Dispense

Programmazione in C

Laboratorio di Calcolo e Fisica Computazionale I

Francesco Marrocco
Riccardo Mordente

Layout a cura di TecoSaur

5 marzo 2024

Indice

Prefazione

Agli studenti del corso di laboratorio di meccanica (o a chiunque entri in possesso di questo file),

Queste dispense nascono con lo scopo di aiutare un gruppo di matricole a organizzare i concetti chiave e i calcoli necessari per l'esame orale di questa materia, grande scoglio del primo anno della laurea triennale, in modo da avere tutto organizzato secondo un ordine che vada leggermente oltre le "pagine del quaderno sparse per la cameretta". L'utilità individuale che abbiamo trovato in questa organizzazione ci ha fatto poi riflettere su come questo pdf possa essere d'aiuto ad altri nostri colleghi, e siccome siamo convinti dell'idea che l'università debba essere un luogo di massima condivisione del sapere e aiuto reciproco, ci siamo decisi a mandare il nostro lavoro a chi ce lo chiedeva, con un risultato molto positivo (almeno a detta loro).

Sia chiaro, queste pagine sono anni luce lontane dal volere o potere sostituire eventuali libri di testo o le dispense dei professori (che trovate a questo link, e da cui abbiamo ripreso moltissime cose) ma cercano piuttosto di essere un buono strumento di sintesi, da utilizzare dopo aver assorbito le nozioni del corso da un punto di vista concettuale, per fare mente locale e non perdersi tra i tantissimi argomenti affrontati.
Parte fondamentale del file sono i calcoli, spesso lunghi e dispersivi, che è necessario affrontare e saper fare per dimostrare molti aspetti fondamentali (basti pensare alla linearizzazione, ai fit lineari...) e che in modo altrettanto frequente vengono *lasciati al lettore per esercizio*. Ora è bene chiarire che non abbiamo nessun problema con questa frase, siccome farsi due conti non fa mai male a nessuno, e finchè non ci si impasta bene le mani con alcune cose non le si comprende a fondo (inoltre se avete passato analisi si presume che li sappiate fare) ma purtroppo capita che non si riesce proprio a risolvere un integrale o una serie, o che la si sa fare ma durante il ripasso molto vasto si perde la memoria di quello che si è fatto, si mischiano i conti, ci si fa prendere dall'ansia o altre mille fattori, e siccome a Fisica ci insegnano a renderci la vita facile, abbiamo deciso di dedicare quanta più attenzione possibile ai calcoli espliciti, con molti passaggi (anche superflui) e metodi per la comprensione facile di alcune dimostrazioni, in modo da non perdere il filo.

Sperando che le nostre dispense possano esservi d'aiuto per lo studio dell'orale di laboratorio di meccanica, vi auguriamo una buona lettura.

Alice, Francesco, Riccardo e Daniele

Per qualsiasi errore nelle dispense, dubbi o altro potete contattarci alla mail `riccardo.mordente@gmail.com`. Il file è graficamente ispirato al layout gratuito a cura di Tecosaur, via https://github.com/tecosaur/BMC.

1 Nozioni Teoriche

Sommario

1.1 Sistemi numerali

Per rappresentare i numeri nei calcolatori non posso usare sistemi come quello romano, in cui i simboli numerici hanno sempre lo stesso valore (in VI e IV il simbolo V è sempre 5). Rappresento allora ogni $n \in \mathbb{N}$ in base 10: $192_{10} = 1 \times 10^2 + 9 \times 10^1 + 2 \times 10^0$, generalizzando:

$$n = \sum_{i=0}^{M-1} C_i b^i \tag{1.1}$$

In cui C è la cifra, quindi $C = \{0, 1...9\}$ in questo caso, e b è la base, quindi nel caso generico avrò $0 \leq C < b$, e M sono le cifre.

Una somma tra due numeri n in base b sarà così:

$$n + m = \sum_{i=0}^{M-1} \alpha_i b^i + \sum_{i=0}^{M-1} \beta_i b^i = \sum_{i=0}^{M-1} (\alpha_i + \beta_i) b^i$$

In cui $\alpha_i + \beta_i = \gamma_i$ e se $\gamma_i > b$ bisogna "ricomporlo" in base b

Esempio 1

$$\begin{aligned} 1492_{10} + 48_{10} &= 1 \times 10^3 + 4 \times 10^2 + 9 \times 10^1 + 2 \times 10^0 + 4 \times 10^1 + 8 \times 10^0 \\ &= 1 \times 10^3 + 4 \times 10^2 + 13 \times 10^1 + 10 \times 10^0 \\ &= 1 \times 10^3 + 4 \times 10^2 + 1 \times 10^2 + 3 \times 10^1 + 1 \times 10^1 + 0 \times 10^0 \\ &= 1 \times 10^3 + 5 \times 10^2 + 4 \times 10^1 + 0 \times 10^0 = 1540_{10} \end{aligned}$$

1.2 Sistema Binario

Il sistema Binario ha base $b = 2$, quindi le cifre sono solo $C = \{0, 1\}$ (il computer, tramite i transistor, capisce solo *acceso/spento*), e le cifre si chiamano *bit*.

n in base 10			n in base 2		
0	1	2	0	1	10
3	4	5	11	100	101
6	7	8	110	111	1000
9	10	11	1001	1010	1101
⋮	⋮	⋮	⋮	⋮	⋮

Tabella 1.1: Primi numeri del sistema decimale tradotti

Overflow: con M bit posso rappresentare fino a $0 \leq n \leq 2^M - 1$, quindi con 3 bit avrò fino al numero $2^3 - 1 = 8 - 1 = 7$, infatti $7_{10} = 111_2$.

La *conversione di base* da base 10 a base 2 si fa dividendo per 2, e se la divisione da resto la cifra è 1, altrimenti la cifra è 0, e a partire dall'ultima alla prima cifra si costruisce il numero

$$
\begin{aligned}
10_{10} = 1010_2 \longrightarrow \frac{10}{2} &= 5 \text{ no resto, } 0 \\
\frac{5}{2} &= 2 \text{ con resto,} 1 \\
\frac{2}{2} &= 1 \text{ no resto, } 0 \\
\frac{1}{2} &= 0 \text{ con resto,} 1 \\
\Longrightarrow 10_{10} = &\qquad 1010
\end{aligned}
$$

Si possono fare operazioni direttamente in base 2, sempre con la tecnica del resto che da come risultato la cifra 1 o la cifra 0

Esempio 2

$\frac{1010_2}{2} = \frac{1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0}{2} = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$ che ha come resto 0, allora $\frac{1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0}{2} = 1 \times 2^1 + 0 \times 2^0$ con resto 1, e $\frac{1 \times 2^1 + 0 \times 2^0}{2} = 1 \times 2^0$ con resto 0, infine $\frac{1 \times 2^0}{2}$ =resto 1 $\Longrightarrow 0101_2 = 5_{10}$, infatti in base 10 avrei semplicemente $\frac{10}{2} = 5$

Questo ragionamento vale per ogni tipo di operazioni: se voglio sommare due numeri in base 2 sommo le cifre, ricordando che $1 + 1 \neq 2$, in base 2 is ha che $1 + 1 = 10$.

1.3 Sistema Esadecimale

Si usa perché molto più compatto del binario, ma è sempre una potenza di 2, quindi la conversione è relativamente facile e rappresento numeri grandi con meno caratteri. La base è $b = 16$ e le cifre sono:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

Tabella 1.2: Cifre del sistema esadecimale

Quindi la conversione si effettua dividendo per 16 e segnando come cifra il resto da 0 a 16

Esempio 3

$$92_{10} \longrightarrow \frac{92}{16} = 5 \text{ con resto } 0{,}75 \text{ ovvero } 7{+}5{=}12(C)$$
$$\frac{15}{16} = 0 \text{ con resto } \quad 5(5)$$
$$\Longrightarrow 02_{10} = \quad 5C_{16}$$

Il sistema esadecimale è comodo per convertire: $1010_2 = 10_{10} = A_{16} \rightarrow 1010_2 \times 16 = 1010000_2$oppure $A \times 16 = A \times 16^1 + 0 \times 16^0 = A0 \Longrightarrow A0_{16} = 1010000_2$ ovvero 160, oppure con numeri più grandi $A7BC = 1010011110111100_2$, converto in gruppi di 4 bit, perché $16 = 2^4$.

1.4 BIT e BYTE

Il *BYTE* è un'unità di misura, in particolare $1B = 8bit$ quindi un numero a due cifre esadecimali occupa 2B.

Ovviamente il *Byte* cresce secondo il modello matematico del sistema internazionale (quindi $1KB = 1000B; 1GB = 10^9B$...) ma nel mondo dei computer questa cosa non è proprio esatta, visto che i dati operano su base 2 in realtà $1KB = 1024B; 1MB = 2048B$...

1.5 Numeri con segno

Per rappresentare i numeri col segno si potrebbe usare un *bit di segno* come 0 per i positivi e 1 per i negativi, quindi $0001_2 = +1$ mentre $1001_2 = -1$, ma oltre ad esserci un limite (± 11) c'è una degenerazione, ovvero $1000 = 0000 = 0$

Un secondo modo più efficiente è usare il *complemento a 2*, ovvero divido in 2 la lista di numeri operabili con M bit, e con la prima metà rappresento i numeri positivi incluso 0, con l'altra metà rappresento i numeri negativi a partire da -1

000	001	010	011	100	101	110	111
0	1	2	3	-4	-3	-2	-1

Tabella 1.3: Cifre con complemento a 2

Dati M bit posso rappresentare $2^{M-1} - 1$ n e $(-)2^{M-1}$ *complementi.*

Il *complemento* di un numero n a M bit è il numero $k : n + k = b^M \implies k = b^M - n$, infatti con $b = 2, M = 3$ il complemento di 1 è $(1) = 2^3 - 1 = 7$ che in binario è 111, infatti nella tabella $-1 = 111$, e siccome $a + (-a) = 0$ dovrò avere che $n + (2^M - n) = 0$, con l'esempio di prima, ovvero $1 + (-1) = 0$ avrò $001 + 111$ che scritto in colonna è

$$\begin{array}{ll} 001 & + \\ 111 & - \\ \hline \not{1}000 & \end{array}$$

che fa 0 perché bisogna considerare solo i primi 3 bit.

1.5.1 Rappresentazione in eccesso N

Similmente alla rappresentazione con il complemento, prendendo M *bit*, $N = 2^{M-1}$ rappresento i numeri con segno sottraendo N, ovvero $-n_2 = m_2 - N$ in un m è un altro numero in base 2

Esempio 4

Con $M = 3$ e $N = 2^{3-1} = 4$ avrò questa rappresentazione:
Per usare i complimenti k nel computer mi scrivo n come $n_i = n_{M-1}...n_0 = 2^M = 1 + \sum_{i=0}^{M-1} 2^i$ ovvero $1_2 + 1_2 + ...1_2$ *n volte* $\implies k = 1 + \sum_{i=0}^{M-1} 2^i - \sum_{i=0}^{M-1} n_i 2^i$
quindi inverto tutti i bit e sommo 1.

