Uso del calcolatore per il laboratorio di Fisica

Luca Baldini (luca.baldini@pi.infn.it)
Carmelo Sgrò (carmelo.sgro@pi.infn.it)

Versione 0.3.0 (Mon Feb 10 14:09:37 2014)

INDICE

COME VANNO LETTI QUESTI APPUNTI?

Dipende ovviamente da cosa sapete già. Vi accorgerete che queste dispense sono di fatto una piccola collezione di programmi, organizzata per tematiche generali, ognuno dei quali è pensato per illustrare (e risolvere) un piccolo problema che si può incontrare in laboratorio.

Non c'è bisogno, dunque, di leggere tutto di un fiato da cima a fondo. Cominciate da "Hello World!" (capitolo 2)—giusto per verificare che siete in grado di eseguire il più semplice programma possibile—e poi sentitevi liberi di passare alla parte che trovate più interessante, tornando eventualmente indietro se c'è qualcosa che trovate particolarmente ostico.

Noterete che i programmi di esempio sono racchiusi in piccole cornici ognuna delle quali ha un *hyperlink*: se clickate con il mouse arrivate direttamente alla versione *online* che potete copiare ed incollare su un file di testo per eseguirla sul vostro computer. Per completezza, nella parte inferiore della cornice, dopo la linea `[Output]`, trovate l'*output* di ogni programma—cioè il risultato che dovreste ottenere.

Infine, sulla pagina web da cui presumibilmente avete scaricato questo documento (`https://bitbucket.org/lbaldini/computing/downloads`), trovare anche un archivio compresso, in formato zip, contenente tutti i programmi ed i dati necessari alla loro esecuzione (oltre al documento stesso). Buon divertimento!

Pisa, 10 febbraio 2014.

1

INTRODUZIONE E CONCETTI DI BASE

1.1 CALCOLATORI E LINGUAGGI DI PROGRAMMAZIONE

In termini generali un calcolatore (o *computer*—nel seguito useremo le due parole senza particolari distinzioni) è un dispositivo in grado di eseguire una sequenza specifica di operazioni aritmetiche e/o logiche. Le operazioni da eseguire vengono specificate attraverso *istruzioni* ed una serie (ordinata) di istruzioni prende il nome di *programma*. Ne vedremo molti esempi nel seguito.

Nonostante si tratti di un argomento tanto interessante quanto complesso, in questa breve introduzione non parleremo affatto dell'*hardware*, cioè degli elementi fisici che costituiscono un calcolatore. Ci limitiamo a dire che, al livello dell'hardware, le istruzioni sono essenzialmente una serie di 0 ed 1 (o *bit*, cfr. la sezione 7.1) codificati nel cosiddetto *linguaggio macchina* e ciascuna istruzione esegue un compito estremamente specifico.

Dal punto di vista di un umano, leggere un programma in linguaggio macchina non è estremamente illuminante, per usare un eufemismo. (In effetti sarebbe di fatto impossibile per chiunque capire il flusso di un programma di media complessità a partire dalla sequenza di bit del linguaggio macchina). Fortunatamente al giorno d'oggi il programmatore, e specialmente il programmatore "casuale", non è quasi mai costretto a confrontarsi con i dettagli dell'hardware: esistono linguaggi di alto livello che agiscono come uno strato intermedio di *astrazione* e la traduzione di un programma scritto in un linguaggio di alto livello in una serie di istruzioni in linguaggio macchina (che possano essere eseguite dal calcolatore) è demandata ad un altro programma.

Un po' di terminologia. Il *codice sorgente* di un programma è l'insieme di istruzioni corrispondente al programma stesso, scritte in un linguaggio di alto livello (cioè leggibile, e comprensibile, da un umano). Il *compilatore* è un programma che traduce il codice sorgente in linguaggio macchina, creando il cosiddetto *eseguibile*. In alcuni casi (in particolare nei linguaggi di *scripting* come Python) la fase di compilazione può avvenire dietro le quinte nel momento in cui si esegue il programma (in tal caso si parla di *interprete* anziché di compiatore), ma per i nostri scopi tutto sommato questo è un dettaglio marginale.

Nel seguito di questa breve introduzione avremo modo di vedere molte linee di codice sorgente. Talvolta chiameremo i nostri programmi *script* o *macro*, utilizzando queste tre parole in modo sostanzialmente indifferenziato.

1.2 PERCHÉ PYTHON E SCIPY?

Se non sapete cosa vogliono dire queste due parole, non preoccupatevi. Lo vedremo tra un attimo. Ma, prima di cominciare, vale la pena di spendere due parole sul *software* che abbiamo scelto di utilizzare in laboratorio (ben consci che ogni scelta implica rinunce e compromessi).

Python è un linguaggio di programmazione generalista noto per essere semplice da utilizzare. È utilizzato e sviluppato da un'ampia comunità e dotato di una libreria standard molto vasta. Insieme a pacchetti come `scipy`, `numpy` e `matplotlib` costituisce la base di un vero e proprio *ecosistema* di software libero per applicazioni scientifiche che si è guadagnato notevole popolarità negli ultimi anni.

1.3 INSTALLAZIONE ED UTILIZZO

Essendo l'installazione materia estremamente *volatile*, nel senso che le informazioni rilevanti tendono a cambiare rapidamente, abbiamo pensato di raccogliere ciò che ci sembrava importante sulla pagine web `https://bitbucket.org/lbaldini/computing/wiki/Installazione`, che segnaliamo qui come punto di partenza. Nel seguito assumeremo che il lettore si in grado di scrivere ed eseguire un semplice programma Python.

2

IL MIO PRIMO PROGRAMMA

Per illustrare ad un principiante la sintassi di base di un linguaggio di programmazione tradizionalmente si utilizza un programma che stampi su un dispositivo di visualizzazione (tipicamente il terminale) le parole "Hello world!"[1] (e noi non faremo eccezione). L'*hello world* di Python è essenzialmente una riga, come mostrato qui sotto.

https://bitbucket.org/lbaldini/computing/src/tip/examples/hello_world.py

```
1 print('Hello world!')
2
3 [Output]
4 Hello world!
```

Fermiamoci per un attimo ad analizzare il frammento di codice, perché ne vedremo molti altri nelle pagine che seguono. Per prima cosa i numeri di linea a sinistra non hanno nessuna funzione strutturale se non quella di identificare, appunto, le linee; ci sarà comodo quando dovremo discutere parti di codice specifiche ma, tanto per essere espliciti, ignorateli quando trascrivete i programmi per eseguirli.

Tutto quello che viene dopo la linea `[Ouput]` rappresenta il risultato che dovreste attendervi quando eseguite il programma, per cui di nuovo: non dovete trascriverlo quando copiate il programma.

