il file utilizza variabili e funzioni definite in altri files sorgente, bisogna che il compilatore possa conoscere almeno le dichiarazioni di queste variabili e funzioni, dichiarazioni che saranno contenute nei files* ***.h*** *che vengono inclusi mediante* ***# include*** *nel file* ***.c****.*

*Dopo aver ottenuto i files oggetto (****.o****) corrispondenti ai singoli files sorgenti (****.c****), essi devono essere collegati in un unico file eseguibile (a cui, sotto Unix, automaticamente dal compilatore viene assegnato il diritto di esecuzione) dal linker.*

I comandi di compilazione (compreso il linkage finale) possono essere battuti dalla shell, ma ciò deve avvenire ogni volta che il codice sorgente è stato modificato e quindi consuma molto tempo e sarebbe difficile evitare errori. In compilatori commerciali, soprattutto su altri sistemi, queste operazioni vengono spesso eseguite attraverso un'interfaccia grafica; sotto Unix normalmente questi comandi vengono raccolti in un cosiddetto ***makefile****, molto simile a uno script di shell, che viene poi eseguito mediante il comando* ***make****. I nostri makefile saranno molto semplici, mentre i makefile di programmi che devono girare su computer in ambienti e configurazioni diversi sono in genere più complessi, perché devono prevedere un adattamento a quegli ambienti e quelle configurazioni.*

Il programma minimo

```
// alfa.c
# include <stdio.h>

int main();
//////////////////////
int main()
{printf("Ciao!\n");}
```

La prima riga è un commento e contiene il nome del file; è un'abitudine utile soprattutto quando il file viene stampato. Una riga di commenti suddivide otticamente il file.

<**stdio.h**> è il file che contiene le dichiarazioni per molte funzioni di input/output, compresa la funzione **printf** che qui viene utilizzata.

int main() appare due volte; la prima volta si tratta della *dichiarazione* della funzione **main**, la seconda volta segue la sua *definizione* (che nel caso di funzioni corrisponde alla programmazione vera e propria), racchiusa tra parentesi graffe. Si noti che la dichiarazione termina invece con un punto e virgola. In entrambi i casi il nome della funzione è preceduto dal tipo del risultato (qui **int**).

`'\n'` nella funzione di output **printf** è il carattere di nuova riga. Si vede che stringhe sono racchiuse tra virgolette, i caratteri tra apostrofi.

Questa settimana

38 Programmare in C
Il programma minimo
I header generali

39 Il preprocessore
printf
I commenti
Calcoliamo il fattoriale

40 Comandi di compilazione
Il makefile
Il comando make
Come funziona make

41 for
Operatori logici
I numeri complessi
La formula di Euler

42 Il campo dei numeri complessi
La disuguaglianza di Cauchy-Schwarz
La disuguaglinza triangolare
La formula di de Moivre
Linux Day al dipartimento

I header generali

```
# include <assert.h>
# include <ctype.h>
# include <errno.h>
# include <fcntl.h>
# include <limits.h>
# include <locale.h>
# include <math.h>
# include <setjmp.h>
# include <signal.h>
# include <stdarg.h>
# include <stddef.h>
# include <stdio.h>
# include <stdlib.h>
# include <string.h>
# include <unistd.h>
# include <sys/ioctl.h>
# include <sys/param.h>
# include <sys/stat.h>
# include <sys/times.h>
# include <sys/types.h>
# include <sys/utsname.h>
# include <sys/wait.h>
# include <termio.h>
# include <time.h>
# include <ulimit.h>
# include <unistd.h>

# include <X11/cursorfont.h>
# include <X11/keysym.h>
# include <X11/Xatom.h>
# include <X11/Xlib.h>
# include <X11/Xos.h>
# include <X11/Xresource.h>
# include <X11/Xutil.h>
```

In genere solo pochi di questi header sono necessari; ad esempio i header per le funzioni grafiche (<**X11**/*>) sono superflui in programmi che non usano X Window.

Il preprocessore

Quando un file sorgente viene compilato, prima del compilatore vero e proprio entra in azione il *preprocessore*. Questo non crea un codice in linguaggio macchina, ma prepara una versione modificata del codice sorgente secondo direttive (*preprocessor commands*) date dal programmatore che successivamente verrà elaborata dal compilatore (in linea di principio almeno, perché in alcune implementazioni le due operazioni sono combinate in un unico passaggio).

Le direttive del preprocessore per noi più importanti sono **# include** e **# define** (lo spazio dopo # può anche mancare).

Se in un file sorgente si trova l'istruzione **# include "alfa.h"**, ciò significa che nella sorgente secondaria che il preprocessore prepara per il compilatore il file **alfa.h** verrà copiato esattamente in quella posizione come se fosse stato scritto nella versione originale. Per il nome del file valgono le regole solite, cioè un nome che inizia con / è un nome assoluto, altrimenti il nome è relativo alla directory in cui si trova il file sorgente.

Il secondo formato, ad esempio **# include** <**stdio.h**>, riconoscibile dalle parentesi angolate, viene usato per quei header che il sistema cerca in determinate directory (a quelle di default possono essere aggiunte altre). In questo caso, soprattutto in sistemi non Unix, il nome formale del header può anche non corrispondere a un file dello stesso nome. Sotto Linux questi files si trovano spesso in **/usr/include**, **/usr/include/sys** e **/usr/X11R6/include/X11**.

Le direttive **# define** vengono utilizzate per definire abbreviazioni. Queste abbreviazioni possono contenere parametri variabili e simulare funzioni; in tal caso si parla di *macroistruzioni* o semplicemente di *macro*. Le più semplici sono del tipo

```
# define base 40
# define nome "Giovanni Rossi"
# define stampa printf(
# define pc )
```

Il nome dell'abbreviazione consiste dei caratteri **[_A-Za-z0-9]**, rappresentati qui in una forma facilmente comprensibile presa in prestito dal Perl e non può iniziare con una cifra. Bisogna distinguere tra minuscole e maiuscole. Dopo il nome segue uno spazio (oppure una parentesi che inizia l'elenco dei parametri), e il resto della riga è l'espressione che verrà sostituita al nome dell'abbreviazione. Si noti che non appare il segno di uguaglianza. Come nel testo sorgente, una riga che termina con \ (a cui non deve seguire un carattere di spazio vuoto) viene, prima ancora dell'intervento del preprocessore, unita alla riga seguente.

Le abbreviazioni nelle due ultime righe possono essere usate per scrivere *stampa "Ciao.\n" pc* invece di *printf("Ciao.\n");*. È solo un esempio da non imitare naturalmente.

In C++ le direttive **# define** semplici vengono usate raramente, perché si possono usare *variabili costanti*. Sono invece piuttosto frequenti e tipiche nel C.

printf

Abbiamo già usato il **printf** a pagina 38; lo vediamo anche a destra nell'output del fattoriale:

```
for (n=0;n<=20;n++) printf("%2d! = %-12.0f\n", n,fattoriale(n));
```

Il primo parametro di **printf** è sempre una stringa. Questa può contenere delle *istruzioni di formato* (dette anche *specifiche di conversione*) che iniziano con `%` e indicano il posto e il formato per la visualizzazione degli argomenti aggiuntivi. In questo esempio `%2d` tiene il posto per il valore della variabile `n` che verrà visualizzata come intero di due cifre, mentre `-12.0f` indica una variabile di tipo **double** di al massimo 12 caratteri totali (compreso il punto decimale quindi), di cui 0 cifre dopo il punto decimale (che perciò non viene mostrato), allineati a sinistra a causa del - (l'allineamento di default avviene a destra). I formati più usati sono:

%d	intero	%f	double
%ld	intero lungo	%ud	intero senza segno
%c	carattere	%s	stringa
%%	carattere %		

I commenti

Normalmente i commenti vengono eliminati prima ancora che entri in attività il preprocessore. Il commento classico del C consiste di una parte del file sorgente compresa tra /* e */ (non contenuti in una stringa), che può estendersi su più righe. Esempio:

```
int n; /* Questo è un commento
su due righe */ n=7;
```

Molti compilatori C, anche il **gcc** della GNU, accettano anche i commenti nello stile C++, che spesso sono più comodi e più facilmente distinguibili. In questo formato se una riga contiene (sempre al di fuori di una stringa) //, allora tutto il resto della riga è considerato un commento, compresa la successione //, che viene quindi usata nello stesso modo come **#** negli shell script o ad esempio in Perl oppure **;** in Elisp (il linguaggio di programmazione che si usa per Emacs) e in molti linguaggi assembler.

Calcoliamo il fattoriale

Normalmente nel file **alfa.c** scriveremo solo la funzione **main** ed eventualmente poche altre funzioni di impostazione generale. Creiamo quindi un file apposito per gli esperimenti matematici e cominciamo con un programma per il prodotto fattoriale.

```
// matematica.c
# include "alfa.h"

double fattoriale (int n)
{double f; int k;
for (f=1,k=1;k<=n;k++) f*=k; return f;}
```

Per chiamare questa funzione modifichiamo il file **alfa.c** nel modo seguente:

```
// alfa.c
# include "alfa.h"

int main();
//////////////////
int main()
{int n;
for (n=0;n<=20;n++)
   printf("%2d! = %-12.0f\n",n,fattoriale(n));
exit(0);}
```

Il header standard <**stdio.h**> passa adesso nel nostro header di progetto **alfa.h** che contiene anche la dichiarazione della funzione **fattoriale**:

```
// alfa.h

# include <stdio.h>
///////////////
// matematica.c

double fattoriale(int);
```

Rimane da modificare la riga bersaglio (*target*) nel Makefile (pagina 40):

```
make: alfa.o matematica.o
```

Comandi di compilazione

A questo punto il nostro progetto consiste di due files sorgente (**alfa.c** e **matematica.c**). Potremmo adesso effettuare la compilazione battendo dalla shell i seguenti comandi, dopo aver creato una cartella **Oggetti** che conterrà i files oggetto affinché non affollino la directory del progetto:

```
gcc -o Oggetti/alfa.o -c alfa.c
gcc -o Oggetti/matematica.o -c matematica.c
gcc -o alfa Oggetti/*.o -lm -lc
```

La prima riga compila la sorgente **alfa.c** creando un file **alfa.o** nella directory **Oggetti**, e lo stesso vale per la riga successiva. L'ultima riga effettua il link, connette cioè i files **.o** e crea il programma eseguibile **alfa**, utilizzando la libreria matematica (di cui in verità in questo programma finora non abbiamo avuto bisogno) e la libreria del C.