1.6 Rappresentazione in virgola mobile

Considero $a = \sum_{i=o}^{M} a_i b^i$ con a cifra ($0 \leq a_i \leq b$), per avere una parte frazionaria inserisco dopo la cifra a una parte che dipende non dalla divisione per 2, ma dalla moltiplicazione per 2, e che non sarà sempre precisa fino all'ultima cifra significativa, in base a quanti bit ho, quindi ci sono delle volte in cui il numero viene approssimato, si chiama *errore di roundoff*.

Esempio: $22,5_{10} \rightarrow 10110.1_2$ in cui $22_{10} = 10110_2$ e la parte frazionaria è $0.5_{10} = .1_2$ perché moltiplicando per due ho $0.5 \times 2 = 1$ con resto 0, quindi mi fermo e lo rappresento come .1

In modo generico quindi, la parte frazionaria p è

$$p = \sum_{i=-N}^{-1} a_i 2^i = a_{-1}2^{-1} + ... + a_{-N}2^{-N} \tag{1.2}$$

Quindi moltiplico per 2 finché non arrivo a 1, infatti $0.5_{10} = .1_2$, mentre $0.25_{10} = .01_2$ perché $0.25 \times 2 = 0.5$ con resto 0 e $0.5 = .1$, quindi per esempio $7.25_{10} = 111.01_2$.

Esempio 5

Facciamo un esempio di decimale più complesso: 0.54_{10} diventa $0.54 \times 2 = 1.08 \Longrightarrow 0.08 \times 2 = 0.16 \Longrightarrow 0.16 \times 2 = 0.32 \Longrightarrow 0.32 \times 2 = 0.64 \Longrightarrow 0.64 \times 2 = 1.28 \Longrightarrow 0.28 \times 2 = 0.56 \Longrightarrow 0.56 \times 2 = 1.12 \Longrightarrow 0.12 \times 2 = 0.24 \Longrightarrow 0.24 \times 2 = 0.48 \Longrightarrow 0.48 \times 2 = 0.96 \Longrightarrow 0.96 \times 2 = 1.92$
e si approssima finendo qui (o più in là con più bit), quindi questa cifra decimale diventa $0.54_{10} = .10001010001_2$

In realtà, come per la notazione negativa, c'è un metodo più efficace, che è la *notazione in virgola mobile* in cui $a = a_0 a_{-1} ... a_{-N}$ (ovvero rappresentazione esponenziale secondo la notazione scientifica), con $1 \leq a_0 < b$, per esempio $32.72 = 3.272 \times 10^1$, o in base 2 si potrebbe avere $11.01_2 = 1.101 \times 2^1$ in cui la parte frazionaria si chiama *MANTISSA* e l'esponente viene rappresentato con un eccesso di $N = 2^{M-1} - 1$.

Le rappresentazioni in virgola mobile sono a *singola precisione* quando occupano 4B, ovvero 32bit, di cui 8 per l'esponente, 23 per la mantissa e 1 bit per il segno, non devo memorizzare l'unica cifra intera perché è sempre la stessa, e sono a *doppia precisione* quando utilizzo 8B, di cui 11 per l'esponente, 54 per la mantissa e 1 per il segno, ovviamente i numeri che posso rappresentare anche con questo metodo sono finiti, non potrò mai rappresentare numeri infiniti su un calcolatore.

Definizione 1

IEEE 754: È lo standard per tutte le macchine per rappresentare i numeri in virgola mobile;

Definizione 2

ε_m *o Epsilon di macchina*: la minima differenza apprezzabile tra due numeri in virgola mobile, dipende dal numero di bit, quindi per avere più differenza devo necessariamente aumentare i bit; l'epsilon di macchina è $\varepsilon_m = 2^{-N}$ con N il numero bit della mantissa e $\forall a, b \implies |a - b| < \varepsilon_m$

Esempio 6

Il numero 13,75 in virgola mobile diventa 0 10000010 10111000000000000000000 in singola precisione, quindi 32 bit, dividisi così:
Il segno è positivo, quindi il bit è 0, l'esponente si ottiene perché in virgola fissa diventa $13,75_{10} = 1101.11_2$ e spostando la virgola in modo che la cifra sia 1 ottengo $1.10111_2 \times 2^3$
quindi l'esponente è 3, che va rappresentato in eccesso a $N = 127$ (secondo lo standard), diventando così 10000010, che sarebbe $3 + 127 = 130$ (perché ho 8 bit, quindi $N = 2^8 - 1 = 255$, quindi posso rappresentare 127 interi e 128 complementi, il range dello standard a eccesso 127 è teoricamente di 255 numeri per l'esponente, ma le formulazioni 00000000 e 11111111 sono usati per i casi speciali) e 130 in codice binario è proprio 10000010. La mantissa resta uguale, perché già convertita, ovvero .101111, ma vanno aggiunti gli 0 fino ad arrivare a 23 bit, quindi ottengo 10111000000000000000000.
Ovviamente si può usare la notazione esadecimale per un risultato più compatto, in questo caso $01000001011100000000000000000000 = B15C0000$.
La rappresentazione del numero in notazione scientifica sarebbe $(-1)^s \times 2^{E-127} \times 1.M$ dove s è il segno 0 o 1, *E-127* sarebbe $E = esp + 127$ (infatti nel caso di 13,75 abbiamo *esp=3* e $E = 3 + 127 = 130$, motivo per cui rappresentiamo l'esponente come 130 ovvero 10000010), e infine *1.M* rappresenta il bit 1 che non viene rappresentato perché implicito e la mantissa (per altri esempi vedi questo *link*).

1.6.1 Casi particolari

I numeri 00_{16} e FF_{16} , ovvero gli "estremi" dei bit vengono usati per dei casi particolari, accorciando i numeri rappresentabili con 32 o 64 bit ma introducendo due casi importanti nella rappresentazione:

$$00_{16} \Longrightarrow \text{l'esponente vale il minimo possibile, ovvero -126;}$$

$$FF_{16} \Longrightarrow \begin{cases} mantissa \neq 0 \Longrightarrow \text{NaN} \\ mantissa = 0 \Longrightarrow \infty \end{cases}$$

In cui *NaN* significa *Not a Number*, usato per espressioni che non sono calcolabili (come 0/0 o altre forme indeterminate).

Nel caso di 00_{16} avrò quindi il più piccolo numero rappresentabile

$$00000000000000000000000000000001$$

Ovvero $1 \times 2^{-23} \times 2^{-126} = 2^{-149}$ in cui assumo che $C = 0$ poiché deve essere il numero più piccolo rappresentabile, quindi non sommo 1.

Il numero più grande rappresentabile è invece

$$C = 1; 01.11111111111111111111111111111111111111111110 \times 2^{1023}$$

Non ho tutti 1 perché altrimenti avrei il caso particolare.

1.6.2 Somma in virgola mobile ed errori di approssimazione

Per sommare due numeri in virgola mobile, ho bisogno che abbiano lo stesso esponente, quindi il calcolatore sposta l'esponente fino a renderlo uguale a quello del numero più piccolo, poi si somma in binario e si ottiene il numero in virgola mobile.

Esempio 7

22.75 + 4.50 = 27.25
Per prima cosa si trasformano entrambi i numeri: $22.75 = 10110.11 = 1.011011 \times 2^4$ che in 32 bit è 01000001101101100000000000000000
$4.50 = 100.1 = 1.001 \times 2^2$ che in 32 bit è 01000000100100000000000000000000
Dopo di ciò si sommano in colonna (ricordandosi sempre che siamo in base 2):

```
01000001101101100000000000000000  +
01000001101001000000000000000000  =
--------------------------------
01000001110110100000000000000000
```

Questo numero è proprio 27.25, infatti il segno è +, l'esponente è 4 e la mantissa è 101101, quindi la rappresentazione decimale è 1.101101×2^4 o 11011.01 in cui in codice binario 11011 = 27 e .01 = 25.

Da notare è che, prima di eseguire l'addizione, il numero $4.50 = 1.001 \times 2^2$ viene portato al numero 0.01001×2^4 per avere lo stesso esponente, e viene fatto presente che nel secondo addendo *C=0*, quindi si procede a sommare solo la mantissa e lasciare l'esponente così com'è perché è già stato posto uguale per entrambi i membri.

Quando si sommano numeri molto grandi con numeri molto piccoli, siccome bisogna spostare di tanto la virgola, capita ci siano errori di approssimazione, spesso non trascurabili, soprattutto se queste addizioni vengono ripetute.

Esempio 8

100000000 + 1 = 100000000 ovvero il numero 1 quando viene portato in virgola mobile con stesso esponente viene praticamente eliminato dall'approssimazione: il numero 100000000 è 10111110101111000010000000 = $1.01111101011110000100000000 \times 2^{26}$ in cui l'esponente è 26 + 127 = 153 = 10011001 quindi in virgola mobile in numero diventa 100000000 = 01001100101111101011110000100000
Allo stesso modo $1 = 1 \times 2^0$ in cui $esp = 0 + 127 = 127 =$ 11111111 quindi ho che 1 = 01111111100000000000000000000000 ma quando

Esempio 8 continued

sommo i due numeri devo cambiare l'esponente del più piccolo, quindi $1 = 1 \times 2^0 = 0.00000000000000000000000001 \times 2^{26}$ ma visto che la mantissa ha "solo" 23 bit e in questa rappresentazione l'1 sta alla 26-esima posizione, viene completamente perso nell'addizione ed è come se si stesse sommando 0:

$$\begin{array}{l} 01001100101111101011110000100000 \quad + \\ 01001100100000000000000000000000 \quad = \\ \hline 01001100101111101011110000100000 \end{array}$$

Che è ancora il numero 100000000 visto che è come se avessimo aggiunto 0.

Questi errori posso essere trascurabili in alcuni casi di compilazione e catastrofici in altri, per esempio dati $a = 10^8, b = 1, c = 10^8$ posso eseguire $\frac{1}{a+b-c}$ in due modi:

$$\begin{cases} \frac{1}{(a+b)-c} = \infty \\ \frac{1}{(a-c)+b} = 1 \end{cases}$$

La differenza sta nel fatto che per fare $a + b$ devo approssimare b con l'esponente di a, ottenendo sempre a come nell'esempio di prima, e visto che $a = c$ ottengo una F.I, mentre facendo prima $a - c$ ottengo al denominatore $0 + 1$.