Nel nostro caso, di fatto, *hello world* si riduce ad una sola riga di codice (la linea 1). Digitandola interattivamente o scrivendola in un file di testo da passare all'interprete dovreste vedere apparire sul terminale qualcosa di simile alla linea 4.

Se siete arrivati a questo punto... Complimenti! Da qui in poi la strada verso le cose più utili è tutta in discesa. Altrimenti è inutile andare avanti. Fate un bel respiro e ricominciate da capo.

2.1 ED IL MIO PRIMO "COMMENTO"

Nel linguaggio dell'informatica un *commento* è un costrutto speciale, senza alcuna funzione strutturale, che il programmatore può usare per inserire nel codice sorgente delle annotazioni utili (ma che il compilatore o l'interprete semplicemente ignorano).

Se vi state chiedendo quale sia l'utilità dei commenti... Aspettate di trovarvi a dover modificare un programma di media complessità scritto da qualcun altro (o dai voi stessi dieci anni prima). Se il codice sorgente non è commentato adeguatamente avrete trovato la risposta che cercate.

Se vi state chiedendo perché parlare di commenti prima ancora di aver visto un programma degno di questo nome... L'abitudine a commentare il codice sorgente è, per la ragione scritta sopra, un'ottima abitudine. Prima la si acquisisce, meglio è.

Detto questo, ci sono due modi per commentare il codice in Python. Per commenti di una riga è sufficiente iniziare la linea stessa con un cancelletto:

1 Nel seguito scriveremo i nostri programmi *in Inglese*, nel senso che utilizzeremo nomi Inglesi per le variabili e scriveremo i commenti in Inglese. L'Inglese è la lingua internazionale dell'informatica e la lingua in cui tipicamente il computer ci risponde (cfr. la sezione **??** sui messaggi d'errore), per cui tanto vale fare l'abitudine fin da subito.

https://bitbucket.org/lbaldini/computing/src/tip/examples/hello_world3.py

```
# This is a one-line comment.
print('Hello world!')

[Output]
Hello world!
```

Per commenti di più righe è sufficiente includere il blocco di testo tra due serie di tre (singoli o doppi) apici:

https://bitbucket.org/lbaldini/computing/src/tip/examples/hello_world4.py

```
"""
And this is a multi-line comment.

It can be long, too...
"""
print('Hello world!')

[Output]
Hello world!
```

Gli esempi di codice che vedremo nel seguito saranno ampiamente commentati. Quando li copiate e li adattate alle vostre esigenze non è strettamente necessario copiare i commenti, ma ricordati quanto detto sopra!

3

VARIABILI E TIPI

Le *variabili* sono i mattoni fondamentali con i quali si costruisce un programma. Come vedremo una variabile è essenzialmente un nome simbolico contenente un *valore* (ad esempio un numero, una stringa, una lista e così via).

In Python una varibile si crea (o, più correttamente si *inizializza*) con una semplice assegnazione:

https://bitbucket.org/lbaldini/computing/src/tip/examples/var_init.py

```
s = 'Hello World!'
```

Nel frammento precedente abbiamo creato una stringa di testo (il suo nome simbolico è `s` ed il suo valore è "Hello World!"), che è possibile riutilizzare nel seguito. Così il nostro primo esempio di programma si può riscrivere equivalentemente (sia pure senza nessun vantaggio reale, in questo caso) come:

https://bitbucket.org/lbaldini/computing/src/tip/examples/hello_world2.py

```
s = 'Hello world!'
print(s)

[Output]
Hello world!
```

L'operazione di assegnazione di una valore ad una variabile ha alcune sfaccettature non banali che non sempre risultano ovvie a chi non ha mai programmato. Quando scriviamo

https://bitbucket.org/lbaldini/computing/src/tip/examples/assignment.py

```
x = 1 + 2
print(x)

[Output]
3
```

quello che effettivamente succede è che il computer per prima cosa calcola il valore a destra dell'operatore `=` (in questo caso 3), quindi assegna il valore stesso alla variabile a sinistra dell'operatore. Ne segue che la variabile può comparire sia a destra che a sinistra dell'operatore di assegnazione (a patto che abbia già un valore definito):

https://bitbucket.org/lbaldini/computing/src/tip/examples/increment.py

```
x = 2
print(x)
x = x + 1
print(x)
x += 2
print(x)

[Output]
2
```

```
3
5
```

In effetti si tratta di una sintassi utile e largamente usata in molti linguaggi di programmazione. Vale la pena soffermarsi a riflettere un attimo sul fatto che in algebra elementare l'analogo della linea 3

$$x = x + 1 \tag{1}$$

non ha senso (o per lo meno è palesemente falso). Parleremo più avanti (a proposito delle espressioni condizionali) della differenza tra gli operatori di assegnazione (=) e di confronto (==).

3.1 I TIPI NATIVI DI PYTHON

Al contrario della maggior parte dei linguaggi di programmazione (e come è evidente dai due esempi appena fatti), Python non richiede di *dichiarare* esplicitamente le variabili. Non è necessario, cioè, specificare il tipo della variabile prima della sua inizializzazione; inoltre, come vederemo, una variabile può cambiare tipo all'interno di un programma. Nonostante questo, ogni variabile, ad un determinato istante, ha un tipo ben preciso, cui si può accedere mediante la funzione `type()`, come mostrato nell'esempio seguente.

https://bitbucket.org/lbaldini/computing/src/tip/examples/built_in_types.py

```
# Initialize some variables...
n = 2
x = 2.
s = 'Hello'
l = [1, 2., 'three']
d = {'I. Newton': 0, 'A. Einstein': 0.5, 'L. Landau': 2}

# ...and print them to the standard output.
print(n, type(n))
print(x, type(x))
print(s, type(s))
print(l, type(l))
print(d, type(d))

[Output]
(2, <type 'int'>)
(2.0, <type 'float'>)
('Hello', <type 'str'>)
([1, 2.0, 'three'], <type 'list'>)
('L. Landau': 2, 'A. Einstein': 0.5, 'I. Newton': 0, <type 'dict'>)
```

Nel seguito di questo capitolo illustreremo le proprietà più importanti di alcuni dei tipi di variabile offerti da Python: numeri interi ed in virgola mobile, stringhe, liste e dizionari.

3.2 VARIABILI NUMERICHE

Python fornisce quattro tipi numerici: due per i numeri interi (`int` e `long`), uno per i numeri in virgola mobile (`float`), ed uno per i numeri complessi (`complex`). Ignoreremo del tutto questi ultimi e torneremo brevemente sulla distinzione tra i due tipi di interi, che per il momento è irrilevante, nel capitolo 7.