Il makefile

Per non dover battere ogni volta i comandi di compilazione dalla tastiera, li scriviamo in un file apposito, un cosiddetto *makefile* che verrà poi utilizzato come descritto in seguito su questa pagina.

```
# Makefile

librerie = -L/usr/X11R6/lib -lX11 -lm -lc

VPATH=Oggetti

progetto: alfa.o matematica.o
TAB gcc -o alfa Oggetti/*.o $(librerie)

%.o: %.c alfa.h
TAB gcc -o Oggetti/$*.o -c $*.c
```

Lo stesso makefile si può usare per il C++, sostituendo **gcc** con **g++**. Se non si usa la libreria grafica, la parte

```
-L/usr/X11R6/lib -lX11
```

può essere tralasciata; in questo caso rimarrebbero quindi soltanto la libreria matematica e la libreria del C:

```
librerie = -lm -lc
```

Talvolta bisogna aggiungere altre librerie, ad esempio *-lcrypt* per la crittografia.

TAB denota il tasto tabulatore; non può qui essere sostituito da spazi. Non dimenticare di creare la directory **Oggetti**.

Il comando make

Il comando **make** senza argomenti effettua la verifica del primo blocco elementare del file **Makefile** (oppure, se esiste, del file **makefile**) nella directory in cui viene invocato.

Con **make a** si può invece ottenere direttamente la verifica del controllo a.

Esaminiamo il nostro makefile. La prima riga inizia con **#** ed è un commento. La riga che segue è un'abbreviazione; più avanti, invece di *$(librerie)* potremmo anche scrivere esplicitamente

```
TAB gcc -o alfa Oggetti/*.o -L/usr/X11R6/lib -lX11 -lm -lc
```

Si noti che qui, in modo simile a come avviene negli shell script, una variabile definita come x viene poi chiamata come *$(x)*; in verità ci sono alcune variazioni, ma per i nostri scopi il formato proposto è sufficiente (anche per programmi piuttosto grandi).

-L/usr/X11R6/lib significa che le librerie vengono cercate, oltre che nelle altre directory standard (soprattutto **/usr/lib**) anche nella directory **/usr/X11R6/lib**.

Da ogni file sorgente **.c** viene creato un file oggetto **.o** come segue dal secondo blocco elementare

```
%.o: %.c alfa.h
TAB gcc -o Oggetti/$*.o -c $*.c
```

in cui abbiamo usato le *GNU pattern conventions*. Con *VPATH=Oggetti* indichiamo al compilatore (o meglio al programma **make**) questa locazione.

Normalmente i comandi vengono ripetuti sullo schermo durante l'esecuzione e ciò è utile per controllare l'esecuzione del **make**; premettendo un **@** a una riga di comando, questo non appare sullo schermo. Si può anche mettere *.SILENT:* all'inizio del file.

Righe vuote vengono ignorate, ma è meglio separare i blocchi mediante righe vuote. Un \ alla fine di una riga fa in modo che la riga successiva venga aggiunta alla prima.

Come funziona make

La parte importante di un makefile (a cui si aggiungono regole piuttosto complicate per le abbreviazioni) sono i *blocchi elementari*, ognuno della forma

```
a: b c ...
TAB α
TAB β
...
```

in cui a, b, c, ... sono nomi qualsiasi (immaginare che siano nomi di files, anche se non è necessario che lo siano) e α, β, ... comandi della shell con alcune regole speciali per il trattamento delle abbreviazioni. a si chiama il *controllo primario* o *bersaglio* (*target*) del blocco, b, c, ... i *prerequisiti*. Un prerequisito si chiama *controllo secondario* se è a sua volta controllo primario di un altro blocco. I prerequisiti possono anche mancare.

Verificare il bersaglio a comprende adesso ricorsivamente le seguenti operazioni:

(1) Vengono verificati tutti i controlli secondari del blocco che inizia con a.

(2) Dopo la verifica dei controlli secondari vengono eseguiti i comandi α, β, ... del blocco salvo nel caso che il controllo sia già stato verificato oppure sia soddisfatta la seguente condizione:

`a` *è il nome di un file esistente (nella stessa directory e nel momento in cui viene effettuata la verifica) e anche i prerequisiti* `b`, `c`, ... *sono nomi di files esistenti nessuno dei quali è più recente (tenendo conto della data dell'ultima modifica) del controllo primario.*

Può essere che un file venga creato mediante i comandi (come avviene ad esempio nella compilazione); l'esistenza viene però controllata nel momento della verifica.

Per capire il funzionamento di **make** creiamo un file **Makefile** così composto:

```
# Prove per capire il makefile
.SILENT:

primobersaglio : a b c
TAB echo io primobersaglio

a:
TAB echo io a

b: d e
TAB echo io b

c: e f
TAB echo io c

d:
TAB echo io d

e:
TAB echo io e

f:
TAB echo io f
```

All'inizio assumiamo che nessuno dei controlli corrisponda a un file esistente nella directory. Dare dalla shell il comando **make** e vedere cosa succede. Creare poi files con alcuni dei nomi **a**, **b**, ... con **touch** oppure eliminare alcuni dei files già creati, ogni volta invocando il **make**. Variare l'esperimento modificando il makefile.

for

Il *for* nel C (cfr. pagina 21 per il Perl) ha la seguente forma:

for(α;A;β) γ;

equivalente a

α;
ciclo: if (A) {γ; β; goto ciclo;}

α e β sono successioni di istruzioni separate da virgole; l'ordine in cui le istruzioni in ogni successione vengono eseguite non è prevedibile. γ è un'istruzione o un blocco di istruzioni (cioè una successione di istruzioni separate da punti e virgola racchiusa tra parentesi graffe). Ciascuno di questi campi può anche essere vuoto.

Da un *for* si esce con *break*, mentre *continue* fa in modo che si torni ad eseguire il prossimo passaggio del ciclo, dopo esecuzione di β. Quindi

for (;A;β) {γ1; if (B) break; γ2;}

è equivalente a

ciclo: if (A)
{γ1; if (B) goto fuori; γ2; β; goto ciclo;}
fuori:

mentre

for (;;β) {γ1; if (B) continue; γ2;}

è equivalente a

ciclo: γ1; if (!B) γ2; β; goto ciclo;

Analizzare bene il significato di questa riga. Il punto esclamativo in C denota la negazione.

Operatori logici

In C la definizione di *vero* e *falso* è più semplice che in Perl: ogni espressione booleana viene convertita in un numero, solo il numero 0 è falso, ogni numero diverso da zero è vero.

Per la congiunzione logica (AND) in C, come nel Perl (rileggere pagina 21 per l'antisimmetria di questi operatori), viene usato l'operatore `&&`, per la disgiunzione (OR) l'operatore ||.

La ragione perché i simboli scelti sono doppi è che quando il C fu inventato le memorie erano piccole e costose ed erano ancora molto usati gli operatori logici *bit per bit* (ad esempio per *flags*) per i quali vennero previsti i simboli `&` e | che esistono ancora oggi ma sono usati solo raramente. Ad esempio $25 = (11001)_2$ e $13 = (01101)_2$ e se effettuiamo un AND bit per bit vediamo che *25&13* è uguale a $(01001)_2 = 9$ (anche in Perl). Spiegare da soli perché *25|13 = 29*.

Abbiamo già visto che il punto esclamativo viene usato per la negazione logica. Se `A` è un'espressione vera (cioe diversa da 0), allora `!A` è falso, cioè 0, e viceversa. In altre parole `!A` è equivalente a `A==0`.

I numeri complessi

Un *numero complesso* è un punto $z = (x, y)$ del piano reale. Secondo questa definizione, i numeri complessi non sono nuovi *come oggetti*. Definiamo però adesso due operazioni, *addizione* e *moltiplicazione*, per i numeri complessi.

Siano $z = (x, y)$ e $c = (a, b)$. Allora

$$\boxed{\begin{aligned} c + z &:= (a + x, b + y) \\ cz &:= (ax - by, ay + bx) \end{aligned}}$$

L'addizione è l'addizione vettoriale nel piano, la moltiplicazione è invece motivata nel modo seguente. L'equazione $x^2 = -1$ non ha soluzioni reali (perché $x \in \mathbb{R}$ implica $x^2 \geq 0$ e $1 + x^2 \geq 1 > 0$). È possibile aggiungere ai numeri reali altri numeri, *numeri immaginari*, tra cui un numero i (i appunto perché immaginario) che soddisfa l'equazione $i^2 = -1$? Naturalmente si vorrebbe che le usuali leggi aritmetiche siano conservate anche con i nuovi numeri. Con la nostra definizione tutto funziona bene:

Chiamiamo il punto $(0, 1)$ del piano i, poniamo cioè $i := (0, 1)$, e identifichiamo il numero reale a con il numero complesso $(a, 0)$ - ciò significa geometricamente che consideriamo la retta reale come sottoinsieme del piano nel solito modo, identificandola con l'asse delle x.