1.7 Altri formati dati

Per rappresentare i caratteri alfabetici utilizzo le *stringhe*, ovvero sequenze di caratteri, in cui ogni carattere viene codificato. Il modello standard *ASCII* (*American Standard Code for Information Interchange*) prevede l'utilizzo di un byte per ogni carattere, con l'utilizzo quindi di 8 bit e 256 ($2^8 - 1 = 255$ numeri rappresentabili, ma codifico anche il numero 0 che non viene considerato nella formula) caratteri rappresentabili, di cui meno della metà rappresentano le lettere minuscole e maiuscole, molte i caratteri speciali (come parentesi, punteggiatura…), altri sono comandi di controllo oppure comandi per la scrittura non rappresentabili come DEL (ovvero delete) oppure SPACE.

In particolare i primi 128 caratteri sono fissi e universali, e codificati per la scrittura della lingua inglese, mentre gli ultimi caratteri sono variabili in base alla codifica di alcune determinate lingue (come quelle latine che hanno accenti, dittonghi…).

Char	ASCII Code (Decimal)
a	97
b	98
c	99
d	100
e	101
f	102
g	103
h	104
i	105
j	106
k	107
l	108
m	109
n	110
o	111
p	112
q	113
r	114
s	115
t	116
u	117
v	118
w	119
x	120
y	121
z	122

Char	ASCII Code (Decimal)
A	65
B	66
C	67
D	68
E	69
F	70
G	71
H	72
I	73
J	74
K	75
L	76
M	77
N	78
O	79
P	80
Q	81
R	82
S	83
T	84
U	85
V	86
W	87
X	88
Y	89
Z	90

Char	ASCII Code (Decimal)
space	32
!	33
"	34
#	35
$	36
%	37
&	38
'	39
(	40
)	41
*	42
+	43
,	44
-	45
.	46
/	47
:	58
;	59
<	60
=	61
>	62
?	63
@	64
[	91
\	92
]	93
^	94
_	95
`	96
{	123
|	124
}	125
~	126
‘	145
’	146
“	147
”	148
•	149
˜	152

Char	ASCII Code (Decimal)
0	48
1	49
2	50
3	51
4	52
5	53
6	54
7	55
8	56
9	57

Char	ASCII Code (Decimal)
€	128
£	163
¥	165
$	36
©	169
™	153
°	176
˜	152
¡	161
¿	191

Figura 1.1: Tabella ASCII

Siccome non si possono rappresentare tutti quei caratteri delle lingue arabe, orientali… è stato creato un nuovo standard, l'*Unicode* (o anche *UTF: Unicode transformation format*) che aumenta il numero di bit in base alla versione (UTF-8, UTF-16, UTF-32…) per avere moltissimi caratteri in più rappresentabili, tanto che nelle ultime versioni è stato inserito l'elfico di Tolkien, caratteri musicali e altri simboli.

Per rappresentare le immagini, invece, si utilizzano i *pixel*, che sono l'unità di misura dell'immagine digitale. Ogni pixel è composto da 3 byte, quindi 24 bit in totale, e con i 3 byte si rappresentano i tre colori primari (rosso, blu, verde o in inglese *RGB*) con i quali si compongono tutti gli altri, questi 3 byte sono i *subpixel*, entità spesso fisiche, come in monitor e tv. Siccome ogni subpixel ha 8 bit, il numero di colori rappresentabili digitalmente è di $(2^8)^3 = 256^3 = 16$ milioni di colori. Siccome un'immagine a tanti milioni di pixel (il 4k ha $4096 \times 3840 = 15720640$ di pixel) occuperebbe un numero di byte enorme ($15720640 \times 3B = 47185920$ byte e ancora più bit), le immagini vengono compresse per occupare meno memoria, oppure vengono rappresentate tramite *vettori*. In un modo simile vengono rappresentati i suoni, associando a ogni Δt un valore che varia di intensità, cosi come i pixel delle immagini.

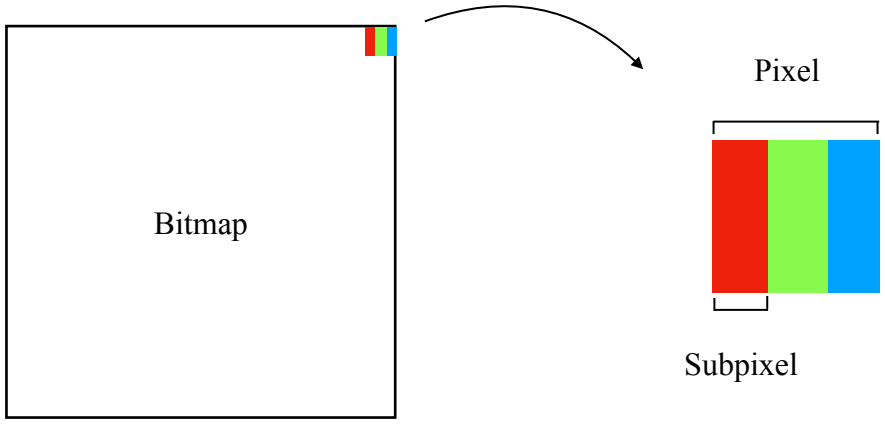


Figura 1.2: Composizione di un'immagine

1.8 Architettura Computer

L'elemento più importante di un computer è la *CPU* (*central processor unit*) che esegue istruzioni semplici come i calcoli (± con numeri $n \in \mathbb{N}$), leggendo e scrivendo dati con la memoria; la *FPU* (*floating point unit*) serve a eseguite i conti in virgola mobile, ovvero quelli che in processore non è in grado di fare; la *RAM* (*random access memory*) è la memoria volatile del computer, memorizza dati scritti su di essa dalla CPU, in particolare ogni dato (rappresentato sempre da una serie di bit, che sia un numero, una lettera, un'immagine…) occupa uno specifico indirizzo; la *SSD* (*solid-state drive*) è la memoria solida del computer, non volatile e fissa.

Per comunicare tra di loro, le componenti del computer usano dei *BUS* di vario tipo (di dati per comunare, di controllo per assicurarsi che nessun altra componente stia usando quel bus e di indirizzi per accedere ai vari indirizzi).

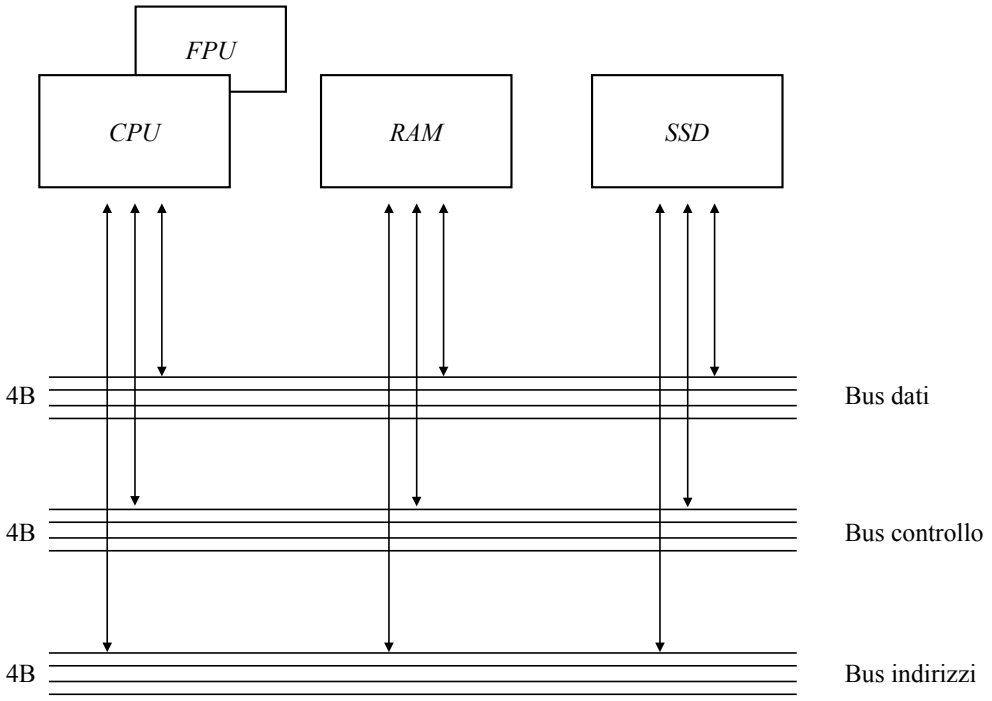


Figura 1.3: Architettura di un computer

Le istruzioni che la CPU da sono chiamate *OPCODE* e possono essere rappresentate da diversi numeri di bit in base alla CPU del computer (8, 64…), essi sono sequenziali e sintatticamente semplici.

Esempio: un'istruzione a 8 bit potrebbe essere 0000001 00001001 a 1B l'uno, che significa "leggi dalla memoria all'indirizzo 9 (1001 in binario è 9) il contenuto di A.

2 Linguaggi e C

Sommario

2.1 Linguaggi di basso e alto livello

Per comunicare ai calcolatori le istruzioni, utilizziamo i linguaggi, che possono essere di *basso livello* se le istruzioni precisano alla macchina passaggi come dove sovrascrivere gli indirizzi, dove andare a leggerli... oppure di *alto livello* se ogni istruzione contiene varie sotto istruzioni che poi vengono codificate in fase di *compilazione.*

Ecco il codice per scrivere *"Hello, World!"* in *Assembly* (linguaggio di basso livello) e il *Python* (linguaggio di alto livello).

```
1  SECTION .data
2  Msg: db "hello world", 10
3   MsgLen: equ $ - Msg
4  SECTION .text
5  global start
6  start:
7     push dword MsgLen
8     push dword Msg
9     push dword 1
10    sub esp, 4
11    mov eax, 4
12    int 80H
13    add esp, 16
14    push dword 0
15    sub esp, 4
16    mov eax, 1
17    int 80H
```

Codice 2.1: "Hello World" in Assembly

```
1  print("Hello, World!")
```

Codice 2.2: "Hello World" in Python

I linguaggi si dividono poi in *compilati* (ovvero che necessitano di essere tradotti in linguaggio macchina, questo avviene con appositi comandi come *-gcc* nel terminale del compilatore) e *interpretati* (ovvero vengono tradotti istantaneamente dal compilatore).