L'assegnazione del tipo avviene automaticamente all'atto dell'inizializzazione della variabile, con il punto (`.`) che funge da separatore decimale.

https://bitbucket.org/lbaldini/computing/src/tip/examples/numeric_types.py

```
1  # Integers...
2  i = 3
3  print(i)
4
5  # ...and floating point numbers.
6  x = 3.
7  y = 3.0
8  z = 3.6
9  pi = 3.1415
10 n_avogadro = 6.02e23
11 print(x, y, pi, n_avogadro)
12
13 # Conversion between types (also known as casting).
14 print(float(i))
15 print(int(z))
16 print(int(n_avogadro))
17
18 [Output]
19 3
20 (3.0, 3.0, 3.1415, 6.02e+23)
21 3.0
22 3
23 60199999999999995805696
```

I numeri in virgola mobile possono essere inizializzati sia in notazione scientifica (come alle linee 6, 7 ed 8) che in notazione ingegneristica (cioè con le potenze di 10, come mostrato alla linea 9). Le definizioni alle linee 6 e 7 sono del tutto identiche.

Le funzioni `int()` e `float()` permettono di convertire numeri in virgola mobile in numeri interi e viceversa—notiamo per completezza che `int()` non arrotonda all'intero più vicino, ma ritorna la parte intera dell'argomento. Chi fosse sorpreso dall'output alla linea 23 troverà informazioni aggiuntive sulle sottigliezze dell'aritmetica in virgola mobile nel capitolo 7.

3.3 LISTE E TUPLE

In Python una lista (`list`) è essenzialmente un contenitore le cui dimensioni possono variare dinamicamente a seconda del contenuto. Come mostrato nell'esempio seguente, questo contenuto non deve necessariamente essere omogeneo (nel senso dei tipi delle variabili), anche se noi non avremo molte occasioni di sfruttare questa flessibilità.

https://bitbucket.org/lbaldini/computing/src/tip/examples/list_init.py

```
1  # Initialize an empty list.
2  l = []
3  print(l, len(l))
4
5  # Append an element at the end of the list.
6  l.append(1)
7
8  # Concatenate two lists.
9  l += [2.0, 'tre']
10 print(l, len(l))
11
12 # Create a list in place.
13 l = [1, 2.0, 'tre']
14 print(l, len(l))
15
16 [Output]
```

```
([], 0)
([1, 2.0, 'tre'], 3)
([1, 2.0, 'tre'], 3)
```

Dal punto di vista delle strutture dei dati, la lista preserva l'ordine di inserimento e permette di indirizzare il contenuto per *indice* (partendo da zero).

https://bitbucket.org/lbaldini/computing/src/tip/examples/list_indexing.py

```
# Initialize a list.
l = [0, 1, 4, 9, 16]
print(l)

# Indexing by index (i.e., position in the list).
print(l[0], l[4], l[-1], l[-2])

# Modify an elemeny of the list.
l[0] = -1
print(l)

# Slicing (this is cool, isn't it?).
print(l[:2], l[2:], l[1:3])

[Output]
[0, 1, 4, 9, 16]
(0, 16, 16, 9)
[-1, 1, 4, 9, 16]
([-1, 1], [4, 9, 16], [1, 4])
```

Per completezza notiamo che Python offre, trai suoi tipi di dati, anche una lista *immutabile* che si chima tupla (`tuple`) e che si inizializza utilizzando parentesi tonde anziché quadre. Le interfacce sono identiche a quelle delle liste, a parte il fatto che i singoli elementi non possono essere modificati dopo che si è instanziato l'oggetto. Si tratta di una differenza tutto sommato sottile su cui non insistiamo oltre.

3.4 STRINGHE

Le stringhe di testo si rappresentano in Python con il tipo `str`. Ne abbiamo incontrato un esempio nel nostro programma di "Hello World!". Una stringa è essenzialmente una sequenza ordinata di caratteri e le interfacce che essa fornisce non sono troppo diverse da qualle di una lista.

https://bitbucket.org/lbaldini/computing/src/tip/examples/string_interfaces.py

```
# Initialize a string.
s = 'Hello '
print(s, len(s))

# Concatenate two strings.
s += "World!"
print(s, len(s))

# Indexing and slicing.
print(s[3], s[5:])

[Output]
('Hello ', 6)
('Hello World!', 12)
('l', ' World!')
```

Come mostrato alle linee 2 e 6, le stringhe di testo si possono racchiudere indifferentemente tra apici singoli (') e doppi apici (" ")—la differenza fondamentale è che gli apici singoli si possono usare come parte della stringa tra doppi apici e viceversa.

3.5 DIZIONARI

Vi sono situazioni in cui l'ordine di inserimento di un dato in una struttura dati non ha particolare importanza. Quel che è importante è essere in grado di recuperare in modo efficiente il dato stesso in base ad una *chiave* di accesso. Se dovessimo, ad esempio, mantenere un archivio elettronico dei libri in una biblioteca, presumibilmente non utilizzeremmo una lista a cui appendere ogni libro nel momento in cui viene acquistato, perché per cercarlo in seguito saremmo costretti ogni volta a scorrere la lista fino alla sua posizione.

Per queste situazioni Python offre il tipo standard dizionario (`dict`), un insieme di coppie chiave, valore indicizzato per chiave.

https://bitbucket.org/lbaldini/computing/src/tip/examples/dict.py

```
1  # Initialize a (chaotic and useless) dictionary
2  d = {'Hello': 'world', 1: 2, 'Hi': [0]}
3
4  # Access the values by key.
5  print(d['Hello'])
6  print(d[1])
7  print(d['Hi'])
8
9  [Output]
10 world
11 2
12 [0]
```

(Notate che non ci sono restrizioni di sorta sui tipi utilizzabili come chiavi o valori: possono essere qualsiasi cosa, incluso dizionari stessi).

I dizionari sono uno strumento estremamente potente e, anche se non avremo occasione di utilizzarlli pesantemente, li abbiamo menzionati per completezza.

3.6 ALTRI TIPI

La breve carrellata che abbiamo fatto fino a questo momento non esaurisce i tipi nativi di Python. Ve ne sono altri per certi versi affini a quelli che abbiamo visto (ad esempio i numeri complessi) ed altri decisamente più esotici—che sono i programmatori più avanzati si trovano a dover utilizzare direttamente.

Vale la pena di introdurre qui l'oggetto `None`, che è utilizzato spesso in Python per segnalare l'assenza di un altro valore.

https://bitbucket.org/lbaldini/computing/src/tip/examples/none.py

```
1  a = None
2  print(a)
3  print(type(a))
4
5  [Output]
6  None
7  <type 'NoneType'>
```

La lista completa dei tipi è disponibile a `http://docs.python.org/3/library/types.html`. Per noi è arrivato il momento di andare oltre.