Siano $a, b, x, y \in \mathbb{R}$. Allora:

(1) $(a, 0) \cdot (x, y) = (ax, ay)$, dove a sinistra il prodotto è il nuovo prodotto per i numeri complessi. Infatti, secondo la nostra definizione,

$$\begin{aligned} &(a, 0) \cdot (x, y) = \\ &\quad = (ax - 0 \cdot y, 0 \cdot x + ay) = \\ &\quad = (ax, ay) \end{aligned}$$

Ciò significa che $(a, 0)(x, y)$ è uguale a $a \cdot (x, y)$, cioè al prodotto dello scalare reale a con il vettore (x, y) del piano.

(2) $\boxed{i^2 = -1}$.

Infatti

$$\begin{aligned} i^2 &= (0, 1) \cdot (0, 1) = \\ &= (0 \cdot 0 - 1 \cdot 1, 1 \cdot 0 + 0 \cdot 1) = \\ &= (-1, 0) = -1 \end{aligned}$$

(3) Più in generale abbiamo

$$\begin{aligned} (a, b) &= (a, 0) + (0, b) \\ &= a + b(0, 1) = a + bi \end{aligned}$$

e quindi anche $(x, y) = x + yi$, e

$$\begin{aligned} &(a + bi)(x + yi) = \\ &\quad = ax + ayi + bxi + byi^2 = \\ &\quad = ax - by + (ay + bx)i \end{aligned}$$

Osserviamo bene quest'ultima formula. Il risultato è in accordo con la nostra definizione per la moltiplicazione per numeri complessi, ma è stato raggiunto eseguendo il calcolo secondo le regole algebriche usuali, a cui abbiamo aggiunto la nuova legge $i^2 = -1$.

(4) E infatti per le operazioni + e − definite all'inizio valgono le stesse leggi algebriche come per i numeri reali, perché, come si verifica adesso facilmente, l'insieme dei numeri complessi con queste operazioni è un anello commutativo in cui il numero reale $1 = (1, 0)$ è l'elemento neutro per la moltiplicazione. Quest'ultima affermazione segue dal punto (1). Vedremo (a pagina 42) che i numeri complessi formano in verità un *campo*, cioè che ogni numero complesso diverso da zero possiede un inverso per la moltiplicazione.

Il campo dei numeri complessi viene denotato con $\mathbb{C}$. In pratica $\mathbb{C} = \mathbb{R}^2$, però con le nuove operazioni. Abbiamo già visto che $(a, b) = a + bi$, e siccome $bi = ib$, si può anche scrivere $(a, b) = a + ib$.

La formula di Euler

Abbiamo introdotto già a pagina 37 la notazione

$$e^{i\alpha} := (\cos\alpha, \sin\alpha)$$

per $\alpha \in \mathbb{R}$. Adesso la possiamo riscrivere nella forma

$$e^{i\alpha} = \cos\alpha + i \sin\alpha$$

(formula di Euler). Questa notazione, per il momento puramente simbolica, non è solo molto comoda, come vedremo adesso, ma anche estremamente importante nella teoria.

Dal teorema di addizione per le funzioni trigonometriche si deduce immediatamente che

$$\boxed{e^{i(\alpha+\beta)} = e^{i\alpha} \cdot e^{i\beta}}$$

per $\alpha, \beta \in \mathbb{R}$ (esercizio 65). Si vede che nel campo complesso il teorema di addizione assume una forma molto più semplice.

Per $z = x + iy$ possiamo anche più in generale definire

$$\boxed{e^z := e^x e^{iy}}$$

Si dimostra (esercizio 72) che $e^{z+w} = e^z \cdot e^w$ per ogni $z, w \in \mathbb{C}$. Ciò significa che la funzione esponenziale è un omomorfismo

$$(\mathbb{C}, +) \longrightarrow (\mathbb{C}, \cdot).$$

Il campo dei numeri complessi

Abbiamo visto a pagina 37 che ogni punto $z = (x, y)$ di $\mathbb{R}^2$ può essere scritto nella forma

$$z = (r\cos\alpha, r\sin\alpha)$$

dove $r \geq 0$ è univocamente determinato ed $\alpha \in \mathbb{R}$. Abbiamo anche visto che $r = |z| = \sqrt{x^2 + y^2}$.

Se $z \neq 0$, anche α è univocamente determinato se chiediamo $0 \leq \alpha < 2\pi$ oppure, come il matematico preferisce dire, univocamente determinato modulo 2π.

Con la formula di Euler possiamo quindi scrivere ogni numero complesso z nella forma

$$z = re^{i\alpha}$$

anch'essa già anticipata a pagina 37, con r ed α come sopra.

Sia adesso anche $\beta \in \mathbb{R}$. Allora

$$e^{i\beta}z = re^{i(\alpha+\beta)}$$

come segue dal teorema di addizione visto a pagina 41. Il prodotto $e^{i\beta}z$ ha quindi la stessa lunghezza di z ed un angolo aumentato di β rispetto a z. Ciò significa che la moltiplicazione con $e^{i\beta}$ è la stessa cosa come una *rotazione* per l'angolo β in senso antiorario.

Prendiamo adesso un numero complesso c arbitrario. Lo possiamo rappresentare nella forma $c = se^{i\beta}$ con $s \geq 0$. Per il prodotto cz otteniamo evidentemente $cz = sre^{i(\alpha+\beta)}$ e quindi vediamo che la moltiplicazione con un numero complesso consiste sempre di una rotazione combinata con un allungamento (o accorciamento se $|c| < 1$).

Anche nel seguito siano sempre $x, y \in \mathbb{R}$. Sia $z = x + iy$. Allora $\overline{z} := x - iy$ si chiama il numero complesso *coniugato* a z. Geometricamente $\overline{z}$ si ottiene mediante riflessione di z rispetto all'asse reale. È chiaro che $\overline{\overline{z}} = z$.

Lemma 1: Sia $z = x + iy$. Allora

$$z\overline{z} = x^2 + y^2 = |z|^2.$$

Dimostrazione: Esercizio 67.

Osservazione 1: Ogni numero complesso $z \neq 0$ possiede un inverso rispetto alla moltiplicazione, infatti

$$z \cdot \frac{\overline{z}}{z\overline{z}} = \frac{z\overline{z}}{z\overline{z}} = 1$$

per cui possiamo porre

$$\frac{1}{z} = \frac{\overline{z}}{z\overline{z}}$$

o, equivalentemente,

$$\frac{1}{x+iy} = \frac{x-iy}{x^2+y^2}$$

$(\mathbb{C}, +, \cdot)$ è quindi un campo. Come in ogni campo anche in $\mathbb{C}$ l'inverso è univocamente determinato.

Definizione: $z = x+iy$ (con $x, y \in \mathbb{R}$ come sempre) sia un numero complesso. Definiamo allora

$$\text{Re } z := x$$
$$\text{Im } z := y$$

Re z si chiama la *parte reale* di z, Im z la *parte immaginaria*.

Osservazione 2: z sia un numero complesso. Allora

$$|\text{Re } z| \leq |z| \quad \text{e} \quad |\text{Im } z| \leq |z|.$$

Dimostrazione: Infatti

$$|\text{Re } z| = |x| = \sqrt{x^2} \leq \sqrt{x^2 + y^2} = |z|.$$

Nello stesso modo per Im z.

La disuguaglianza di Cauchy-Schwarz

Teorema: Siano $x = (x_1, \ldots, x_n)$ e $y = (y_1, \ldots, y_n)$ due punti di $\mathbb{R}^n$. Allora

$$|(x, y)| \leq |x||y|$$

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 (pagina 11):

$$(x, y) = |x||y|\cos\alpha$$

e siccome $|\cos\alpha| \leq 1$ abbiamo

$$|(x, y)| = |x||y||\cos\alpha| \leq |x||y|$$

Lemma 2: Siano $v = (v_1, v_2)$ in $\mathbb{R}^2$ e $a = (a_1, a_2) = (-v_2, v_1)$. Allora per ogni $w \in \mathbb{R}^2$ vale

$$|(v, w)|^2 + |(a, w)|^2 = |v|^2|w|^2$$

Dimostrazione: Utilizziamo la figura a pagina 31. Abbiamo

$$|(v, w)| = |v||w||\cos\alpha|$$

$$\begin{aligned} |(a, w)| &= |a||w||\cos\beta| = \\ &= |a||w||\sin\alpha| = \\ &= |v||w||\sin\alpha| \end{aligned}$$

per cui

$$\begin{aligned} &|(v, w)|^2 + |(a, w)|^2 = \\ &= |v|^2|w|^2(\cos^2\alpha + \sin^2\alpha) = \\ &= |v|^2|w|^2 \end{aligned}$$

La disuguaglianza triangolare

Proposizione: Siano $x = (x_1, \ldots, x_n)$ e $y = (y_1, \ldots, y_n)$ due punti di $\mathbb{R}^n$. Allora

$$|x + y| \leq |x| + |y|$$

Dimostrazione: Ciò è una facile conseguenza della formula

$$|x + y|^2 = |x|^2 + |y|^2 + 2(x, y)$$

(pagina 11, sostituendo y con $-y$, cfr. esercizio 12) 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

$$|x + y| \leq |x| + |y|$$

La formula di de Moivre

Sia $z = re^{i\alpha} \in \mathbb{C}$. Per $n \geq 1$ allora, secondo le formule viste precedentemente, abbiamo

$$z^n = r^n e^{in\alpha}$$

Abraham de Moivre (1667-1754) era un matematico francese emigrato giovane in Inghilterra. Ha scritto un famoso trattato sul calcolo delle probabilità (*Doctrines of chances*, 1718).