2.2 Comandi e funzionalità di C

C è un linguaggio di alto livello, è *sequenziale* (ovvero le istruzioni vengono eseguite in sequenza una dopo l'altra), la sua struttura di base è questa:

```
1  #include<stdio.h>
2
3  int main(void){
4
5      istruzione 1; //questo e' un commento
6      istruzione 2;
7      .
8      .
9      .
10     istruzione n;
11 }
```

Codice 2.3: Struttura base di un codice in C

Nella scrittura di C l'istruzione `#include` rappresenta l'inclusione di librerie con funzionalità non presenti altrimenti nel compilatore, e che in fase di pre-compilazione vengono aggiunge prima del codice, per dare istruzioni per esempio sui comandi come printf, scanf…

2.3 Immissione e stampa variabili

In C, per usare delle variabili, esse vanno dichiarate tramite una serie di comandi, che dipendono dal tipo di variabile immessa:

- `int`: serve per dichiarare una variabile intera con segno come 0,1,-4… (32 bit)
- `float`: serve per dichiarare una variabile float, ovvero a virgola mobile in singola precisione, quindi a 32 bit;
- `int`double: serve per dichiarare una variabile a virgola mobile in doppia precisione, quindi 64 bit;
- `char`: serve per dichiarare una variabile carattere che viene codificata in ASCII (8bit);

Le variabili vanno usate in base all'utilizzo che se ne fa, poiché variabili di tipo double occupano più memoria di quelle float e int, quindi non massimizzano l'efficenza del calcolatore, soprattutto in programmi con un numero alto di calcoli eseguiti. Gli altri comandi base, che riguardano la stampa e l'immissione delle variabili sono:

- `printf`: serve per stampare su schermo una stringa, che va inserita tra le virgolette "...";
- `scanf`: serve per l'immissione del valore di una variabile, in precedenza dichiarata (se è stata dichiarata con un valore, la scanf lo sovrascriverà).

In particolare, nell'uso della printf e della scanf, l'immissione dei dati dipende dal tipo di variabile, quindi avrò una struttura ti questo tipo

```
1 printf("%lf", a)
```

In cui in base alla scrittura del primo argomento, corrisponderà un tipo di variabile diversa:

- %d: serve per indicare un intero;
- %f: serve per indicare un float (virgola mobile singola precisione);
- %lf: serve per indicare un long float (virgola mobile doppia precisione);
- %x: serve per indicare un intero positivo in esadecimale;
- %u: serve per indicare un Unsigned;
- %c: serve per indicare un char, ovvero un carattere della tabella ASCII;
- %s: serve per indicare una stringa, ovvero una serie di caratteri ASCII;
- %p: serve per indicare un puntatore, stampa in esadecimale l'indirizzo di una variabile;

Una struttura in C con l'utilizzo di printf e scanf sarà di questo tipo:

```c
#include<stdio.h>

int main(void){

    int a, b=2;
    double c=3.12;

    printf("Inserisci il valore di a\n");
    //L'ultimo comando serve per andare a capo
    scanf("%d", &a);
    printf("I valori sono a=%d, b=%d e c=%lf", a, b, c);
}
```

Si possono poi modificare le variabili tramite i modificatori in modo da rendere il range di rappresentazione più vasto o più corto, aumentando o diminuendo anche il peso in bit delle variabili stesse:

- `long`: serve per aumentare la precisione di una variabile (es. long double, ovvero un numero in virgola mobile a doppia precisione e 128 bit);
- `short`: serve per diminuire la precisione di una variabile (es. short int, un intero con segno a 16 bit);
- `signed`: serve per esplicitare il fatto che una variabile ha segno;
- `unsigned`: serve per "eliminare" il fattore segno alle variabili (es. unsigned int è un intero positivo a 8 bit, posso rappresentare numeri perché non ho limitazioni dalle rappresentazioni con l'eccesso);

2.4 Operazioni e Operatori

In C le variabili possono essere operate tra loro, tramite una serie di comandi. Le operazioni più semplici sono quelle elementari di somma e prodotto, con l'aggiunta dell'*operatore modulo* che da il resto di una divisione tra interi, e questi operatori sono rappresentati dai seguenti simboli:

+	-	*	/	%
Operatore somma	Operatore sottrazione	Operatore prodotto	Operatore divisione	Operatore modulo

Tabella 2.1: Operatori principali

Se voglio operare con variabili di tipo diverso posso farlo, ma bisogna essere coscienti del risultato, per esempio una divisione tra interi dichiarati mi darà come risultato solo la parte intera, scartando quella decimale, così come l'immissione tramite scanf di un numero non intero, quando viene dichiarata l'immissione di un int, porterà a memorizzare solo la parte intera di ciò che si scrive.

In alternativa, posso usare il *casting esplicito* che consiste nel convertire una variabile, per poi operare con le opportune variabili dello stesso tipo o del tipo desiderato:

```
1  #include<stdio.h>
2
3  int main(void){
4
5      double a=3.2;
6      int b;
7      printf("%d", 2./3.); //ottengo come risultato 0, perche' il
8      //comando %d indica la stampa di un intero
9      b=a; //conversione esplicita
10     b=(int)a;
11     printf("((double)2)/3"); // converte il 2 in un double
12
13 }
```

Altri tipi di operatori sono gli *operatori di assegnazione* che operano appunto assegnando valori alle variabili, o modificandone il valore, per lo più servono a scrivere operazioni esprimibili con gli operatori elementari in modo sintetico, e sono:

- =: pone una variabile uguale a un'altra ($a = b$);

- +=: è un modo sintetico per aumentare il valore di una variabile con un'altra ($a+ = 2$ è come dire $a = a + 2$ in cui gli a non si semplificano perché C è sequenziale, aumenta

solo il valore di a tramite quello precedentemente usato;

- -=: simile al precedente ($a -= b \implies a = a - b$);
- /=: simile al precedente ($a /= b \implies a = a/b$);
- *=: simile al precedente ($a* = b \implies a = a * b$);

Operatori simili a questi sono gli *operatori unari,* relativi agli incrementi + + e - - che sono di due tipi: *postfix* e *prefix,* i primi si pongono dopo la variabile e i secondi prima, e in particolare gli unari prefix aumentano e restituiscono il valore della variabile, mentre quelli postfix prima restituiscono il valore della variabile nell'operazione, e poi lo incrementano:

```
1  #include<stdio.h>
2
3  int main(void){
4      double a=1, b;
5      a++; //ovvero a=a+1, ora a=2
6      b=a++; //ora b=2 e a=3 perche' il prefix prima usa
7      //la variabile e poi la incrementa
8      b=++a; //ora b=a=4
9
10 }
```

Gli *operatori logici* invece sono quelli detti anche *booleani* e sono delle equivalenze sulle variabili:

- &&: ovvero *and*, indica la proposizione e ($a = 2$ && $b = 3$);
- ||: ovvero *or*, indica la proposizione o;
- !: ovvero *not*, indica la proposizione non;

Gli *operatori relazionali* infine esprimono rapporti di maggioranza, minoranza e diversità delle variabili:

- < o >: indicano maggioranza o minoranza di una variabile;
- <= o >=: indicano i simboli $\leq$ e $\geq$;
- ==: non indica uguaglianza ma è l'*operatore di confronto* (per esempio 2 == 3 darà come risultato 0 perché l'affermazione è falsa);
- !=: ovvero *diverso*;

2.5 Macroprocessori

Il processore non accetta `#include` solo come comando base all'inizio di un programma C: posso definire delle costanti che nomino come voglio, e che in fase di pre-processing vengono sostituite con il valore che gli avevo assegnato (utile per non scrivere ogni volta valori come ... o per altre casistiche. Il comando in questione è `#define`

```
#include<stdio.h>
#define K 2.45

int main(void){
    double a;
    scanf("%lf", &a);
    printf("%lf", cos(a)/K); //al posto di K verra'
    //messo nel codice 2.45

}
```

Posso usare i macro per definire delle funzioni, per esempio `#define somma(x,y) ((x)+(y))` mi porterà ad avere una funzione somma, in cui se inserirò opportunamente i dati come `val=somma(2,3)` (è un esempio), otterrò la somma tra 2 e 3 in questo caso.

2.6 Costrutti if/else

I costrutti if/else servono per eseguire delle istruzioni o delle altre solo se sono soddisfatte alcune condizioni. In genere l'istruzione dell'if è seguita da quella dell'else, ma non è una regola, infatti possono esserci anche più if in base al numero di casi necessari.

```
#include<stdio.h>
int main (void){
    float a, b;
    scanf("%f %f", &a, &b);
    if (a>b) {
        printf ("a e' maggiore di b");}
    if (a<b){
        printf("b e' maggiore di a");}
    if (a==b){
        printf("a e' uguale a b");}
}
```

Si possono usare gli if/else insieme alle macro per creare codici che funzionano sulla base della dedizione di alcune macro, tramite i comandi `#ifdef` e `#endif`. Per utilizzare propriamente gli ifdef/else, bisogna compilare o meno il codice con il comando:

```
gcc -DA=... File.c -o File.exe
```

In cui DA=... indica che si sta definendo la costante A con un certo valore, mentre se questo non viene fatto e si compila normalmente, il programma seguirà l'else.

```c
#include<stdio.h>

int main (void){
    #ifdef A
        printf("A definito");
    #else
        printf("A non definito");
    #endif
}
```

2.7 Cicli for e while

I cicli for e while si utilizzano per iterare delle istruzioni un numero arbitrario di volte (anche 0), e si costruiscono cosi

```
for (i=..; i<...; i+=...) {
    codice;
}
```

```
while (condizione){
    codice;
}
```

La differenza tra i due è che while non viene eseguito necessariamente, ma solo se le condizioni sono verificate (si può usare come un if ma con più funzionalità), mentre il for eseguirà le istruzioni almeno una volta (quelle tra parentesi sono un esempio di variabile con un valore iniziale e finale e un incremento arbitrario, ma può essere costruito anche in maniera diversa.

Si possono anche costruire dei cicli nidificati che eseguono le istruzioni con condizioni, incrementi…

```
#include<stdio.h>

int main(void){
    for(i=0; i<3; i++) {
        for(j=0; j<3; j++){
            printf("i=%d, j=%d\n", i, j);
        }
    }
}
//9 stampe totali
```

Il comando `break` serve poi a far interrompere il ciclo (per esempio con un determinato if), e il comando `continue` a farlo ricominciare con l'iterazione successiva e tralasciando tutti i comandi sotto il continue, interni al ciclo, della stessa iterazione. Nei cicli nidificati che dipendono da più variabili, può esserci la necessità di inizializzare le variabili prima che il ciclo più esterno ricominci, o queste verranno aumentate di volta in volta dopo ogni ciclo esterno.

2.8 Buffering

Il buffer è una zona di memoria "di transito" (in inglese è letteralmente memoria tampone) che serve a massimizzare e velocizzare alcuni processi di trasmissioni dati o alcune operazioni. Un particolare uso del buffer è quello di memorizzare temporaneamente i dati di alcuni operatori come la scanf, prima eventualmente di mandarli su schermo. Il buffer è funzionale, ma ci sono dei casi in cui può essere disastroso se non gestito: siccome la scanf "assorbe" anche i caratteri che non può memorizzare (per esempio se immetto in una `scanf("%d")` i caratteri `"tr11"` oppure `"11.357"` mi memorizzerà solo 11, ovvero gli interi) e li manda nel buffer, ma se ho bisogno di un ciclo in cui richiedo l'immissione della scanf finché non soddisfa alcune caratteristiche, devo svuotare il buffer dopo ogni scanf, o il buffer pieno porterà a degli errori del ciclo come loop infiniti o blocchi del programma.

```
1  #include<stdio.h>
2
3  int main(void){
4      int i, j=0, res;
5
6      while (j==0){
7          res=scanf("%d", &i);
8          while(getchar()!='\n');//comando che svuota il buffer
9      if (res==1)
10         j++;
11     }
12 }
```

Il `while(getchar()!='\n');` è un ciclo che non fa niente finché non è falsa l'affermazione che il buffer è composto solo da uno spazio, quindi è vuoto, e permette allora di fare le operazioni successive.