3.7 AIUTO! CHE SUCCEDE?

(A rigore questo argomento non è legato in senso stretto ai tipi, ma...) Siamo arrivati fino a questo punto senza nemmeno un errore. La vita reale sarà sicuramente meno fortunata. Chiunque abbia un minimo di esperienza di programmazione sa che il programmatore sbaglia continuamente. A volte si tratta di errori banali (ad esempio errori di battitura), altre volte di errori più profondi (ad esempio errori logici del programma).

Le conseguenze degli errori variano da situazione a situazione. Se siamo sfortunati l'errore non causa un *crash* del programma e semplicemente otteniamo una risposta sbagliata. Se è chiaramente sbagliata, poco male: dobbiamo trovare l'errore e ricominciare; se non è ovviamente sbagliata può essere anche estremamente difficile rendersi conto che cè qualcosa che non va—e questi sono i casi più complicati. Ma se siamo fortunati, l'interprete arresta l'esecuzione del programma e ci dice esattamente cosa è andato storto, come nel caso seguente, in cui cerchiamo di accedere all'undicesimo elemento di una lista di lunghezza 4.

https://bitbucket.org/lbaldini/computing/src/tip/examples/err.py

```
l = [1, 2, 3, 4]
print(l[10])

[Output]
Traceback (most recent call last):
  File "/data/work/teaching/computing/examples/err.py", line 2, in <module>
    print(l[10])
IndexError: list index out of range
```

In questi casi il miglior consiglio è: non disperare e leggere *attentamente* il messaggio d'errore!

4

FUNZIONI

Le variabili non sono l'unica cosa che serve per memorizzare e manipolare dati. Ogni volta che si deve ripetere più volte un insieme specifico di operazioni (magari su variabili diverse) è utile incapsulare la funzionalità in una *funzione*. Una funzione è essenzialmente un frammento di codice dedicato ad un compito ben preciso.

https://bitbucket.org/lbaldini/computing/src/tip/examples/funcs.py

```
1  def square(x):
2      """ Return the square of the argument x.
3      """
4      return x*x
5
6  a = square(2)
7  b = square(2)
8  print(a, b)
9
10 [Output]
11 (4, 4)
```

In questo caso la nostra funzione calcola semplicemente il quadrato di un numero che passiamo come *argomento* (tra parentesi tonde alla riga 1). Le funzioni possono accettare uno o più argomenti e possono restituire un valore (tramite il comando `return` alla linea 2).

Per completezza: se una funzione non include esplicitamente una istruzione `return`, il valore restituito di default è `None`.

https://bitbucket.org/lbaldini/computing/src/tip/examples/funcs_none.py

```
1  def square(x):
2      """ Return the square of a real number x.
3      """
4      x2 = x*x
5      # Opss... forgot the return!
6
7  a = square(2)
8  print(a)
9
10 [Output]
11 None
```

4.1 IL RUOLO DELL'INDENTAZIONE

I frammenti di codice appena mostrati illustra come *l'indentazione*[1] delle linee sia usata in Python per determinare il raggruppamento logico del codice. Con riferimento al primo esempio di questo

1 L'indentazione delle linee è irrilevante nella maggior parte dei linguaggi di programmazione. Python e FORTRAN, tra gli altri, fanno eccezione.

capitolo, il fatto che la linea 2 contenga al suo inizio 4 spazi dice che essa è parte della funzione definita alla linea 1. La linea 4 non ha spazi al suo inizio per cui è al di fuori della funzione.

Vedremo altri esempi di indentazione quando parleremo di *cicli* ed espressioni condizionali.

Fixme: Aggiungere args e kwargs.

5

CONTROLLO DEL FLUSSO DEL PROGRAMMA

In questa sezione introduciamo brevemente due tecniche di programmazione fondamentali: le espressioni condizionali e i *cicli* (o *loop*).

5.1 ESPRESSIONI CONDIZIONALI

Ci sono casi in cui è necessario che un programma esegua azioni diverse a seconda del contesto (ad esempio di un input esterno o del valore di una variabile ad un certo punto dell'esecuzione).

Nel frammento successivo l'uso delle espressioni condizionali è illustrato nell'implementazione di una semplice funzione che restituisce il valore assoluto di un numero reale.

https://bitbucket.org/lbaldini/computing/src/tip/examples/flow_if.py

```
def absvalue(x):
    """ Poor man's implementation of the absolute value.

    python provides an abs() function that does the job (and does
    it better, too), so there is no need to reinvent the wheel, here.
    """
    if x >= 0:
        return x
    else:
        return -x

# Test that it actually works.
print(absvalue(2.0), absvalue(0.0), absvalue(-2.0))

[Output]
(2.0, 0.0, 2.0)
```

(Va da sé che nella vita reale non vi è nessuna necessità di re-implementare una funzione che già esiste—e vi sono invece molti rischi!)

Per completezza le espressioni condizionali possono includere anche uno o più blocchi `elif`:

https://bitbucket.org/lbaldini/computing/src/tip/examples/flow_if2.py

```
def sign(x):
    """ Print out the sign of the argument x.
    """
    if x > 0:
        return 'Positive'
    elif x == 0:
        return 'Zero'
    else:
        return 'Negative'

print(sign(2), sign(0), sign(-2))

```

```
[Output]
('Positive', 'Zero', 'Negative')
```

Notate alla linea 6 l'operatore == che testa l'eguaglianza tra due valori (e che è completamente diverso dall'operazione di assegnazione =).

5.2 CICLI FOR

I cicli `for` sono un ingrediente fondamentale per il controllo del flusso del programma in qualsiasi linguaggio. Lo illustriamo con un semplice esempio che impiega le liste.

https://bitbucket.org/lbaldini/computing/src/tip/examples/flow_for1.py

```
# Fill a list with the squares of the integers between 1 and 10.
# Note the usage of the range() function.
l = []
for i in range(1, 11):
    l.append(i**2)
print(l)

# Loop directly over a list.
for scientist in ['Newton', 'Einstein', 'Bohr', 'Dirac']:
    print(scientist)

[Output]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
Newton
Einstein
Bohr
Dirac
```

Per inciso, Python permette anche forme più concise (che utilizzeremo spesso nel seguito) come:

https://bitbucket.org/lbaldini/computing/src/tip/examples/flow_for2.py

```
# This is more concise (thought not necessarily faster).
l = [i**2 for i in range(1, 11)]
print(l)

[Output]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Notiamo l'uso della funzione nativa `range` di Python, che permette di iterare su una serie di interi:

https://bitbucket.org/lbaldini/computing/src/tip/examples/range.py

```
# Use cases of the range() native function.
a = [i for i in range(5)]
b = [i for i in range(1, 5)]
c = [i for i in range(1, 8, 2)]

print(a)
print(b)
print(c)