Leonhard Euler (1707-1783), svizzero, è stato uno dei più prolifici matematici di tutti i tempi. Ha lavorato su quasi tutti i campi della matematica pura e applicata del suo tempo.

Linux Day al dipartimento di Matematica

Sabato 1 dicembre 2001 nell'aula magna del dipartimento di Matematica con inizio alle ore 9.30.

La giornata sarà organizzata in due momenti. La mattinata sarà dedicata alla presentazione del sistema GNU/Linux e del software libero con interventi di imprenditori locali, rappresentanti della amministrazione locale e della scuola/università. Seguirà un dibattito sui temi della manifestazione.

La seconda parte della giornata, nel pomeriggio, sarà dedicata a tutti coloro che vogliono avvicinarsi al mondo GNU/Linux. Sarà possibile recarsi alla manifestazione con il proprio PC ed essere seguiti nell'installazione del sistema operativo da volontari del flug. Alcuni soci saranno a disposizione per rispondere alle domande dei partecipanti e per aiutarli nella risoluzione dei problemi di installazione/configurazione del sistema, oppure semplicemente per fare due chiacchiere e scambiarsi opinioni.

Il programma dettagliato delle attività e degli interventi sarà disponibile sul sito Internet (*www.ferrara.linux.it/*) del *Ferrara Linux Users Group* nei prossimi giorni.

ALGORITMI E STRUTTURE DI DATI

Corso di laurea in matematica Anno accademico 2001/2002 Numero 10 ◇ 26 Novembre 2001

Vettori e puntatori

*Un **vettore** è un indirizzo fisso insieme a un tipo, un **puntatore** è un indirizzo variabile insieme a un tipo.*

La grandezza (size) di un vettore deve essere indicata all'atto della dichiarazione (tranne nel caso che il vettore sia argomento di una funzione) in modo diretto (ad esempio con `int a[10];` *per dichiarare un vettore con spazio per 10 interi) o indiretto (ad esempio con* `int a[]=0,1,2,3,4;` *per dichiarare un vettore con spazio per 5 interi i cui componenti iniziali siano 0,...4). Successivamente si possono modificare i valori di questi componenti, ad esempio con* `a[3]=7;`*, però soltanto all'interno dello spazio previsto, quindi in questo caso solo* `a[0], ..., a[4]`*, ma non* `a[5]`*. Vettori possono essere anche a più dimensioni, ad esempio* `int a[3][4];`*.*

Puntatori vengono usati in due modi. Da un lato possono essere utilizzati come indirizzi variabili per muoversi all'interno della memoria. Dall'altro vengono usati in modo simile ai vettori per destinare delle aree di memoria in cui conservare dei dati. In tal caso bisogna riservare quest'area di memoria con una delle istruzioni di allocazione di memoria che impareremo più avanti oppure con l'assegnazione diretta di una stringa. Per distinguere puntatori da vettori useremo iniziali maiuscole per i puntatori e in genere minuscole per i vettori; è solo un'abitudine che proponiamo, non è necessario.

Se `X` *è un puntatore a un intero,* `*X` *è l'intero che risiede all'indirizzo* `X`*. Per questa ragione un puntatore a un intero viene dichiarato in questo modo:* `int *X;`*. I puntatori sono variabili come le altre, e quindi possono essere modificati. Ad esempio con* `int a[]=1,3,8,7,*X; X=a+2;` *si definiscono un vettore* `a` *di interi e un puntatore* `X` *a interi che successivamente punta a* `a[2]`*, cioè in questo caso a 8 (gli indici iniziano con 0). Non sono invece permesse le istruzioni* `a=a+2;` *oppure* `a=X;`*. A parte questo, molte operazioni sono uguali per puntatori e vettori. Ad esempio dopo* `int *X,a[10]; X=a;` *le istruzioni* `a[4]=7;` *e* `X[4]=7;` *sono equivalenti. Un significato analogo ha* `*X` *se* `X` *è un puntatore a elementi di un altro tipo.*

Se `w` *è una variabile di tipo t,* `&w` *è il puntatore a* `w` *(naturalmente dello stesso tipo t). Perciò* `*&w` *è il valore di* `w`*; ad esempio dopo la dichiarazione* `int n;` *le istruzioni* `n=7;` *e* `*&n=7;` *hanno lo stesso effetto.*

Stringhe e caratteri

Una stringa in C è una successione di caratteri terminata dal carattere con codice ASCII 0. Caratteri vanno racchiusi tra apostrofi, stringhe tra virgolette.

L'istruzione *char *X="Alberto";* è permessa; viene cercato prima uno spazio di 8 byte adiacenti che viene riempito con i caratteri *'A', 'l', ..., 'o', 0*, dopodiché *X* diventa l'indirizzo del primo byte della stringa. Lo spazio riservato per *X* è di esattamente 8 byte; quindi è possibile trasformare la stringa in *"Roberto"* mediante le istruzioni *X[0]='R'; X[1]='o';*, mentre *X[20]='t';* in genere causerà un errore perché si scrive su uno spazio non più riservato.

Per inizializzare una stringa si possono anche usare vettori: *char a[]="Alberto";* (ancora 8 byte) oppure *char a[200]="Alberto";*. In quest'ultimo caso nell'indirizzo *a* sono riservati 200 byte; *a[7]* è occupato dal carattere 0 e designa per il momento la fine della stringa, ma successivamente si può scrivere in tutti i 200 byte riservati.

Alcuni caratteri speciali: *'\n'* è il carattere di nuova riga, *'\t'* il tabulatore, *'\\'* il backslash, *'\"'* una virgoletta semplice, *'\''* un apostrofo, *'\0'* il carattere ASCII 0; *"\""* è una stringa il cui unico carattere è una virgoletta. Il codice ASCII dello spazio è 32, quello del carattere *escape* 27.

Questa settimana

43 Vettori e puntatori
Stringhe e caratteri
Aritmetica dei puntatori
Operatori abbreviati

44 Confronto di stringhe
if ... else
Puntatori generici
Conversioni di tipo

45 Parametri di main
Input da tastiera
Passaggio di parametri

46 Variabili di classe static
Lisp
Il λ-calcolo
Prolog
Gli altri linguaggi

47 Le strutture del C
Valore assoluto e quoziente di numeri complessi
Estensioni di campi
Le equazioni di Cauchy-Riemann

Aritmetica dei puntatori

Puntatori variabili vengono spesso usati in cicli, per esempio

*for(X=a;X-a<4;X++) printf("%d ",*X);*

Se *X* è un puntatore (o vettore) di tipo t e *n* un'espressione di tipo intero (anche lungo), allora *X+n* è il puntatore dello stesso tipo t e indirizzo uguale a quello di *X* aumentato di *nd*, dove *d* è lo spazio che occupa un elemento del tipo t; quindi se immaginiamo la parte di memoria che parte nell'indirizzo corrispondente a *X* occupata da elementi di tipo t, *X+n* punta all'elemento con indice *n* (cominciando a contare da zero). In altre parole, **(X+n)* è lo stesso elemento come *X[n]*.

Puntatori possono essere confrontati tra di loro (hanno senso quindi le espressioni *X¡Y* oppure *X==Y* per due puntatori *X* e *Y*); la sottrazione di puntatori dà un intero (cfr. sopra); si possono usare le istruzioni *X++* e *X–* come per variabili intere.

***

Operatori abbreviati

Invece di *a=a+b;* si può anche scrivere *a+=b;*, e similmente si possono abbreviare assegnazioni per gli altri operatori binari, quindi si userà *a*=b;* per *a=a*b;*, *a/=b;* per *a=a/b;* e *a-=b;* per *a=a-b;*. È una buona notazione e comoda, infatti *resistenza/=2;* e più breve e più leggibile di *resistenza=resistenza/2;*.

Confronto di stringhe

Definiamo una funzione **us** per l'uguaglianza di stringhe nel modo seguente:

```
int us (char *A, char *B)
{for (;*A;A++,B++) if (*A!=*B) return 0;
return (*B==0);}
```

La condizione nel `for` è che `*A` non sia zero; questo avviene se e solo se il carattere nella posizione a cui punta `A` (che varia durante il `for`) non è il carattere 0 (che, come abbiamo detto, viene usato per indicare la fine di una stringa). Quindi l'algoritmo percorre la prima stringa fino alla sua fine e confronta ogni volta il carattere nella prima stringa con il carattere nella posizione corrispondente della seconda. Quando trova la fine della prima, controlla ancora se anche la seconda termina.

Si noti che per percorrere le due stringhe usiamo le stesse variabili `A` e `B` che all'inizio denotano gli indirizzi delle stringhe. Che questo non cambia verso l'esterno i valori di `A` e `B` è una peculiarità del C che verrà spiegata fra poco.

Adesso `us(A,B)` restituisce il valore 1, se le due stringhe `A` e `B` sono uguali, altrimenti 0. Per provarla possiamo usare la funzione

```
static void provestringhe()
{printf("%d %d %d\n",us("alfa","alfa"),
us("alfa","alfabeto"),us("alfa","beta"));}
```

In verità per il confronto di stringhe conviene usare la funzione **strcmp** del C, che tratteremo però soltanto molto più avanti quando parleremo delle *librerie standard* del C:

```
int us (char *A, char *B)
{return (strcmp(A,B)==0);}
```

Si vede qui che un'espressione booleana in C è un numero (uguale a 0 o 1) che può essere risultato di una funzione.