2.9 Array

Gli array sono strutture dati che rappresentano vettori o matrici in base alla loro dimensione (non del vettore o della dimensione, ma dell'array), in particolare:

- Gli *array unidimensionali* rappresentano i vettori e si indicano con `array[n]` in cui la dimensione n è quella del vettore;

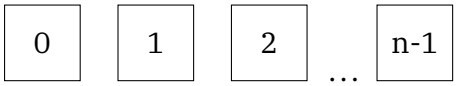
- Gli *array bidimensionali* rappresentano le matrici e si indicano con `array[n][m]` in cui la dimensione $n \times m$ è quella della matrice.

Un array si dichiara con `tipo_nome_[dimensione]` in cui il tipo indica il tipo di dato delle caselle di memoria dell'array (ovvero int, double…). Dopo la dichiarazione dell'array, esso può essere inizializzato con degli specifici valori, oppure possono essere definiti in seguito, ma bisogna sempre specificare l'indice che si sta inserendo:

```
1  #include<stdio.h>
2
3  int main(void){
4      double v[3]={1,2,3};//sto dichiarando il vettore con
5      //componenti v1=1, v2=2 e v3=3
6      double m[2][2]={{1,2},{3,4}};
7      double w[3];
8      w[0]=1;
9      w[1]=43;
10     w[2]=7;
11 }
```

In cui la post immissione del vettore w avviene a partire dall'indice 0 fino al 2, perché gli array vengono memorizzati in caselle di memoria che partono dalla 0, quindi l'ultima casella avrà come numero $n-1$ in cui n è la dimensione dell'array.

Struttura di un'array:



Nell'esempio di prima l'array `v[3]` rappresenta il vettore $\begin{pmatrix}1\\2\\3\end{pmatrix}$ e l'array bidimensionale `m[2][2]` rappresenta la matrice $m = \begin{pmatrix}1 & 2\\3 & 4\end{pmatrix}$. Un errore comune nella scrittura degli array è di cercare di inserire dati in caselle successive a $n-1$ dopo l'inizializzazione, ma ciò è un errore grave e porta alla stampa di caratteri casuali o ad altri errori (per esempio `x[3]=7`).

L'inizializzazione degli array deve essere fatta nel modo giusto, in base a ciò che si vuole ottenere, per esempio se pongo int v[3]={0}; avrò un vettore inizializzato a 0, ovvero $\begin{pmatrix}0\\0\\0\end{pmatrix}$, mentre se pongo int int v[3]={1}; non avrò il vettore $\begin{pmatrix}1\\1\\1\end{pmatrix}$ ma bensì il vettore $\begin{pmatrix}1\\0\\0\end{pmatrix}$, quindi se voglio il vettore con tutti 1 non potrò inizializzare come prima ma dovrò inserirlo separatamente per ogni indice, oppure con

```
for (i=0; i<3; i++) {
    v[i]=1;
}
```

Anche le matrici vengono memorizzate come i vettori, ossia in caselle di memoria in cui le prime i caselle indicano i primi i indici della prima riga, poi la seconda riga e cosi via fino alla riga j:

Struttura di una matrice 3 x 2:

Visto ciò, si possono allora utilizzare gli array unidimensionali come se fossero delle matrici, se vengono dichiarati come `array[N*M]`, e per accedere alle caselle di riga e colonna si indicano con `array[i*M+j]` vista la sequenza di memoria degli array (quest'ultima è solo una curiosità).

Gli array possono essere utilizzati per ordinare una sequenza di numeri, in particolare tramite l'algoritmo *bubble sort* che dato un array con una sequenza di numeri, scambia gli ultimi due indici se l'ultimo è più grande del penultimo, facendolo salire, e lo fa finché il numero non ne incontra un altro più grande.

Questo processo viene iterato finché l'array non è in ordine, quindi il numero minimo di iterazioni per fare questo processo è 0 (caso banale, array già in ordine), mentre il numero massimo è $\frac{N(N-1)}{2}$.
Ovviamente una volta che il 5 è arrivato in cima, non ci sarà più bisogno di un ciclo che arrivi fino allo slot di memoria del 5, ovvero v[n-1], ma si può fermare tranquillamente a v[n-2] siccome sarà il numero più grande da posizionare dopo le prime iterazioni. L'algoritmo bubble sort può essere scritto cosi:

```c
#include <stdio.h>
#define NC 5

int main(){
  int data[NC]={4,5,1,2,3};
  int i,j, temp;

  printf("data[] = {"); // stampo l'array iniziale
  for (j=0;j<NC;j++){
      printf("%d",data[j]);
      if (j < NC-1)
        printf(",");
    }
  printf("};\n");
  printf("ordino dal piu' grande (data[0]) al piu'
  piccolo (data[4])\n");
  for (i=0;i<NC-1;i++){
      for (j=NC-1; j > i; j--)
      if(data[j] >= data[j-1])
        {
          temp=data[j-1]; //temp e' una variabile temporanea
          //per non perdere uno dei due elementi
          data[j-1]=data[j];
              data[j]=temp;
        }
    }
      printf("iterazione i= %d: data[] = {", i);// stampo l'array
      //ordinato
      for (j=0;j<NC;j++)
        {
          printf("%d",data[j]);
          if (j < NC-1)
            printf(",");
        }
      printf("};\n");

    }
```

2.10 Stringhe

Le stringhe, ovvero sequenze di dati formate da caratteri con ordine prestabilito, non esistono propriamente in C, ma vengono utilizzate sfruttando i vettori e trasformandoli in array di testo. Come un qualsiasi array, le stringhe si inizializzano con `char_nome_[dimensione]` in cui sta volta la dimensione indica il numero di caratteri in ASCII che l'array contiene, in alternativa si può solo inizializzare così:

`testo[]="Testo" oppure testo[]='T', 'e', 's', 't', 'o', '0'`

Nota bene che se l'immissione dei caratteri è diretta, cella per cella, come nel secondo esempio, bisogna far attenzione a dichiarare una dimensione in più (se si dichiara) e ad aggiungere all'ultima casella di memoria 0 che C aggiunge alla fine di ogni stringa.

Un'input formato da una stringa avrà questa forma:

```
1 char testo[];
2 int res;
3
4 res=scanf("%s", testo);
```

Ovvero senza il classico & prima della variabile, e con il %s che è il comando di immissione variabile delle stringhe (vedi Sezione 2.3). Un uso avanzato, in cui l'immissione dei caratteri è limitata a un numero specifico, potrebbe essere fatto così:

```
1 char testo[];
2 int res;
3
4 res=scanf(   %255[^\n]   , testo);
```

In cui immette massimo 255 caratteri, siccome il 256-esimo è il NULL di fine array (mentre `[^\n]` indica che vengono memorizzati tutti i caratteri che non sono spazi, altrimenti ridurrei il range), e posso usarlo per esempio per combinare la richiesta di scanf, il buffer e le stringhe per richiedere nuovamente immissioni di caratteri se questi non rispettano alcune condizioni.

Le stringhe si possono copiare con un ciclo for, ma solo se le due stringhe hanno la stessa dimensione, altrimenti, come per gli array, si crea un buffer overflow. In alternativa, si può usare il comando `strcpy`, sempre con le stesse accortezze.

2.11 Generazione di numeri casuali

La generazione dei numeri casuali avviene tramite due comandi del tipo rand (ovvero *random*) e sono gestiti da un *seed*. La generazione sulla base del seed fa intendere che i numeri non siano del tutto casuali ma pseudo-casuali, infatti ogni seed lega i numeri con una relazione solo apparentemente casuale, motivo per cui lo stesso seed darà la stessa sequenza di numeri. Per ovviare a questo problema si pone come seme il comando seed=time(0) in cui time(0) è un'istruzione che fa parte di #include<time.h> e che da come risultato i secondi passati dal 01/01/1970, motivo per cui un seed del genere sarà sempre casuale.

Una generazione di 10 numeri casuali avrà una struttura come questa:

```
#include <stdio.h>
#include<stdlib.h> //include i comandi di rand
#include<time.h>
#define N 10 //i numeri verrano estratti nell'intervallo [0;N]
int main (void){
int i, x, seed=time(NULL);
srand(seed); //inizializza il seed
for(i=0; i<10; i++){
    x=((double)rand()/(RAND_MAX+1))*(N+1);
    printf("%d \n", x);}
}
```

In cui `rand` fornisce un numero casuale tra 0 e `RAND_MAX` che è già impostato nell'istruzione, quindi per avere un intervallo di generazione di `[0;N]` uso la formula:

$$x = \frac{random}{RANDMAX + 1} \times (N + 1) \tag{2.1}$$

In alternativa, posso usare il comando `srand48()`:

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
int main (void){
int i, x, seed=time(NULL), min=0, max=100;
srand48(seed);
for (i=0; i<10; i++) {
    x=(drand48()*(max-min+1)+min);
    printf("x=%d\n", x);
    }
}
```

2.12 Puntatori

Si è visto come ogni variabile viene memorizzata in RAM ad uno specifico indirizzo di memoria, ognuno con il suo "peso" in base al tipo. Una variabile che contiene l'indirizzo di memoria di un'altra variabile si chiama *puntatore*, infatti "punta" uno specifico indirizzo di memoria.

```
#include<stdio.h>
int main(){
int a=2;
int *puntatore;
puntatore = &a;
printf("a=%d, Indirizzo di a=%p \n", a, puntatore);
//in questo caso l'indirizzo e' 0x16b7a377c
}
```

In questo caso il puntatore è usato specificatamente per stampare l'indirizzo di *a* in formato esadecimale con l'utilizzo di %p (vedi Sezione 1.1).

Come le altre variabili, anche i puntatori hanno un "tipo" che va dichiarato, d'altronde sono variabili che hanno un loro slot di memoria, quindi la loro struttura di base sarà:

```
\tipo_*nome;
```

In cui però il tipo non dipende dal puntatore in sé, ma dalla variabile che si decide di fargli puntare (ogni variabile occupa un numero diverso di byte in base al tipo in memoria, il puntatore si comporterà di conseguenza).

Con il comando *p indico invece il contenuto della locazione di memoria puntata dal puntatore (ovvero il valore della variabile), quindi posso cambiare indirettamente il valore della variabile con il processo di *dereferenziazione:*

Un esempio di puntatore è quando scriviamo scanf("%d", &a); in cui & indica di porre l'intero immesso nella casella di memoria di a, in alternativa potrei scrivere scanf("%d", p); con p puntatore, avrei la stessa cosa.