[Output]
[0, 1, 2, 3, 4]
[1, 2, 3, 4]
[1, 3, 5, 7]
```

Il calcolo del fattoriale di un numero intero costituisce un'applicazione elementare del ciclo `for`:

https://bitbucket.org/lbaldini/computing/src/tip/examples/factorial_for.py

```
def factorial(n):
    """ Return the factorial of the argument n.
    """
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result

print(factorial(0), factorial(1), factorial(5))

[Output]
(1, 1, 120)
```

5.3 CICLI WHILE

L'istruzione `while` permette di eseguire cicli con una logica leggermente diversa che, a seconda della situazione, può risultare più comoda da usare del `for`.

https://bitbucket.org/lbaldini/computing/src/tip/examples/flow_while.py

```
# Fill a list with a while loop.
l = []
n = 3
while n >= 0:
    l.append(n)
    n -= 1
print(l)

[Output]
[3, 2, 1, 0]
```

Il calcolo del fattoriale si può implementare con un ciclo `while` anziché un ciclo `for`:

https://bitbucket.org/lbaldini/computing/src/tip/examples/factorial_while.py

```
def factorial(n):
    """ Return the factorial of the argument n.
    """
    result = 1
    while n > 1:
        result *= n
        n -= 1
    return result

print(factorial(0), factorial(1), factorial(5))

[Output]
(1, 1, 120)
```

Notiamo infine, per inciso, che la flessibilità con cui Python permette di utilizzare le funzioni permette di creare costrutti interessanti come il seguente, in cui la funzione `factorial` è chiamata ricorsivamente dentro se stessa.

https://bitbucket.org/lbaldini/computing/src/tip/examples/factorial.py

```
def factorial(n):
    """ Return the factorial of the argument n.
    """
```

```
    if n == 0:
        return 1
    else:
        return n*factorial(n - 1)

print(factorial(0), factorial(1), factorial(5))

[Output]
(1, 1, 120)
```

Interessante, no?

6

INPUT/OUTPUT

L'Input/Output (I/O) è sostanzialmente la comunicazione tra il computer ed il mondo esterno. L'istruzione `print()`, che abbiamo utilizzato copiosamente, è un esempio di istruzione di output—nel caso in questione output su terminale. Esiste anche una corrispondente funzione `input()` per l'input dal terminale.

Per i nostri scopi è più interessante l'I/O su file. Il seguente frammento legge un file di testo (contenente colonne di dati separati da spazi o tab) e memorizza una riga alla volta come una lista di numeri in virgola mobile.

https://bitbucket.org/lbaldini/computing/src/tip/examples/input_data.py

```
# Open a text file and convert the row content to float.
for line in open('input_data.txt'):
    # Ignore lines starting with a # (they are comments).
    if not line.startswith('#'):
        # Got a valid data line.
        row = [float(item) for item in line.split()]
        # Print the data (in real life you would actually use them).
        print(row)


[Output]
[1.0, 6.08]
[2.0, 2.34]
[3.0, 9.34]
```

Notate che ignoriamo le linee che iniziano con un carattere # (che rappresentano dei commenti nel file di ingresso).

Il frammento successivo permette invece di scrivere dei numeri in un file di testo:

https://bitbucket.org/lbaldini/computing/src/tip/examples/output_data.py

```
# Open the output file in write ('w') mode.
outputFile = open('output_data.txt', 'w')
for i in range(10):
    outputFile.write('%d\t%d\n' % (i, i**2))
# Close the output file.
outputFile.close()
```

Lasciamo al lettore il compito di indovinare come appare il file di uscita. Ci limitiamo a notare la formattazione del testo in uscita alla linea 4, in cui `\t` rappresenta un TAB e `\n` un'andata a capo (EOL).

Fixme: Mettere un esempio di errore per problemi di path.

7

NUMERI E LORO RAPPRESENTAZIONE

Abbiamo già visto nel capitolo 3 cosa può accadere convertendo un numero in virgola mobile in un numero intero:

https://bitbucket.org/lbaldini/computing/src/tip/examples/avogadro.py

```
n_avogadro = 6.02e23
print(n_avogadro, int(n_avogadro))

[Output]
(6.02e+23, 601999999999999995805696L)
```

(In questo caso non sarebbe stato lecito aspettarsi che il risultato della conversione fosse esattamente 602000000000000000000?)

Questo semplice esempio mostra come l'aritmetica in virgola mobile su un calcolatore sia, per sua natura, *approssimata*. La ragione ultima è costituta dal fatto che, in qualsiasi base, esistono numeri reali che non possiedono uno sviluppo finito e che dunque non possono essere espressi *esattamente* con un numero finito di cifre nella consueta notazione posizionale.

In questo capitolo ci occupiamo brevemente della rappresentazione dei numeri in un calcolatore e dei problemi numerici più comuni che si incontrano nell'aritmetica in virgola mobile.

7.1 IL SISTEMA DI NUMERAZIONE BINARIO

Quando scriviamo il numero 127 (in notazione posizionale in base 10), quello che intendiamo è:

$$(127)_{10} = 1 \times 10^2 + 2 \times 10^1 + 7 \times 10^0.$$

(Abbiamo indicato esplicitamente la base per evitare ambiguità). In realtà non vi è niente di magico nella base 10. Qualsiasi numero intero maggiore $n > 1$ funziona altrettanto bene[1] e, per $n = 2$, 8, 16 e 60 si hanno i sistemi di numerazione (largamente usati in alcuni contesti) binario, ottale, esadecimale e sessagesimale.

La quasi totalità dei computer funziona in logica binaria. (Per inciso, questo deriva largamente dal fatto che i dispositivi di memorizzazione più diffusi si basano, a livello elementare, sul *flip-flop*—che è un circuito elettronico a due stati.) Convertire dal sistema binario a quello decimale è banale:

$$(1101)_2 = 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^0 = (13)_{10}$$

Nel linguaggio tipico dei computer il singolo elemento di informazione (che può valere 0 o 1) si chiama *bit*. Una *parola* di n bit può assumere 2^n valori differenti. Così con 2 bit si possono esprimere i quattro valori $(00)_2$, $(01)_2$, $(10)_2$ ed $(11)_2$, mentre con 64 bit si possono esprimere 18446744073709551616 valori differenti.

1 In realtà non è necessario essere così restrittivi: i sistemi di numerazione in base negativa, immaginaria o complessa, ad esempio, hanno alcune proprietà degne di nota.

7.2 NUMERI INTERI

La rappresentazione dei numeri interi è relativamente semplice. Come abbiamo detto, una parola di n bit può rappresentare tutti i numeri da 0 a $2^n - 1$ compresi (per un totale di 2^n valori). Se si vogliono rappresentare anche numeri negativi, il segno, necessita di uno dei bit di informazione.

Molti linguaggi di programmazione offrono un controllo granulare sui numeri interi, fornendo tipi predefiniti a 8, 16, 32 e 64 bit, nella versione con segno (*signed*) e senza (*unsigned*). Ad esempio un *unisgned* ad 8 bit può contenere gli interi da 0 a 255, mentre un *signed* ad 8 bit può contenere gli interi da -128 a 127. Tutto ciò permette di ottimizzare l'uso della memoria ma, allo stesso tempo, richiede una qualche attenzione nel far sì che, nel flusso del programma, una variabile di un determinato tipo non ecceda mai i limiti del tipo stesso (nel qual caso si hanno problemi di *overflow* o *underflow*).

In Python la gestione dei tipi interi è completamente trasparente all'utente. Il linguaggio offre due tipi di interi, entrambi con segno: uno (`int`) a 32 o 64 bit (a seconda della piattaforma) ed uno (`long`) con profondità *illimitata*[2]. Il passaggio da un tipo all'altro avviene automaticamente all'atto dell'inizializzazione oppure quando il valore di una determinata variabile diventa troppo grande per essere contenuto in un `int`. Ad esempio, su un sistema a 64 bit il tipo `int` può assumere valori compresi tra $-(2^{63} - 1)$ e $(2^{63} - 1)$ (ricordate il bit necessario a codificare il segno).

https://bitbucket.org/lbaldini/computing/src/tip/examples/maxint.py