Attenzione: Per l'uguaglianza di stringhe non si può usare `(A==B)`, perché riguarderebbe l'uguaglianza degli indirizzi in cui si trovano le due stringhe, tutt'altra cosa quindi. Provare a descrivere la differenza!

Definiamo adesso una funzione **uis** (uguaglianza inizio stringhe) di due stringhe che restituisce 1 o 0 a seconda che la prima stringa è sottostringa della seconda o no.

```
int uis (char *A, char *B)
{for (;*A;A++,B++) if (*A!=*B) return 0;
return 1;}
```

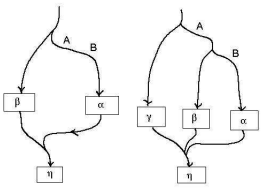
La differenza tra **uis** e **us** non è grande. In cosa consiste? Giustificare l'algoritmo. Anche qui potremmo usare le funzioni della libreria standard:

```
int uis (char *A, char *B)
{return (strncmp(A,B,strlen(A))==0);}
```

---

if ... else

if (A) α; η;
if (A) α; else β; η;
if (A) if (B) α; else β; η; (*)
if (A) {if (B) α;} else β; η; (**)
if (A) if (B) α; else β; else γ; η;
if (A) α; else if (B) β; else γ; η;
if (A) α; else if (B) β; else if (C) γ; η;
if (A) α; else if (B) β; else if (C) γ; else δ; η;
if (A) if (B) α; else β; else if (C) γ; else δ; η;



Studiare con attenzione queste espressioni e descrivere ciascuna di esse mediante un diagramma di flusso; a quali righe corrispondono i due diagrammi di flusso a destra? α, β, ... sono istruzioni oppure blocchi di istruzioni (successioni di istruzioni separate da punto e virgola e racchiuse da parentesi graffe). Si vede che talvolta bisogna usare addizionali parentesi graffe per accoppiare **if** ed **else** nel modo desiderato: qual'è la differenza tra (*) e (**)? Si noti che (**) potrebbe essere scritto anche così: *if (A&&B) α; else β; η;*

Puntatori generici

Talvolta il programmatore avrebbe bisogno di strutture e operazioni che funzionino con elementi di tipo qualsiasi. Allora si possono usare *puntatori generici*, che formalmente vengono dichiarati come `void *`. Un esempio:

```
void applica (void (*f)(), void *X)
{f(X);}
void scrivi (int *X)
{printf("%d\n",*X);}
void aumenta (int *X)
{(*X)++;}
```

Adesso con

```
int a=8; applica(aumenta,&a);
applica(scrivi,&a);
```

otteniamo l'output 9. Si osservi il modo in cui una funzione viene dichiarata come argomento di un'altra funzione. Qui bisogna menzionare una differenza fra il C e il C++. In C una dichiarazione della forma `t f();` (dove t è il tipo del risultato) significa che in fase di dichiarazione non vengono fissati il numero e il tipo degli argomenti di f; in C++ invece questa forma indica che f non ha argomenti. Una funzione senza argomenti in C invece viene dichiarata con `t f(void);`.

Conversioni di tipo

Puntatori di tipo diverso possono essere convertiti tra di loro. Se `X` è un puntatore di tipo t, allora `(s) X` è il puntatore con lo stesso indirizzo di `X`, ma di tipo s. Ad esempio `X+2` punta all'elemento di tipo t con indice 2 a partire dall'indirizzo corrispondente ad `X`, ma `(char *)X+2` punta al secondo byte a partire da quell'indirizzo. Qual'è invece il significato di `(char *)(X+2)`?

Conversioni di tipo fra puntatori sono frequenti soprattutto quando si utilizzano *puntatori generici* (indirizzi puri, cfr. sopra). Dopo `void *A; int *B;` con `B=(int *)A;` il puntatore `B` di tipo **int** viene a puntare sull'indirizzo corrispondente ad `A`.

In alcuni casi sono possibili anche conversioni di tipo fra variabili normali, ma in genere è preferibile usare funzioni apposite (ad esempio per ottenere la parte intera di un numero reale).

Nelle operazioni di input si usano spesso le funzioni **atoi**, **atol** e **atof** che convertono una stringa in un numero (risp. di tipo **int**, **long** e **double**). Bisogna includere il header <**stdlib.h**>.

```
int n; double x;
n=atoi("3452"); x=atof("345.200");
```

Abbiamo usato questa conversione (che in Perl è automatica) in alcuni esempi.

Parametri di main

Ogni progetto deve contenere esattamente una funzione **main**, che sarà la prima funzione ad essere eseguita. Essa viena usata nella forma `int main()` (più precisamente nella forma *int main(void)*) oppure nella forma `int main(int na, char **va)`, se deve essere chiamata dalla shell. In questa seconda forma *na* è il numero degli argomenti più uno (perché viene anche contato il nome del programma), *va* il vettore degli argomenti, un vettore di stringhe in cui la prima componente *va[0]* è il nome del programma stesso, mentre *va[1]* è il primo argomento, *va[2]* il secondo, ecc. I nomi per le due variabili possono essere scelti dal programmatore, in inglese si usano spesso *argc* (*argument counter*) per *na* e *argv* (*argument vector*) per *va*. Facciamo un esempio:

```
// alfa.c
# include "alfa.h"
int main();
////////////////////
int main (int na, char **va)
{int n;
if (na==1) fattoriali(); else
{n=atoi(va[1]); printf("%d! = %-12.0f\n",
     n,fattoriale(n));}
exit(0);}
```

Notiamo in primo luogo che nella dichiarazione di **main** non abbiamo indicato gli argomenti. Ciò è possibile in C; se volessimo usare questa funzione anche in C++, dovremmo, nella dichiarazione (in cui comunque è sufficiente indicare il tipo, non necessariamente il nome degli argomenti) scrivere `int main(int,char**);`.

Il *test di uguaglianza* avviene mediante l'operatore ==; bisogna stare attenti a non confondere questo operatore con l'operatore di assegnazione =. Se scrivessimo infatti *if (na=1)*, verrebbe prima assegnato il valore 1 alla variabile *na*, la quale quindi, avendo un valore diverso da zero, sarebbe considerata vera, per cui verrebbe sempre eseguito la prima alternativa dell'*if*.

Abbiamo qui definito una nuova funzione

```
void fattoriali()
{int n;
for (n=0;n<=20;n++)
   printf("%2d! = %-12.0f\n",n,fattoriale(n));}
```

che visualizza ancora i fattoriali da 0! a 20!. Si noti l'uso della funzione **atoi** di cui abbiamo parlato a pagina 44, con cui la stringa immessa dalla shell come argomento (quando presente - e ciò viene rilevato dall'*if*) viene convertita in un numero intero.

Esercizio: Fare in modo che, se dalla shell si chiama **alfa a b**, vengano visualizzati i fattoriali da $a!$ a $b!$.

Input da tastiera

Per l'input di una stringa da tastiera in casi semplici si può usare la funzione **gets**:

```
char a[40];
gets(a);
```

Il compilatore ci avverte però che *the 'gets' function is dangerous and should not be used*. Infatti se l'utente immette più di 40 caratteri (per disattenzione o perché vuole danneggiare il sistema), scriverà su posizioni non riservate della memoria. Nei nostri esperimenti ciò costituirebbe raramente un problema, ma può essere importante in programmi che verranno usati da utenti poco esperti oppure malintenzionati. Si preferisce per questa ragione la funzione **fgets**:

```
char a[40];
fgets(a,38,stdin);
```

In questo caso nell'indirizzo *a* vengono scritti al massimo 38 caratteri; ricordiamo che *stdin* è lo *standard input*, cioè la tastiera (**fgets** può ricevere il suo input anche da altri files). A differenza da **gets** **fgets** inserisce nella stringa anche il carattere '\n' che termina l'input e ciò è un po' scomodo. Definiamo quindi una nostra funzione di input (esaminarla bene):

```
void input (char *A, int n)
{if (n<1) n=1; fgets(A,n+1,stdin);
for (;*A;A++); A--;
if (*A=='\n') *A=0;}
```

Passaggio di parametri

I parametri (argomenti) di una funzione in C vengono sempre passati per valore. Con ciò si intende che in una chiamata *f(x)* alla funzione *f* viene passato solo il valore di *x*, con cui la funzione esegue le operazioni richieste, ma senza che il valore della variabile *x* venga modificato, anche nel caso che all'interno della funzione ci sia un'istruzione del tipo *x=nuovovalore;*. Infatti la variabile *x* che appare all'interno della funzione è un'altra variabile, che riceve come valore iniziale il valore della *x*. Per questa ragione è corretta la funzione

```
int us (char *A, char *B)
{for (;*A;A++,B++)
if (*A!=*B) return 0;
return (*B==0);}
```

che abbiamo introdotto a pag. 44. Se questa funzione viene chiamata con

```
char *A="Giovanni",*B="Giacomo";
if (us(A,B)) ...
```

nel ciclo *for (;*A;A++,B++)* della funzione non sono i due puntatori *A* e *B* che si muovono, ma copie locali create per la funzione. Quindi dopo l'esecuzione della funzione *A* e *B* puntano ancora all'inizio delle due stringhe e non ad esempio ai caratteri *'o'* rispettivamente *'a'* dove le loro versioni locali si sono fermate.