```
#include<stdio.h>
int main(){
    int a;
    int *p; p = &a;
    *p=2;
    printf("a=%d \n", a);
    //il valore che esce e' 2
}
```

Quando faccio operazioni sui puntatori, posso modificare il contenuto della locazione, come nell'esempio di prima, ma se agisco solo sulla variabile puntatore quello che modifico è la casella di memoria che esso punta, quindi per esempio sommare 1 a un puntatore intero vuol dire sommargli 8 byte:

```
#include<stdio.h>
   int main(){
 int i, data[5]={1,2,3,4,5};
 int *p;
 p = &(data[0]);//ora p punta l'indirizzo della prima
 casella
 *(p+1)=3; //ora pongo data[1]=3
 printf("data={");
 for (i=0; i<=4; i++){
 }
  printf("%d,", data[i]);
  printf("}\n");
}
//il risultato della stampa è "data=1,3,3,4,5"
```

Ovviamente posso fare lo stesso discorso con tipi diversi di puntatori, come i puntatori char, ecco un esempio:

```
#include<stdio.h>

int main(){
    int i;
    char txt[]="ciao";
    char *p;
    p = &(txt[0]);//ora p punta l'indirizzo della prima casella
    p[2]='A'; //ora pongo txt[2]="A"
    printf("txt=%s\n", txt);
}
//il risultato della stampa è "txt=ciAo"
```

2.12.1 Puntatori a file

Un altro modo per utilizzare i puntatori è per accedere ai file di testo, ovvero i *UTF-8* in cui sono scritti caratteri e numeri, che possono essere importati e registrati nel programma C per altri usi. Il puntatore usato per questo tipo di uso è il `FILE *pointer` che ha una sua struttura specifica:

```
#include<stdio.h>
#include<stdlib.h>

int main(){
    FILE *p;
    fp=fopen("nome file", "modalita'");
    .
    .
    .
    fclose(fp);
}
```

In cui quindi si dichiara il puntatore, si apre il file, si eseguono le istruzioni relative al file e si chiude. Le modalità di utilizzo del file sono:

- `"r"` ovvero modalità *read*, in cui il file può essere letto;
- `"w"` ovvero modalità *write*, in cui il file può essere scritto, ma sovrascrivendo ciò che già era scritto sul file;
- `"a"` ovvero modalità *append*, in cui vengono aggiunte informazioni al file senza intaccare la sua struttura antecedente;
- `"r+"` ovvero modalità *read/write*, in cui il file può essere letto e scritto contemporaneamente;

Le azioni principali che vengono eseguite sul file sono invece:

- `fscanf(fp, "\%..", \&variabile);` ovvero si registrano delle variabili dichiarate in precedenza con del contenuto del file;
- `fprintf(fp, "\%..", variabile);` ovvero si trascrive il file con delle variabili presenti all'interno del programma e dichiarate;

È sempre buona norma porre un if di controllo che verifiche che quando `fopen==NULL` il programma viene chiuso e si stampa un errore. Questo succede quando il nome del file è sbagliato, quando è stato eliminato o ci sono altre problematiche. Questa è la sintassi:

```
if (fopen==NULL){
    printf("Errore nell'apertura file.\n");
    exit(1);
}
```

Dove `exit(1);` appartiene alla libreria `#include<stdlib.h>` e fa uscire dal programma.

Un esempio di programma in cui si trascrivono dati in un file è questo:

```
#include <stdio.h>
#include <stdlib.h>

int main(){
int num, k=0, j=0, res=0, i;
double a;
FILE *fp;
printf ("Quanti numeri vuoi inserire?\n");

while(k==0){
    res=scanf("%d", &num);
    while(getchar()!='\n');
    if(res==0 || num<0)
        printf("Valore non valido! Riprova\n");
    else
        k++;
}
//inizio scrittura file
fp = fopen ("dati.txt", "w");
  if (fp == NULL){
      printf ("C'e' stato qualche problema nell'
      apertura del file!\n");
      exit (1);}
  for (i=0; i<num; i++){
    printf("Inserisci un nuovo numero: \n");
    scanf("%lf", &a);
    fprintf (fp, "%lf \t", a);
    fflush (fp);
  }
fclose (fp);
//fine scrittura file
}
```

2.13 Funzioni

Le funzioni in C sono blocchi di codice riutilizzabile, usati per una maggiore efficienza nella compilazione e nella struttura, suddividendo il codice principale in sottocodici con input e output. Alcune di queste funzioni sono già impostate nelle librerie standard di C per esempio.

La notazione delle funzioni è questa:

```
<tipo><nome>(<argomento1>, <argomento2>...<argomenton>){
    .
    .
    .
    return(variabile);
}
```

In cui la variabile del `return` è la stessa del tipo, così come `int main()` restituisce un intero, ovvero 0 per convenzione. In alternativa posso porre void come argomento della funzione, e la funzione non restituirà niente (se per esempio la funzione ha una serie di istruzioni di scanf, printf... ma non prevede la classica meccanica di input/output di numeri come le funzioni analitiche). Le funzioni possono anche essere inizializzate sopra il main e definite al di fuori di esso, in seguito, alla fine del programma.

Esempio di uso di funzione:

```
#include<stdio.h>
#include<math.h>

double f(double x){
    double d;
    d = sin(x)+cos(2*x)*3*x;
    return d;
}

int main(){
    double a=1.2;
    printf("f(1.2)=%lf \n", f(a));
}
```

Un altro concetto importante è quello delle *variabili globali*, ovvero variabili che non vengono dichiarate all'interno del main o delle funzioni ma al di fuori, e sono in comune tra tutte le funzioni, quindi la loro sintassi è:

```
#include <stdio.h>
int a, b=2;

int main(){
  .
  .
  .
}
```

Queste variabili però vanno mantenute tali e mai riscritte all'interno delle funzioni (a meno che non lo si faccia in modo cosciente per qualche motivo specifico), perché in questo modo si creano delle *variabili nascoste*, ovvero la variabile locale della funzione copre quella globale, che quindi non potrà mai essere usata nella funzione, perché ci si riferirà sempre a quella locale.

2.13.1 Puntatori a funzione

I puntatori non si riferiscono solo a locazioni di memoria relative alle variabili, ma a qualsiasi locazione, compresa quella relativa alla funzione stessa. Un esempio è:

```
#include <stdio.h>

void f(int p[2][2]);
void g(int p[2][2], void (*f)(int p[2][2])); //gli argomenti di g
//sono una matrice e un puntatore a funzione