```
1  import sys
2
3  print(sys.maxint)
4  print(2**63 - 1)
5  print(repr(2**62))
6  print(repr(2**63))
7
8  [Output]
9  2147483647
10 9223372036854775807
11 4611686018427387904L
12 9223372036854775808L
```

Nel frammento di codice qui sopra la `L` alla fine della linea 12 indica che internamente il numero in questione è codificato come un `long` anziché un `int`.

7.3 NUMERI IN VIRGOLA MOBILE

Il discorso è più complicato (ed interessante) per i numeri in virgola mobile (cioè quelli che potremmo essere tentati di chiamare numeri *reali*). Essi sono di norma rappresentati internamente nella forma

$$x = \pm m \times 2^{\pm e}, \tag{2}$$

in cui m prende il nome di *mantissa* ed e è chiamato esponente (i due segni di $\pm$ servono a ricordare il bit di informazione a loro riservato). Ancor prima di entrare nel dettaglio del numero di bit riservati alla mantissa ed all'esponente, è chiaro che esistono numeri che non possono essere rappresentati *esattamente* nella forma (2), a meno di non avere infiniti bit disponibili. Così come la frazione decimale 1/3 non può essere rappresentata esattamente in notazione posizionale in base 10 con un numero finito di cifre, la frazione decimale 1/10 non ha uno sviluppo binario finito e non è dunque rappresentabile esattamente in un calcolatore a logica binaria.

https://bitbucket.org/lbaldini/computing/src/tip/examples/number_representation.py

```
1  a = 0.1
2  print('%.55f' % a)
```

2 Ovviamente non si può eccedere la quantità di memoria fisica disponibile sul calcolatore

```
3
4 [Output]
5 0.1000000000000000055511151231257827021181583404541015625
```

(Notate che in questo frammento abbiamo *formattato* la nostra variabile a richiedendo 55 cifre per il valore stampato sul terminale).

Il valore stampato alla linea 5 è il numero *più vicino* a $(0.1)_{10}$ rappresentabile in logica binaria sul calcolatore su cui è stato eseguito il programma. Come vedremo brevemente nel seguito, il carattere intrinsecamente *approssimato* dell'aritmetica in virgola mobile ha conseguenze interessanti e di grande rilevanza pratica.

7.3.1 *Aritmetica elementare in virgola mobile*

Nessuno avrà dubbi sul fatto che l'uguaglianza seguente

$$\frac{1}{10} + \frac{2}{10} = \frac{3}{10} \tag{3}$$

sia palesemente vera. Ebbene: su un calcolatore digitale anche l'aritmetica elemntare può riservare sorprese.

https://bitbucket.org/lbaldini/computing/src/tip/examples/rounding_errors.py

```
1 a = 0.1
2 b = 0.2
3 c = a + b
4 print('%.55f' % c)
5 print(c == 0.3)
6
7 [Output]
8 0.3000000000000000044408920985006261616945266723632812500 0
9 False
```

In questo caso, per i motivi che abbiamo delineato sopra, l'equivalente dell'uguaglianza (3) risulta falso. Come vediamo alla linea 5, se nel flusso del nostro programma avessimo un'espressione condizionale basata sull'eguaglianza tra `c` e `0.3`, questa uguaglianza sarebbe *non* verficata. Quasi certamente questo *non* sarebbe il comportamento voluto. Ne segue che, mentre l'uguaglianza tra numeri interi è ben definita, *non è buona pratica utilizzare l'uguaglianza tra due numeri in virgola mobile in un'espressione condizionale, a meno di non prendere le dovute precauzioni.*

Consideriamo un'altra espressione chiaramente vera:

$$(x + \varepsilon) > x \quad \forall \varepsilon > 0. \tag{4}$$

Anche in questo caso la *traduzione* nel linguaggio di un calcolatore digitale riserva (apparentemente) sorprese.

https://bitbucket.org/lbaldini/computing/src/tip/examples/test_inequality.py

```
1 x = 6.02e23
2 epsilon = 6.67384e-11
3 s = x + epsilon
4 print('%.10f' % x)
5 print('%.10f' % s)
6 print(s > x)
7
8 [Output]
9 601999999999999995805696.0000000000
10 601999999999999995805696.0000000000
11 False
```

Qui il problema è che `epsilon` è più piccolo dell'errore insito nella rappresentazione di `x`, da cui impariamo che *serve cautela nel sommare un numero molto grande ad un numero molto piccolo.*

Sappiamo dalle scuole elementari che la somma di numeri reali soddisfa la proprietà associativa:

$$(a + b) + c = a + (b + c). \tag{5}$$

Ebbene: anche questo è in generale falso nell'aritmetica in virgola mobile su un calcolatore digitale.

https://bitbucket.org/lbaldini/computing/src/tip/examples/test_associativity.py

```
a = 1.0e8
b = -0.9999999e8
c = 0.4e-7
print('%.55f' % ((a + b) + c))
print('%.55f' % (a + (b + c)))
print(((a + b) + c) == (a + (b + c)))