Per la stessa ragione per aumentare il valore di una variabile intera non si può usare la seguente funzione:

```
void aumenta (int x)
{x++;}
```

Se la proviamo con *int x=5; aumenta(x); printf("%d\n",x);* otteniamo l'output 5, perché l'aumento non è stato eseguito sulla variabile *x* ma su una copia interna che all'uscita dalla funzione non esiste più.

Il modo corretto di programmare questa funzione è di passare alla funzione l'indirizzo della *x* (mediante l'uso di un puntatore oppure, in C++, di un riferimento) e di aumentare il contenuto di quell'indirizzo:

```
void aumenta (int *X)
{(*X)++;}
```

Invece di *(*X)++;* si può anche usare **X=*X+1;*, ma non **X++;* che aumenterebbe l'indirizzo *X* (secondo le regole dell'aritmetica dei puntatori, cfr. pag. 43), senza nessuna altro effetto.

Variabili di classe static

Con *int x;* si dichiara una variabile di tipo *int*. Nella dichiarazione l'indicazione del tipo può essere preceduta dalla *classe di memoria* (*storage class*) che deve essere un'espressione tra le seguenti: **static**, **extern**, **register**, **auto** e **typedef**.

Variabili possono essere dichiarate anche al di fuori di una funzione. In questo secondo caso la classe di memoria **static** significa che la variabile non è visibile al di fuori dal file sorgente in cui è contenuta. Ciò naturalmente vale ancor di più per una variabile dichiarata internamente a una funzione. In tal caso l'indicazione della classe **static** ha una conseguenza pecualiare che talvolta è utile, ma che può implicare un comportamente della funzione misterioso, se non si conosce la regola.

Infatti mentre normalmente, se una variabile è interna a una funzione, in ogni chiamata della funzione il sistema cerca di nuovo uno spazio in memoria per questa variabile, alle variabili di classe **static** viene assegnato uno spazio in memoria fisso, che rimane sempre lo stesso in tutte le chiamate della funzione (ciò evidentemente ha il vantaggio di impegnare la memoria molto meno); i valori di queste variabili si conservano da una chiamata all'altra.

Inoltre una eventuale inizializzazione per una variabile **static** viene eseguita solo nella prima chiamata (è soprattutto questo che puo confondere). Esempi:

```
void provastatic1()
{static int s=0,k;
for (k=0;k<5;k++) s+=k; printf("%d\n",s);}
```

La prima volta che utilizziamo questa funzione viene visualizzato 10 (la somma dei numeri 0,1,2,3,4), la seconda volta però 20, la terza volta 30, proprio perché l'inizializzazione *s=0* viene effettuata solo la prima volta. Probabilmente ciò non è quello che qui il programmatore, che forse intendeva soltanto risparmiare memoria utilizzando una variabile **static**, desiderava fare.

È facile rimediare comunque, senza rinunciare a **static**, perché l'istruzione *s=0*, se data al di fuori della dichiarazione, viene eseguita normalmente ogni volta:

```
void provastatic2()
{static int s,k;
for (s=0,k=0;k<5;k++) s+=k;
printf("%d\n",s);}
```

Esercizio: Inventare una situazione in cui può essere utile che una variabile interna di una funzione conservi il suo valore da una chiamata all'altra.

Lisp

Il Lisp, creato alla fine degli anni '50 da John McCarthy, è ancora oggi uno dei più potenti linguaggi di programmazione. Funzioni e liste sono gli elementi di questo linguaggio, e funzioni possono essere sia argomenti che valori di altre funzioni. Per questo in Lisp (e in Perl) si possono scrivere applicazioni non realizzabili in C. Il Lisp è stato dagli inizi il linguaggio preferito dell'intelligenza artificiale. È un linguaggio difficile che non viene in nessun modo incontro al programmatore, e per questo è poco diffuso; sembra però che i programmatori in Lisp guadagnino il doppio dei programmatori in C. Tutto ciò che si può fare in Lisp comunque lo si può fare anche in Perl - entrambi contengono il λ-calcolo e ciò costituisce la differenza fondamentale con altri linguaggi. Ci sono parecchi dialetti del Lisp: il più importante è il *Common Lisp*, considerato la norma del Lisp, mentre lo *Scheme* è una versione minore didattica; in *Elisp*, un dialetto abbastanza simile al Common Lisp, è programmato e programmabile l'editor *Emacs*, uno dei più formidabili strumenti informatici. Un miniprogramma in Lisp:

```
#! /usr/local/bin/clisp
(defun cubo (x) (* x x x))
(format t "~a~%" (cubo 3))
```

Si noti che gli operatori precedono gli operandi. Siccome, a differenza dal PostScript, gli operatori (ad esempio *) non hanno un numero di argomenti fisso, bisogna usare le parentesi. *format* è un'istruzione di output, abbastanza simile al *printf* del C.

Il λ-calcolo

Siccome bisogna distinguere tra la funzione f e i suoi valori $f(x)$, introduciamo la notazione $\bigcirc_x f(x)$ per la funzione che manda x in $f(x)$. Ad esempio $\bigcirc_x \sin(x^2+1)$ è la funzione che manda x in $\sin(x^2+1)$. È chiaro che ad esempio $\bigcirc_x x^2 = \bigcirc_y y^2$ (per la stessa ragione per cui $\sum_{i=0}^{n} a_i = \sum_{j=0}^{n} a_j$), mentre $\bigcirc_x \bigcirc_y xy \neq \bigcirc_y \bigcirc_x xy$ (così come $\sum_i a_{ij} \neq \sum_j a_{jj}$) e, come non ha senso l'espressione $\sum_i \sum_i a_{ii}$, così non ha senso $\bigcirc_x \bigcirc_x x$. Siccome in logica si scrive $\lambda x.f(x)$ invece di $\bigcirc_x f(x)$, la teoria molto complicata di questo operatore si chiama λ-calcolo. Su esso si basano il Lisp e molti concetti dell'intelligenza artificiale.

Prolog

Durante gli anni '70 e i primi anni '80 il Prolog divenne popolare in Europa e in Giappone per applicazioni nell'ambito dell'intelligenza artificiale. È un linguaggio dichiarativo, negli intenti della programmazione logica il programmatore deve solo descrivere il problema e indicare le regole di base della soluzione; ci pensa il sistema Prolog a trovare la soluzione. Purtroppo la cosa non è così facile e non è semplice programmare in Prolog. Un esempio in GNU-Prolog:

```
% alfa.pl

:-initialization(q).

padre(giovanni,maria).
padre(federico,alfonso).
padre(alfonso,elena).
madre(maria,elena).

genitore(A,B):- padre(A,B); madre(A,B).
nonno(A,B):- padre(A,X), genitore(X,B).

trovanonni:- setof(X,nonno(X,elena),A),
   write(A), nl.

q:- trovanonni.
% output: [federico,giovanni]
```

Gli altri linguaggi

Algol e **PL/1** erano famosi linguaggi procedurali degli anni '60. Anche il **Cobol** è un linguaggio antico, ancora oggi utilizzato in ambienti commerciali, ad esempio nelle banche. **RPG** è un linguaggio per la creazione di tabulati commerciali ancora usato.

Apl è un linguaggio vettoriale interpretato che richiede tasti speciali. **Ada** doveva diventare un gigantesco linguaggio universale soprattutto per l'utilizzo negli ambienti militari. **Modula-2** era una continuazione del **Pascal**. **Snobol**, **Simula**, **Smalltalk** e **Eiffel** sono linguaggi piuttosto accademici per l'elaborazione dei testi e la programmazione orientata agli oggetti. **Tcl/Tk** è una combinazione di linguaggi usata per creare interfacce grafiche in linguaggi come Perl e Python.

Maple, **Mathematica** e **Matlab** sono usati da matematici e ingegneri nella matematica computazionale.

Il **Ruby** è un linguaggio ancora nel nascere che in pratica dovrebbe diventare, nelle intenzione del suo creatore, un Perl più semplice e più leggibile. **HyperTalk** era un linguaggio quasi naturale per Macintosh e PC, purtroppo praticamente scomparso per ragioni commerciali.

Alla programmazione funzionale sono dedicati vari linguaggi ancora poco diffusi come **Haskell** e **ML**.

Le strutture del C

Definiamo un tipo di dati con due componenti di tipo **double** nel modo seguente:

```
typdef struct double x,y; unzeta;
```

Dopo aver inserito questa definizione di tipo in **alfa.h** possiamo dichiarare un elemento z del tipo *unzeta* con *unzeta z;*. I due componenti di z sono $z.x$ e $z.y$. Possiamo così creare funzioni per l'addizione e la moltiplicazione di numeri complessi:

```
unzeta add (unzeta z1, unzeta z2)
{unzeta w;
w.x=z1.x+z2.x; w.y=z1.y+z2.y; return w;}
unzeta molt (unzeta z1, unzeta z2)
{unzeta w;
w.x=z1.x*z2.x-z1.y*z2.y; w.y=z1.x*z2.y+z2.x*z1.y;
return w;}
```

Il valore assoluto di un numero complesso

Nella divisione e nel calcolo del valore assoluto o della radice quadrata di un numero complesso dobbiamo invece evitare la formazione di risultati intermedi troppo grandi (che vengono approssimati male al calcolatore). Consideriamo prima il valore assoluto di un numero complesso $z = x + iy$.