int main(){
  int a[2][2]={{1,2}, {3,4}};
  g(a, f); //passo a g l'array a la funzione f, equivalente
  //a g(a, &f)
}
void f(int p[2][2]){
  p[0][1]=3; //cambia il valore del coefficiente a12 della matrice
}
void g(int p[2][2], void (*f)(int p[2][2])){
  (*f)(p); //dereferenziazione
}
```

2.13.2 Memorie HEAP e STACK

Le memorie *heap* e *stack* sono due tipi differenti di memoria, che hanno un ruolo specifico nell'archiviazione delle variabili in memoria durante la compilazione di un codice:

Nella memoria di tipo *stack*, veloce, volatile e per lo più piccola (8MB all'incirca) sono memorizzate le variabili implementate nelle funzioni, poiché la memoria stack è come una piccola pila di memoria più grande, in cui ogni pila è riservata all'uso esclusivo di una singola funzione, in questo modo si ha una maggiore efficienza nella gestione dei dati e nell'esecuzione;

Nella memoria di tipo *heap* vengono memorizzate le variabili di tipo globali, che sono in comune con tutte le funzioni e sono fuori dal main (anche il main è una funzione).

Conoscere la differenza tra i due tipi di memoria è di cruciale importanza per evitare errori che possano compromettere l'esecuzione del programma, per esempio dichiarare un'array di un miliardo di double risulta in un errore, perché la *stack frame* (ovvero il frame di memoria) della funzione non riesce a lavorare con una simile quantità di bit.

2.14 Implementazione numerica

Così come nell'analisi, nasce il problema dell'implementazione del calcolo delle aree, ovvero data $f(x) \in [a;b]$ si cerca l'area sottesa alla curva nell'intervallo $[a;b]$, ovvero l'integrale definito della funzione tra i due estremi, che si calcola tramite le primitive $F(x)$, quindi l'area sarà

$$A = \int_a^b f(x)\,dx = F(b) - F(a)$$

Questo nei calcolatori non è sempre semplice, perché bisogna trovare la primitiva delle funzioni, cosa non sempre immediata. Piuttosto, per un computer, si preferisce usare dei metodi analitici di approssimazione molto accurata:

Metodo dei rettangoli

Consiste nel formulare la somma (superiore o inferiore) di rettangoli ugualmente distanti tra loro di base Δx_i e tale che $\sum_{i=1}^{n} \Delta x_i = x_b - x_a$ ovvero la lunghezza dell'intervallo, quindi ogni rettangolo avrà come estremi $x_i e x_{i+1}$ e l'altezza del rettangolo è $f(x_i)$ per le somme inferiori e $f(x_{i+1})$ per le somme superiori (per funzioni crescenti, l'inverso per funzioni decrescenti). Quindi l'area approssimata della funzione sarà la somma dei rettangoli, ovvero $A \approx \sum_{i=1}^{n} f(\Delta x_i) \times \Delta x_i$

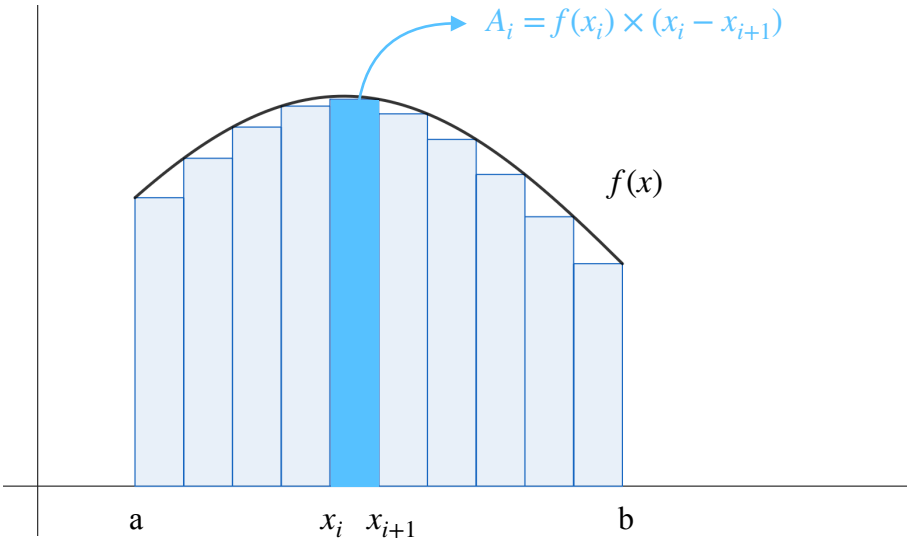


Figura 2.1: Partizione di una funzione per il calcolo integrale

L'errore che si fa nell'approssimazione dell'area dipende da una costante, e la troviamo nel seguente modo: chiamata $\tilde{\tilde{A}}$ l'area approssimata ho che $\tilde{\tilde{A}}_i = f(x_i)\Delta x_i$ e che la differenza tra le due aree è $|\tilde{\tilde{A}} - A| = k\Delta x = o(\Delta x)$ quindi più piccola sarà la base e ovviamente migliore sarà l'approssimazione, in cui $\Delta x_i = x_{i+1} - x_i = \frac{a-b}{n}$ con n il numero di punti di divisione della partizione dell'intervallo, quindi in definitiva l'area approssimata con il metodo dei rettangoli è:

$$\tilde{\tilde{A}} = \frac{b-a}{n}\sum_{i=0}^{n} f(x_i)$$

Metodo del punto di mezzo

Approssimo la funzione nel suo punto di mezzo, ovvero:

$$\tilde{\tilde{A}}_i = \Delta x_i \frac{f(x_i + x_{i+1})}{2}$$

E la somma compone l'area approssimata totale della funzione, in cui $|\tilde{\tilde{A}} - A| = k(\Delta x)^2$;

Metodo dei trapezi

Simile al metodo dei rettangoli, la funzione si approssima semplicemente con delle figure geometriche diverse, i trapezi, che a parità di divisione dell'intervallo possono avere in genere un'approssimazione migliore. In questo caso $f(x_i)$ non sarà più l'altezza del rettangolo ma la base (minore o maggiore), mentre $f(x_{i+1})$ è l'altra base, e l'altezza è Δx_i quindi seguendo la formula dell'area del trapezio $\tilde{\tilde{A}}_i = \Delta x_i \frac{f(x_i) + f(x_{i+1})}{2}$ e l'area totale approssimata è:

$$\tilde{\tilde{A}} = \Delta x_i \sum_{i=0}^{n} \frac{f(x_i + x_{i+1})}{2}$$

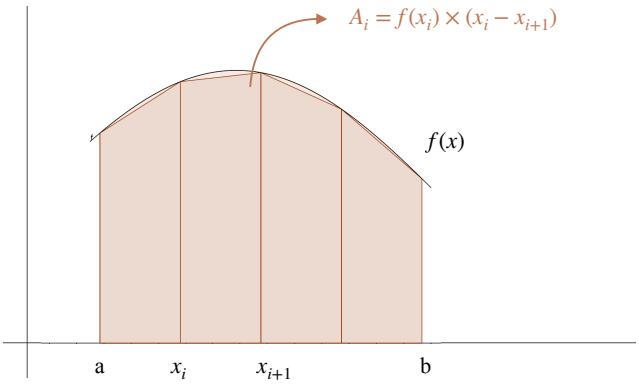


Figura 2.2: Partizione per il metodo dei trapezi

Metodo Montecarlo:

Chiamato anche metodo *hit and miss* è l'unico metodo valido quando non si ha più a che fare con funzioni $f : \mathbb{R} \to \mathbb{R}$ come quelle viste fin ora, ma con funzioni del tipo $f : \mathbb{R}^n \to \mathbb{R}$ in n variabili. Il metodo permette quanto meno di definire una stima, e questo viene fatto utilizzando il teorema della media del calcolo integrale (ovvero che data $f(x) \in [a;b]$ si può approssimare f tramite

$$\tilde{f} = f(z) = \frac{\int_b^a f(x)dx}{b-a}$$

Con $z \in [a;b]$) utilizzando al posto dell'integrale una delle tecniche viste sopra, e siccome posso calcolarmi la media della funzione generando n punti casuali x_i all'interno dell'intervallo, utilizzo questo metodo piuttosto che calcolare la media tra punti equidistanti tra loro (per questo *hit and miss*).

3 Python

Sommario

3.1 Comandi e funzionalità

Python è un linguaggio di alto livello ed è interpretato, quindi non richiede una compilazione ogni volta che si vuole usare, ma piuttosto si eseguirà tramite il comando da terminale:

`python3 nomefile.py`

Una delle funzionalità di Python, attuata nel calcolo scientifico, è la capacità di mostrare dei grafici da dati importati da altri file. La struttura di base di un programma python è questa:

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 .
5 .
6 .
```

In cui il comando `import` è simile a quello `#include` che si trova in C, ovvero serve a importare delle librerie con funzioni specifiche, in questo caso quelle per il plot dei grafici di funzioni e quelle per la creazione di array.

3.2 Struttura base dei grafici

La struttura basilare di Python per il plot dei grafici è questa:

```
1  import matplotlib.pyplot as plt
2  import numpy as np
3
4  plt.title('Titolo')
5  plt.xlabel('Etichetta asse x')
6  plt.ylabel('Etichetta asse y')
7  x, y = np.loadtxt('File.dat', usecols=(0, 1), unpack=True)
8  plt.plot(x, y, label='-m')
9  plt.savefig('Figura.pdf')
10 plt.show()
```

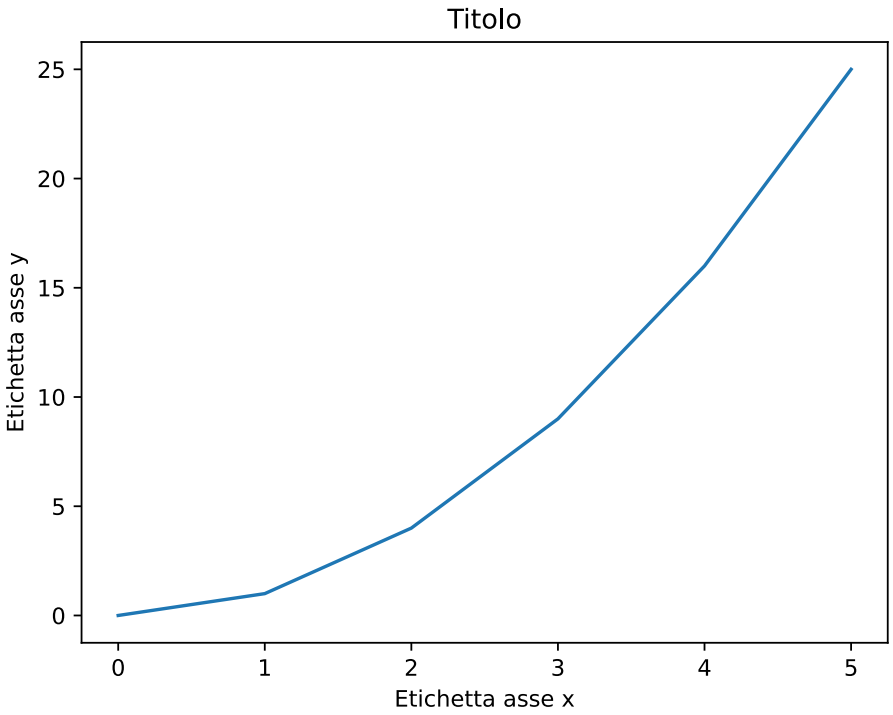
In cui i comandi hanno le seguenti funzionalità:

- `plt.xlabel` definisce il nome degli assi (x in questo caso, basta cambiare lettera);
- `np.loadtxt` crea un array (non sono come quelli in C) a partire dai dati di un file, con l'utilizzo di specifiche colonne (che si può tranquillamente omettere se il file è chiaro e non ci sono colonne extra);
- `plt.plot` definisce la struttura vera e propria del grafico con l'aggiunta di curve o dataset (l'importante qui è capire il funzionamento non il significato), in cui il terzo argomento indica il tipo di curva (in questo caso con '-m' una linea color magenta, ma ce ne sono altri);
- `plt.savefig` salva la figura con il nome e l'estensione data nella directory in cui è presente il file python;
- `pt.show()` mostra il grafico;

Nell'esempio il file da cui vengono importati i dati contiene i seguenti dati:

0	0
1	1
2	4
3	9
4	16
5	25

Il grafico che ne esce fuori è questo:



Per mostrare una semplice retta invece (quindi senza l'utilizzo di importare dati) si utilizza:

```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(-10, 50, 100)
y = x
plt.plot(x, y, label='retta y=x')
plt.title('Titolo')
plt.xlabel('Asse x')
plt.ylabel('Asse y')
plt.savefig('Figura.pdf')
plt.legend()
plt.show()
```

In cui `plt.legend()` mostra la legenda sul grafico.

3.3 Istogrammi

Similmente alle funzioni, gli istogrammi in Python si formano con l'importazione di dati da un file (sempre con due tipologie di dati, uno relativo all'altezza della barra e uno relativo al dato a cui è associato quel valore), e il comando per il plot degli istogrammi è

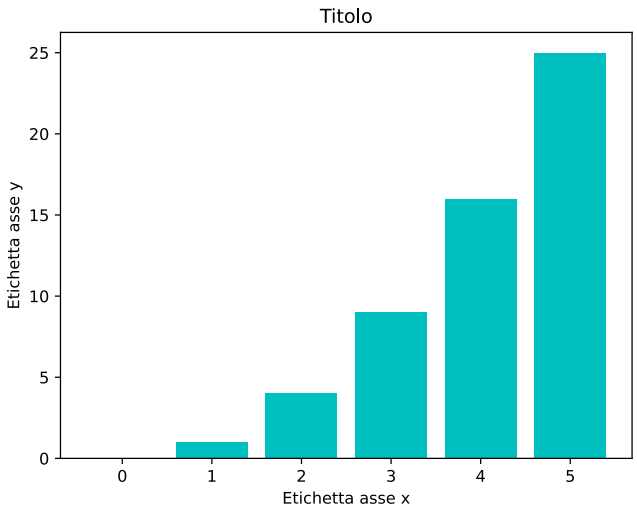
```
import matplotlib.pyplot as plt
.
.
.
plt.bar(x, y, color='c',label='Testo')
```

Il resto dei comandi, soprattutto per l'importazione delle variabili, rimangono inalterati. Un esempio di istogramma (sempre con i dati del primo esempio dallo stesso file di testo) è questo:

```
import matplotlib.pyplot as plt
import numpy as np

plt.title('Titolo')
plt.xlabel('Etichetta asse x')
plt.ylabel('Etichetta asse y')
x, y = np.loadtxt('File.dat', usecols=(0, 1), unpack=True)
plt.bar(x, y, color='c',label='Istogramma')
plt.savefig('Istogramma.pdf')
plt.show()
```

Il cui grafico è:



4 Appendice

Sommario

4.1 Comandi utili in C

- `sizeof`: indica la “grandezza” in byte di una variabile (es. se a è un double allora sizeof(a) darà come risultato 8, ovvero 8 byte e 64 bit);