[Output]
10.0000000399999997569011611631140112876892089843750000000
10.0000000447034835815429687500000000000000000000000000000
False
```

Cioè in generale *le proprietà elementari della quattro operazioni non valgono necessariamente.*

7.3.2 *Operazioni più avanzate in virgola mobile*

Come ultimo esempio prendiamo un'uguaglianza che probabilmente desterebbe scarso interesse su un testo di analisi matematica:

$$\log_{10}\left(10^b\right) = b. \tag{6}$$

Anche in questo caso le cose sono problematiche sul nostro calcolatore

https://bitbucket.org/lbaldini/computing/src/tip/examples/overflow.py

```
import math

b = 1555.5
print(math.log10(10**b))

[Output]
Traceback (most recent call last):
  File "/data/work/teaching/computing/examples/overflow.py", line 4, in <module>
    print(math.log10(10**b))
OverflowError: (34, 'Numerical result out of range')
```

poiché il numero che stiamo cercando di rappresentare è troppo grande. In generale *occorre cautela nel maneggiare numeri molto grandi o numeri molto piccoli.*

8

MATEMATICA ELEMENTARE

Entriamo nel vivo della discussione con una breve panoramica delle principali funzioni matematiche in Python.

8.0.3 *Aritmetica elementare*

Il frammento seguente illustra l'utilizzo degli operatori per l'aritmetica elementare in Python.

https://bitbucket.org/lbaldini/computing/src/tip/examples/aritmetics.py

```
# Four basic operations.
print(1 + 4)
print(12 - 6)
print(6*8)
print(1.54/5.65)

# Integer division (mind this is different in python 3.x vs. 2.x).
print(15/2)
print(15 % 2)

# Powers
print(2**8)

[Output]
5
6
48
0.272566371681
7
1
256
```

Il tutto dovrebbe essere autoesplicativo. La divisione tra interi merita un piccolo commento: nella stragrande maggioranza dei linguaggi di programmazione l'operatore di divisione `/` restituisce *la parte intera* nella divisione tra due interi (mentre l'operatore `%` restituisce il resto della divisione). Questo è vero anche nella versione 2 di Python, ma non nella versione 3 (in cui `/` restituisce un numero in virgola mobile e `//` restituisce la parte intera). In generale è buona regola prestare attenzione nella divisione tra interi (e convertire uno dei due in `float` quando necessario).

8.0.4 *Il modulo `math`*

Per le funzione matematiche più avanzate è necessario utilizzare un apposito *modulo*, `math`, parte della libreria standard[1] di Python. Si tratta del primo modulo di libreria che incontriamo in questa

1 La libreria standard di Python è organizzata in una serie di moduli che vanno importati all'occorrenza per ottimizzare l'utilizzo della memoria a seconda delle esigenze del programma.

nostra rassegna, ed i moduli si caricano (o, tecnicamente, si *importano*) con il comando `import`, come illustrato nel frammento seguente.

https://bitbucket.org/lbaldini/computing/src/tip/examples/mathematics.py

```
import math

# A couple of mathematical functions.
a = math.sin(math.pi/4)
b = math.log(100, 10)

# This will give the integral of a standard gaussian between 0 and 1,
# i.e., a half of the usual ~68%.
c = 0.5*math.erf(1/math.sqrt(2.))

print(a, b, c)

[Output]
(0.7071067811865475, 2.0, 0.341344746068543)
```

Il modulo `math` mette a disposizione le seguenti funzioni, di cui non dovrebbe essere difficile intuire il significato:

```
acos, acosh, asin, asinh, atan, atan2, atanh, ceil, copysign, cos, cosh, degrees, e, erf,
erfc, exp, expm1, fabs, factorial, floor, fmod, frexp, fsum, gamma, hypot, isinf, isnan,
ldexp, lgamma, log, log10, log1p, modf, pi, pow, radians, sin, sinh, sqrt, tan, tanh, trunc
```

9

NUMERI PSEUDO-CASUALI E METODI MONTE CARLO

La parola *simulazione* è ormai di uso comune. Spesso i fisici fanno simulazioni di fenomeni naturali. Nella maggio parte dei casi, l'ingrediente fondamentale di una simulazione è un generatore di numeri casuali (o generatore *random*)—nel qual caso si parla di metodi Monte Carlo—per cui in questo capitolo introduciamo brevemente l'argomento.

Cominciamo la discussione con un esempio concreto. Supponiamo di voler calcolare la superficie di un cerchio inscritto in un quadrato di lato unitario (senza sapere, ovviamente, che la risposta è $\pi/4$). Operativamente potremmo pensare di disegnare un quadato di lato $l = 1$ m (ed il cerchio inscritto) sul pavimento e di lasciar cadere dei coriandoli in modo che essi coprano nel modo più uniforme possibile la superficie del quadrato stesso (che sappiamo essere 1 m^2). Il rapporto tra il numero di coriandoli n_c che sono caduti dentro il cerchio ed il numero totale N di coriandoli che sono caduti dentro il quadrato tenderà (per $N \to \infty$) al rapporto tra le due aree. Quindi, per ogni N finito, il procedimento fornisce una stima dell'area del cerchio—ed una stima mediamente tanto più accurata quanto più N è grande.

Ecco allora la connessione con l'argomento del capitolo: se avessimo a disposizione un meccanismo per generare numeri casuali indipendenti distribuiti *uniformemente* tra -0.5 e 0.5, potremmo facilmente simulare il processo, come mostrato in figura 1, e dare una stima dell'area del cerchio—o, equivalentemente, di π.

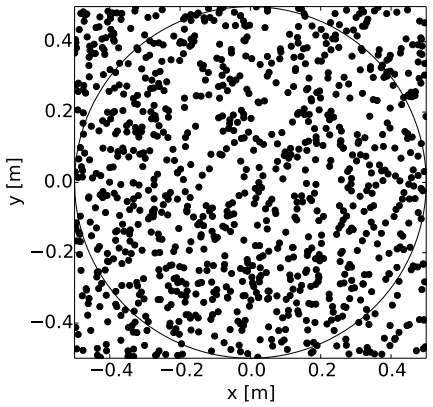

FIGURA 1. Rappresentazione grafica di $N = 1000$ punti casuali generati uniformemente (vedremo tra un attimo come) entro un quadrato di lato unitario. In questa particolare realizzazione i punti che finiscono all'interno del cerchio inscritto sono esattamente $n_c = 778$, il che dà una stima di $\pi \approx 4 \cdot 778/1000 = 3.11$.

L'utilizzo dei numeri casuali inizia negli anni '20 del secolo scorso con tabelle (cartacee) di cifre *random* precompilate a partire da varie fonti (e talvolta dalle dubbie proprità statistiche). Poiché tabelle di grandi dimensioni non sono agevoli da manipolare, con l'avvento dei calcolatori digitali si è cominciato a studiare la possibilità di produrre numeri casuali (o, meglio, pseudo-casuali) sfruttando l'aritmetica elementare.

9.1 ALGORITMI PER LA GENERAZIONE DI NUMERI PSEUDO-CASUALI

Una delle prime idee è dovuta a John Von Neumann e risale alla metà degli anni '40 del '900. L'algoritmo di base è molto semplice: dato un numero intero di n cifre[1], prendiamo le n cifre centrali del suo quadrato come elemento successivo della nostra sequenza. Supponiamo ad esempio di voler generare numeri casuali interi a 6 cifre. Se partiamo da 851846, elevando al quadrato otteniamo 725641607716, le cui 6 cifre centrali sono 641607; questo è il prossimo numero nella nostra sequenza. Procedendo iterativamente otteniamo 659542, 995649, e così via (prima di andare oltre, notiamo che dividendo per 10^6 otteniamo una sequenza di numeri reali tra 0 ed 1, che è più vicina al nostro problema di partenza).

Perché questo procedimento dovrebbe dare una sequenza di numeri casuali? L'idea di base è che, dopo l'elevamento al quadrato, le cifre centrali, in un certo senso, non hanno memoria del numero da cui siamo partiti. Ovviamente possiamo legittimamente porci la domanda: come possiamo definire casuale una sequenza in cui ogni numero è determinato esattamente dal precedente? La risposta è: la sequenza non è certamente casuale, ma se si comporta *come se lo fosse*, questo è sufficiente per i nostri scopi. (Anzi, ci sono situazioni in cui la possibilità di generare più volte la stessa identica sequenza è utile.)

Il metodo di Von Neumann (generalmente noto con il nome di *middle square*) per numeri interi ad n cifre si può *matematizzare* come:

$$X_{n+1} = (X_n^2 / 10^{\frac{n}{2}}) \bmod 10^n. \tag{7}$$

(provate per convircervene). In questo contesto / indica la divisione intera e mod il resto della divisione intera; il primo termine della sequenza (X_0) prende il nome di *seed* (seme). Il frammento di codice seguente illustra un semplice esempio per $n = 2$.

https://bitbucket.org/lbaldini/computing/src/tip/examples/random_middle_square.py