Intuitivamente $|z| = \sqrt{x^2 + y^2}$, ma il risultato intermedio $x^2 + y^2$ diventa molto più grande del risultato finale. Si usano quindi le formule

$$|z| = |x|\sqrt{1 + (\tfrac{y}{x})^2} \quad \text{(usata per } |y| \le |x| \neq 0\text{)}$$

$$|z| = |y|\sqrt{(\tfrac{x}{y})^2 + 1} \quad \text{(usata per } |x| \le |y| \neq 0\text{)}$$

che portano alla seguente funzione:

```
double vass (unzeta z)
{double vassx,vassy,t;
if (z.x==0) return fabs(z.y);
if (z.y==0) return fabs(z.x);
vassx=fabs(z.x); vassy=fabs(z.y);
if (vassx>=vassy)
{t=z.y/z.x; return vassx*sqrt(1+t*t);}
t=z.x/z.y; return vassy*sqrt(1+t*t);}
```

La funzione *fabs* del C calcola il valore assoluto di un numero di tipo *double*.

Il quoziente di numeri complessi

Per il quoziente

$$\frac{x+iy}{x'+iy'} = \frac{(x+iy)(x'-iy')}{x'^2+y'^2} = \frac{xx'+yy'+i(yx'-xy')}{x'^2+y'^2}$$

(se $x'^2 + y'^2 \neq 0$) si procede in modo analogo. L'ultima frazione può essere scritta come

$$\frac{x+y\frac{y'}{x'}+i(y-x\frac{y'}{x'})}{x'+y'\frac{y'}{x'}} \quad \text{(usata per } |y'| \le |x'| \neq 0\text{)}$$

$$\frac{x\frac{x'}{y'}+y+i(y\frac{x'}{y'}-x)}{x'\frac{x'}{y'}+y'} \quad \text{(usata per } |x'| \le |y'| \neq 0\text{)}$$

Possiamo quindi programmare la divisione così:

```
unzeta divc (unzeta z1, unzeta z2)
{double q,t; unzeta w;
if (fabs(z2.x)>=fabs(z2.y))
{q=z2.y/z2.x; t=z2.x+z2.y*q;
w.x=(z1.x+z1.y*q)/t; w.y=(z1.y-z1.x*q)/t; return w;}
q=z2.x/z2.y; t=z2.x*q+z2.y;
w.x=(z1.x*q+z1.y)/t; w.y=(z1.y*q-z1.x)/t; return w;}
```

Estensioni di campi

Se E è un campo e K un sottocampo di E, E si chiama un'*estensione* di K. La coppia (E, K) in tal caso viene detta un'estensione di campi e denotata con $E : K$. E è allora in modo naturale uno spazio vettoriale su K (il prodotto $c\alpha$ di $c \in K$ con $\alpha \in E$ è semplicemente il prodotto in E a cui anche c appartiene). Questa osservazione ha moltissime applicazioni.

Nella teoria dei campi (e nelle sue applicazioni, ad esempio nella teoria dei numeri) è molto importante la questione quali sono i *campi intermedi* di un'estensione.

Ad esempio quali sono i campi intermedi di $\mathbb{C} : \mathbb{R}$?

La dimensione di $\mathbb{C}$ su $\mathbb{R}$ è 2, quindi ogni sottospazio vettoriale reale L di $\mathbb{C}$ che non coincide con $\mathbb{C}$ deve essere una retta passante per l'origine in $\mathbb{R}^2$. Però se L è un campo intermedio, deve contenere $\mathbb{R}$ e ciò implica $L = \mathbb{R}$.

Può un'altra retta L in $\mathbb{R}^2$ (non necessariamente contenente $\mathbb{R}$ e quindi non necessariamente un campo intermedio di $\mathbb{C} : \mathbb{R}$) essere un campo? L deve essere della forma $L = \mathbb{R}z$ con $z = a + ib \neq 0$, dove $a, b \in \mathbb{R}$. Siccome L è un sottocampo di $\mathbb{C}$, anche $\frac{1}{z} = \frac{a-ib}{a^2+b^2} \in L$ e quindi anche $a - ib \in L$. Se facciamo un disegno, vediamo subito che ciò è possibile solo se $b = 0$ (in tal caso $L = \mathbb{R}$) oppure se $a = 0$. In questo secondo caso $L = \mathbb{R}i$. Però questa retta (l'asse delle y o asse immaginaria) non è un campo: infatti $i^2 = -1$ non appartiene più ad essa.

Esistono invece moltissimi campi intermedi tra $\mathbb{C}$ e $\mathbb{Q}$, ad esempio $\mathbb{Q} + \mathbb{Q}i$, $\mathbb{Q} + \mathbb{Q}\sqrt{5}$, $\mathbb{Q} + \mathbb{Q}\sqrt[3]{2} + \mathbb{Q}\sqrt[3]{4}$. È chiaro che questi insiemi sono chiusi rispetto a addizione e sottrazione. Verificare che lo sono anche rispetto a prodotto e divisione (con denominatore $\neq 0$)! Nei calcoli scrivere α per i, $\sqrt{5}$, $\sqrt[3]{2}$.

Se $E : K$ è un'estensione di campi, con $|E : K|$ si denota la dimensione di E come spazio vettoriale su K e se adesso $|E : K| < \infty$, allora si dimostra nel corso di Algebra che per ogni campo intermedio L si ha $|E : K| = |E : L||L : K|$. In particolare $|L : K|$ deve essere un divisore di $|E : K|$. Se ad esempio $|E : K| = 7$, possiamo concludere come prima per $\mathbb{C} : \mathbb{R}$ che non esistono campi intermedi non banali.

Evariste Galois (1811-1832) ha introdotto uno strumento ancora più fine per la classificazione dei campi intermedi, la teoria dei *gruppi*.

Le equazioni di Cauchy-Riemann

Consideriamo ancora l'estensione di campi $\mathbb{C} : \mathbb{R}$. Per ogni $c = a + ib \in \mathbb{C}$ l'applicazione $\bigcirc_z cz : \mathbb{C} \longrightarrow \mathbb{C}$ è $\mathbb{R}$-lineare (infatti queste applicazioni sono esattamente le applicazioni $\mathbb{C} \longrightarrow \mathbb{C}$ che sono addirittura $\mathbb{C}$-lineari) e rispetto alla base canonica di $\mathbb{R}^2$ su $\mathbb{R}$ corrisponderà quindi a una matrice A. Com'è fatta questa matrice?

La prima colonna è l'immagine di $(1, 0)$ tramite questa applicazione; ma $(1, 0)$ è uguale a 1 come numero complesso e $c1 = a + ib = (a, b)$, mentre la seconda colonna è l'immagine di $(0, 1)$. Ma $(0, 1) = i$, e quindi la seconda colonna è uguale a $ci = (a + ib)i = -b + ai = (-b, a)$ e ritroviamo il nostro vettore magico. La matrice della moltiplicazione con c è quindi $A = \begin{pmatrix} a & -b \\ b & a \end{pmatrix}$ e viceversa una matrice $A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}$ corrisponde alla moltiplicazione con un numero complesso (o, equivalentemente, definisce un'applicazione $\mathbb{C}$-lineare) se e solo se $a_{12} = -a_{21}$ e $a_{22} = a_{11}$. Queste condizioni prendono il nome di *equazioni di Cauchy-Riemann*.

8 Ottobre 2001

1. Siano $x, y \in \mathbb{R}^n$ e $s \in \mathbb{R}$. Allora $sx + (1-s)y = x + t(y-x)$ per un $t \in \mathbb{R}$. Per quale t?

2. Siano a, b e t numeri reali e $0 \le t \le 1$. Allora $a \le ta + (1-t)b \le b$.

3. Siano $x, y \in \mathbb{R}^n$, $t \in \mathbb{R}$ e $p := 0.3x + 0.7y$. Esprimere la distanza di p da x e la distanza di p da y in termini di $|x-y|$.

4. Risolvere con la regola di Cramer il sistema

$$\begin{aligned} 3x - 2y &= 9 \\ 2x + 6y &= 5 \end{aligned}$$

5. Dimostrare la forma generale del teorema fondamentale a pag. 3.

6. Risolvere il sistema

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

7. Risolvere il sistema

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

8. Risolvere il sistema

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

9. Trovare l'equazione di una delle due rette parallele alla retta

$$R = \{(1,2) + t(3,4) \mid t \in \mathbb{R}\}$$

che hanno distanza 5 da R.

10. Trovare l'equazione della retta passante per i punti $(2,4)$ e $(1,6)$.

11. Trovare l'equazione della retta passante per i punti $(-3,1)$ e $(10,3)$.

12. Siano $x, y \in \mathbb{R}^n$ e l'angolo α definito come a pag. 11. Quali dei seguenti enunciati sono corretti?

1. $(x,y) = |x||y| \cos\alpha$.
2. $(x,y) = |x||y| \sin\alpha$.
3. $|x+y|^2 = |x|^2 + |y|^2 + 2(x,y)$.
4. $|x-y|^2 = |x|^2 + |y|^2 - 2(x,y)$.

***

Date dei compitini:

Giovedì, 8 novembre, 11.45-13.00
Giovedì, 29 novembre, 11.45-13.00.

Il voto v verrà calcolato con la formula $v = \dfrac{2m+p}{3}$, dove m denota il voto migliore, p il voto peggiore.

15. Ottobre 2001

13. Quali delle seguenti affermazioni sono corrette?

A C è un linguaggio interpretato.
B Perl è un linguaggio interpretato.
C Perl è più veloce del C.

14. In quale linguaggio di programmazione sono scritte le seguenti istruzioni?

A *A5 F0 D8 0A 85 F1*
B *begin p:=0 for i:=1 to 4*
C *if n=0 then 200*
D *int n,k; double a;*
E *STA $F1*
F *class punto {...}*
G *real a(4), b(4)*
H *sub f {my $x=shift;}*
I *System.out.println(a);*
J *def f(x):*

15. $(2AEF)_{16}$.