- `<float.h>`: da includere tramite include dà una serie di parametri relativi ai limiti di macchina come o minimi e massimi numeri rappresentabili;

- `strcpy`: serve per copiare due stringhe tra loro;

- `fflush(fp)`: serve per svuotare il buffer di un file aperto tramite puntatore FILE*, in questo caso con il nome fp;

4.2 Comandi utili in Python

- `linespace(a, b)`: restituisce un vettore di numeri equi spaziati nell'intervallo , di norma il vettore contiene 50 numeri ma si può aggiungere la specificazione di tale numero con una terza variabile, per esempio `np.linespace(0, 10, 100)`;
- `plt.xlim(a, b)`: imposta il limite dell'asse x tra i valori a e b, allo stesso modo il comando `plt.ylim(a, b)` imposta quello per l'asse y;

- `plt.legend(loc='upper left')`: imposta la posizione della legenda nel grafico, in questo caso in alto a destra. Nota sempre che per fare in modo che la legenda mostri qualcosa, il grafico deve avere un label, quindi quando definisco `plt.plot(x, y, 'm', label='grafico')` devo includere il label come in questo esempio;

Tabella dei marker per i grafici:

Marker	Marker Option	Marker	Marker Option
.	Point marker	h	Hexagon marker
,	Pixel marker	H	Hexagon marker
o	Circle marker	+	Plus marker
v	Triangle marker	X	X marker
p	Pentagon marker	D	Diamond marker
<	Triangle marker	d	Diamond marker
>	Triangle marker	|	Vline marker
s	Square marker	_	Hline marker

LineStyle Option		Color Options	
-	Solid line	b	Blue
- -	Dashed line	g	Green
-.	Dash-dotted	r	Red
:	Dotted line	c	Cyan
		m	Magenta
		y	Yellow
		k	Black
		w	White

4.3 Esercitazione 9

Si scriva un programma chiamato *hitmiss.c* per calcolare con il metodo Monte Carlo l'area A della superficie che si ottiene considerando l'intersezione delle superfici delle due ellissi individuate dalle seguenti equazioni:

$$\frac{x^2}{4} + y^2 = 1 \text{ e } x^2 + \frac{y^2}{4} = 1$$

Il valore analitico dell'integrale è $A = 4 \arcsin\left(\frac{4}{5}\right) \approx 3.71$ In particolare il programma dovrà:

1. Chiedere in input il numero Np (intero positivo) di punti da generare per stimare A con il metodo Monte Carlo;

2. 2. Calcolare l'area A come segue (metodo hit and miss):
   - Generare a caso N_p punti all'interno del quadrato di lato pari a 4 centrato anch'esso nell'origine degli assi, ovvero generare a caso punti di coordinate (x, y) con 2 < x < 2 e 2 < y < 2;
   - Contare il numero di volte N_h che i punti di coordinate (x, y) cadono all'interno della superficie che si ottiene considerando l'intersezione delle superfici delle due ellissi definite dalle equazioni (1) e (2). Suggerimento: la condizione in C perché un punto di coordinate (x, y) sia in A si può scrivere con `x*x/4 + y*y < 1 && x*x + y*y/4 < 1`. Una stima dell'area A sarà dunque fornita dalla seguente formula $A = 16\frac{N_h}{N_p}$ dove 16 è l'area del quadrato all'interno del quale si stanno generando gli Np punti casuali di coordinate (x, y);
3. 3. Stampare su schermo il risultato dell'integrazione (ossia il valore di A) per Np = 107 ed il valore analitico utilizzando 2 cifre dopo la virgola. Nello scrivere il programma si richiede che vengano implementate le seguenti funzioni:
   - inserimento(...), che richiede l'inserimento di un numero intero positivo;
   - genera_coordinata(), che restituisce un numero casuale in virgola mobile compreso tra 2 e 2.

Testo del codice *hitmiss.c* :

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<time.h>
int inserimento(int *Np);
double genera_coordinata();
int main(){
    int Np, seed=time(NULL), i, Nh=0;
    int *p;
    double A, x, y;
    p=&Np;
    srand48(seed);
    printf("\nBenvenuto, questo e' un programma che calcola l'area
    dell'intersezione tra due ellissi.\n");
    inserimento(p);
    for(i=0; i<Np; i++){
        x=genera_coordinata();
        y=genera_coordinata();
        if ((x*x/4 + y*y)<1 && (x*x + y*y/4)<1)
            Nh++;
    };
    A=16*((double)Nh/Np);
    printf("\nIl valore dell'area per %d punti generati vale
    %.2lf, mentre il valore analitico dell'integrale
    %.2lf\n\n", Np, A, 4*asin((double)4/5));
}
////////////////////////////////////////////////////////////
int inserimento(int *Np){
    int N, res, k=0;
    printf("\nInserire un numero intero positivo di punti da
    generare per stimare l'area A, quindi Np=");
    while(k==0){
        res=scanf("%d", &N);
        while(getchar()!='\n');
        if(res==0 || N<0){
            printf("Immissione non valida, riprova. Quindi Np=");
            continue;
        }
        else{
            k++;
            break;
        }
```

```
    }
    *Np=N;
    return *Np;
}
////////////////////////////////////////////////////////////
double genera_coordinata(){
    int min=-2, max=2;
    double n;
    n=drand48()*(max-min+1)+min;
    return n;
}
```

Dal terminale, con $N_p = 10^7$ il risultato è 2.38.

Seconda parte: copiate il file hitmiss.c in un nuovo file chiamato hmsave.c e modificatelo in modo che i punti generati, che cadono all'interno di A, vengano salvati su di un file chiamato punti.dat. Generate Np = 1000 punti e con python graficateli insiemi alle due ellissi indicate nelle equazioni (1) e (2), Modificate quindi tale codice aggiungendo (tra plt.ylim e plt.show) le opportune istruzioni per graficare anche i punti contenuti nel file punti.dat. Salvate infine il grafico così ottenuto nel file punti.png.

All'interno del codice vengono modificate queste linee di codice:

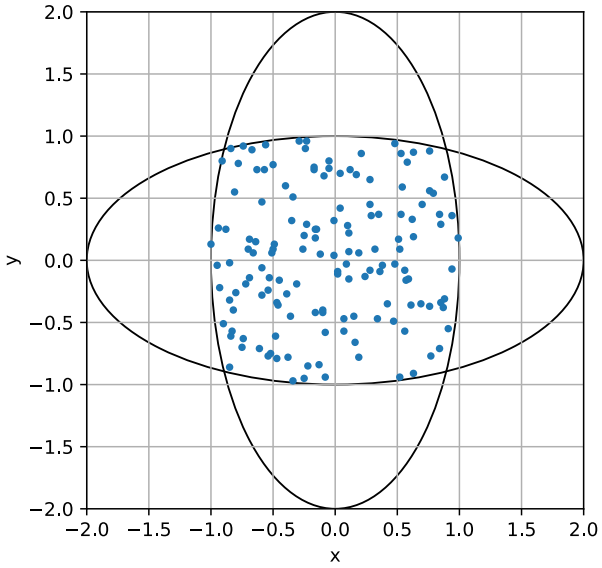
```
   [..]
int main(){
   [..]
    //apertura e scrittura file
    fp=fopen("punti.dat", "w");
    if(fp==NULL){
        printf("\nErrore nell'apertura del file.\n");
        exit(1);
    }
    for(i=0; i<Np; i++){
        x=genera_coordinata();
        y=genera_coordinata();
        if ((x*x/4 + y*y)<1 && (x*x + y*y/4)<1){
            Nh++;
            fprintf(fp, "%.2lf \t %.2lf\n", x, y);
        }
    }
    fclose(fp);
    //chiusura file, stampa risultati
   [..]
}
```

Il file python è:

```
import matplotlib.pyplot as plt
from matplotlib.patches import Ellipse
import numpy as np

plt.title('Metodo Hit and miss')
#grafico ellissi
a = plt.subplot(1,1,1, aspect='equal')
a.add_patch(Ellipse((0, 0), 4, 2, 0, fill=False))
a.add_patch(Ellipse((0, 0), 4, 2, 90, fill=False))
plt.xlabel('x')
plt.ylabel('y')
plt.grid()
plt.xlim(-2, 2)
plt.ylim(-2, 2)
#grafico punti
x, y = np.loadtxt('punti.dat', unpack=True)
plt.plot(x, y, '.')
plt.savefig('hitmiss.pdf')
plt.legend()
plt.show()
```

Il grafico per 1000 punti è questo:



4.4 Moltiplicazione matrice vettore

Scrivere un programma in C che implementi la moltiplicazione matrice per vettore e vettore per matrice:

```c
#include<stdio.h>
#include<stdlib.h>

int main(){
int i, j, k=0, res, dumb, choice=0, MxV[3], temp[3][3];
int M[3][3]={{1,2,3}, {4,5,6}, {7,8,9}}, V[3]={1,2,3};

//Stampe iniziali e scelta moltiplicazione
printf("\nQuesto e' un programma che calcola la moltiplicazione
Matrice x Vettore o Vettore x Matrice, premi
INVIO per continuare.\n");
getchar();
//stampa matrice e vettore
printf("La matrice M   :\n\n");
for(i=0; i<3; i++){
    printf("\t %d \t %d \t %d \n", M[i][0], M[i][1], M[i][2]);}
printf("\nIl vettore V   :\n\n");
for(i=0; i<3; i++){
    printf("\t %d", V[i]);
}
//Scelta moltiplicazione
printf("\n\nDigitare 1 se si vuole eseguire la moltiplicazione
a sinistra, ovvero MxV, oppure 2 se si vuole eseguire la
moltiplicazione a destra, ovvero VxM\n\n");
while(k==0){
    res=scanf("%d", &choice);
    while(getchar()!='\n');
    if(res==0)
        printf("\nImmissione non valida, riprova.\n");
    else if(choice==1 || choice==2)
        k++;
    else
        printf("\nImmissione non valida, riprova.\n");
 }
//Moltiplicazizone a sinistra
if(choice==1){
    printf("\nHai scelto 1! Il risultato    il vettore MxV=\n\n");
    for (i=0; i<3; i++){
```

```
39          for (j=0; j<3; j++){
40              temp[i][j]=M[i][j]*V[j];
41          }
42              MxV[i]=temp[i][0]+temp[i][1]+temp[i][2];
43              printf("\t %d", MxV[i]);
44          }
45  }
46 printf("\n\n");
47 //Moltiplicazione a destra
48 if(choice==2){
49     printf("\nHai scelto 2! Il risultato    il vettore VxM=\n\n");
50     for (i=0; i<3; i++){
51         for (j=0; j<3; j++){
52             temp[i][j]=M[j][i]*V[j];}
53             MxV[i]=temp[i][0]+temp[i][1]+temp[i][2];
54             printf("\t %d", MxV[i]);}
55  }
56 printf("\n\n");
57 }
```