16. Rappresentare 9258 in base 16.

17. Quali delle seguenti affermazioni sono corrette?

A C è un linguaggio compilato.
B Xlib è un linguaggio interpretato.
C Delphi è un ambiente di sviluppo per C++.
D In Fortran il tipo delle variabili non deve essere dichiarato.
E length è un metodo della classe String di Java.
F atof trasforma una stringa in un intero.

18. Usare la funzione *somma* a pagina 20 per definire una funzione *media* che calcola la media aritmetica di una lista di numeri.

19. Usare la funzione *media* e *map* per calcolare la media geometrica dalla media aritmetica.

20. Usare la funzione *media* e *map* per calcolare la media armonica dalla media aritmetica.

22 Ottobre 2001

21. $d|a$ e $d|b \Longrightarrow d|a+b$ e $d|a-b$.
22. $d|a \Longrightarrow d|ka$ per ogni k.
23. $d|a \Longleftrightarrow d|-a \Longleftrightarrow d||a|$.
24. $1|a$ e $a|0$ per ogni a.
25. $a|1 \Longrightarrow a = \pm 1$.
26. $0|a \Longrightarrow a = 0$.
27. $b|a \Longrightarrow \mathrm{mcd}(a,b) = |b|$.
    In particolare $\mathrm{mcd}(0,b) = |b|$ per ogni b.
28. $a|b$ e $b|a \Longrightarrow a = \pm b$.
29. $a|b \Longrightarrow a \leq |b|$.
30. $(G,+)$ sia un gruppo abeliano e A, B sottogruppi di G. Allora $A+B$ è un sottogruppo di G che contiene sia A che B.
31. Nelle ipotesi dell'esercizio 30 dimostrare che, se H è un altro sottogruppo di G che contiene sia A che B, allora $A+B \subset H$. $A+B$ è quindi il più piccolo sottogruppo di G che contiene A e B. Facile.
32. Scrivere una funzione *rapp16* in Perl per la rappresentazione esadecimale.
33. Calcolare $(A,E,F,F,3,0,A)_{16}$ a mano con lo schema di Horner.
34. —
35. $(G,+)$ sia un gruppo abeliano e A, B sottogruppi di G. Allora $A \cap B$ è un sottogruppo di G.
36. Siano $a, b \in \mathbb{Z}$ ed m l'unico numero naturale m tale che $\mathbb{Z}a \cap \mathbb{Z}b = \mathbb{Z}m$. Allora m è il minimo comune multiplo di a e b, gode cioè delle seguenti proprietà:
    (1) $a|m$ e $b|m$.
    (2) Se $k \in \mathbb{Z}$ e $a|k, b|k$, allora $m|k$.

29 Ottobre 2001

37. $\begin{vmatrix} 3 & 5 & 1 \\ 2 & 6 & 0 \\ 8 & 1 & 1 \end{vmatrix}$
38. $\begin{vmatrix} 8 & 4 & 3 \\ 6 & 2 & 5 \\ 1 & 1 & 1 \end{vmatrix}$
39. $\begin{vmatrix} 1 & 2 & 3 & 4 \\ 0 & 1 & 4 & 1 \\ 2 & 1 & 3 & 4 \\ 0 & 2 & 1 & 0 \end{vmatrix}$
40. Inventare un algoritmo di Gauß per il calcolo di un determinante con l'utilizzo della prop. 28/2, e ricalcolare i determinanti precedenti.
41. Trovare l'intersezione delle rette $(1,0)+\mathbb{R}(2,1)$ e $(2,5)+\mathbb{R}(1,3)$ nel piano col metodo indicato a pagina 27.
42. Trovare le equazioni per le rette dell'esercizio 41 e determinare il punto di intersezione risolvendo un sistema nelle incognite x_1 e x_2.
43. Trovare la proiezione ortogonale del punto $(1,2)$ sulla retta $(-1,0)+\mathbb{R}(3,1)$ e calcolare la distanza del punto dalla retta.
44. Calcolare la proiezione ortogonale dell'origine sulla retta $y = 2x+3$ e la distanza dell'origine dalla retta.
45. Siano v e a come a pagina 29. Dimostrare che per ogni $x \in \mathbb{R}^2$ valgono le relazioni
    $$(v,x) = -\det(a,x)$$
    $$(a,x) = \det(v,x)$$
46. Come diventano le formule per m a pagina 29 se $|a| = |v| = 1$?
47. v, w sia una base ortonormale di $\mathbb{R}^2$, cioè $|v| = |w| = 1$ e $(v,w) = 0$ (w è allora il nostro a oppure $-a$). Ogni vettore $x \in \mathbb{R}^2$ è allora della forma $x = \lambda v + \mu w$. Calcolare λ e μ.
48. Considerando il vettore $x := q - p$ e un sistema di coordinate con il centro in p, utilizzare l'esercizio 47 per dare una giustificazione geometrica delle formule di proiezione (nel caso speciale dell'esercizio 46). Fare un disegno.
49. Sia dato un vettore $v \neq 0$ di $\mathbb{R}^n$. Come si trova un vettore di lunghezza 1 che mostra nella stessa direzione di v e quindi determina la stessa retta (con lo stesso orientamento)?

5 Novembre 2001

50. $w \times v = -v \times w$ per $v, w \in \mathbb{R}^3$.

51. $(u, v \times w) = (u \times v, w)$ per $u, v, w \in \mathbb{R}^3$.

52. Dimostrare l'identità di Jacobi per il prodotto vettoriale.

53. p, q ed r siano tre punti di uno spazio vettoriale reale V. I vettori $q - p$ ed $r - p$ sono linearmente indipendenti se e solo se i punti p, q ed r non stanno sulla stessa retta.

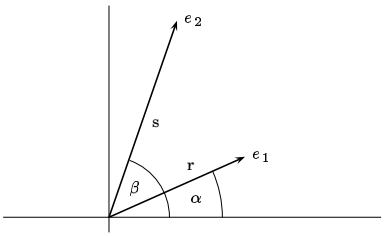
54. Trovare l'equazione del piano passando per $(0, 1, 2), (1, 2, 3)$ e $(4, 7, 1)$.

55. Trovare una forma parametrica per il piano con equazione $3x + 6y + 10z = 7$.

56. Una base ordinata e_1, e_2 di $\mathbb{R}^2$ può essere scritta nella forma

$$e_1 = (r\cos\alpha, r\sin\alpha)$$
$$e_2 = (s\cos\beta, s\sin\beta)$$

con $r, s > 0$ e $0 \le \alpha < \beta < 360°$ come nella figura. Quand'è che la base ordinata e_1, e_2 è positivamente orientata?



57. $(A, +, \cdot)$ sia un anello. Definiamo un'operazione binaria bilineare con $[a, b] := ab - ba$. Allora vale l'identità di Jacobi:

$$[a, [b, c]] + [b, [c, a]] + [c, [a, b]] = 0$$

Attenzione: La moltiplicazione in A non è necessariamente commutativa (altrimenti tutto è banale), quindi ad esempio in genere $abc - acb$ non sarà uguale a zero. Si dice che $(A, +, [\,])$ è un *anello di Lie* (però non è un anello!).

È chiaro che $[a, b] = 0 \iff ab = ba$, quindi l'anello di Lie descrive in un certo senso la commutatività o non-commutatività dell'anello originale. $[a, b]$ si chiama il *commutatore* di a e b.

58. Sia $n \ge 2$. Allora $\binom{n}{2} = n \iff n = 3$.

12 Novembre 2001

59. Programmare in PostScript la funzione definita da $f(x) = (x^2 + 1)(x^3 + 6)$ usando il diagramma di flusso per lo stack.

60. Programmare la stessa funzione in PostScript usando un dizionario.

61. Calcolare il volume del parallelepipedo generato da (1,2,3), (2,0,5), (4,1,2).

62. Programmare in PostScript la funzione definita da $f(x, y) = (x^2+y^2)(x-y)$ usando un dizionario.

63. Qual'è la matrice di una rotazione per 90°? Ritrovare la formula per il vettore magico.

64. A quale operazione geometrica corrisponde la moltiplicazione di un vettore di $\mathbb{R}^2$ con la matrice $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$?

19 Novembre 2001

65. $e^{i(\alpha+\beta)} = e^{i\alpha}e^{i\beta}$ per $\alpha, \beta \in \mathbb{R}$.

66. Qual'è il risultato di *printf("%d\n",13&27)* in C? Cosa si ottiene per 13|27? Spiegare.

67. $z\overline{z} = |z|^2$ per ogni $z \in \mathbb{C}$.

68. Dare una dimostrazione diretta (utilizzando le coordinate) del lemma 42/2.

69. $\overline{zw} = \overline{z} \cdot \overline{w}$ per ogni $z, w \in \mathbb{C}$.

70. $|zw| = |z||w|$ per ogni $z, w \in \mathbb{C}$.

71. Per ogni $z \in \mathbb{C}$ si ha

$$\text{Re } z = \frac{z + \overline{z}}{2}$$
$$\text{Im } z = \frac{z - \overline{z}}{2i}$$

72. $e^{z+w} = e^z e^w$ per ogni $z, w \in \mathbb{C}$.

Scrivere $z = x + iy$ e $w = u + iv$ e usare dall'analisi che $e^{a+b} = e^a e^b$ per $a, b \in \mathbb{R}$.

73. Il vettore magico di $z \in \mathbb{C}$ è proprio iz.

Fine