```
1  # Implementation of a toy base 10 middle-square generator (Von Neumann, 1946),
2  # generating two-digits "random" numbers (between 0 and 99).
3
4  # Define the seed.
5  x = 81
6  print(x)
7
8  # Generate 10 numbers.
9  for i in range(10):
10     x = (x**2 // 10) % 100
11     print(x)
12
13 [Output]
14 81
15 56
16 13
17 16
18 25
19 62
20 84
21 5
22 2
23 0
24 0
```

Il cuore dell'algoritmo è alla linea 10, in cui // indica la divisione intera (sia nelle versioni più recenti della serie 2x di che in tutta la serie 3x di Python) e % indica il modulo (o resto della divisione intera).

1 Per semplicità qui lavoreremo in notazione decimale, ma è utile tenere in mente che, internamente al calcolatore, l'aritmetica è tipicamente eseguita in rappresentazione binaria.

9.1.1 *Aspetti generali dei generatori pseudo-casuali*

L'algoritmo descritto nella sezione precedente, benché di scarsa rilevanza pratica per i motivi che illustreremo tra un attimo, è importante perché illustra alcuni aspetti generali dei generatori pseudo-casuali. Il primo è l'idea di base di generare ricorsivamente gli elementi (interi) della sequenza

$$X_{n+1} = f(X_n) \tag{8}$$

a partire da un seme X_0. Detto X_{max} il valore massimo restituito da $f(X)$ ($10^n - 1$ nell'esempio precedente), possiamo semplicemente dividere X_n per X_{max}

$$U_n = \frac{X_n}{X_{max}} \tag{9}$$

per ottenere una sequenza di numeri reali tra 0 ed 1.

La prima conseguenza immediata della (8) è che il *periodo* della sequenza è necessariamente limitato. Ad un certo punto la nostra funzione f trasformerà l'elemento n-esimo in un numero che abbiamo già ottenuto in precedeza e, da quel punto in poi, la sequenza si ripeterà in un ciclo infinito. Questo è illustrato alle righe 23 e 24 dell'esempio precedente, in cui il numero 0 si ripete (e continuerebbe a ripetersi all'infinito, se andassimo avanti nella sequenza). Avere un periodo lungo è di enorme rilevanza pratica: nel momento in cui la sequenza comincia a ripetersi, smette di comportarsi *come se fosse casuale*, per cui il periodo costituisce un limite superiore al numero di elementi che si possono utilmente estrarre dalla sequenza. Questo è uno dei motivi per cui il metodo *middle square* non è di fatto utilizzato in pratica.

9.1.2 *Un esempio di rilevanza pratica*

Lo schema lineare congruenziale (*linear congruential method*), proposto all fine degli anni '40 del secolo scorso, è ancora oggi abbastanza popolare, e lo discutiamo brevemente. La definizione di base è

$$X_{n+1} = (aX_n + c) \bmod m \tag{10}$$

in cui il moltiplicatore a, l'incremento c ed il modulo m sono tre numeri fissi opportunamente scelti. (Ovviamente abbiamo anche bisogno di un seme X_0 per definire univocamente la sequenza.)

Il frammento seguente implementa un generatore *giocattolo* per numeri a due cifre decimali utilizzando questo schema. Notiamo, per inciso, che il suo periodo è il massimo possibile (100) per un generatore a due cifre decimali.

https://bitbucket.org/lbaldini/computing/src/tip/examples/random_linear_congruential.py

```
# Implementation of a toy linear congruential generator, generating
# two-digits "random" numbers (between 0 and 99).

# Define the seed and the other stuff.
x = 81
a = 11
c = 3
m = 73
print(x)

# Generate 10 numbers.
for i in range(10):
    x = (a*x + c) % m
    print(x)

```

```
[Output]
81
18
55
24
48
20
4
47
9
29
30
```

L'esempio che segue illustra un generatore realistico, utilizzabile in applicazioni pratiche, dotato di un periodo di $2^{64} \approx 1.8 \times 10^{19}$.

https://bitbucket.org/lbaldini/computing/src/tip/examples/linear_congruential_generator.py

```
m = 2**64
a = 6364136223846793005
c = 1442695040888963407
seed = 1

# Set the initial value to the seed.
x = seed

# Generate and print out a few pseudo-random numbers.
for i in range(5):
    x = (a*x + c) % m
    print(float(x)/(m - 1))

[Output]
0.423209170873
0.509407442884
0.648359393963
0.382863390508
0.795447749254
```

Vedremo tra un attimo che sia la libreria standard di Python che `numpy` offrono alternative preconfezionate più veloci ed affidabili.

9.2 IL MODULO RANDOM DI PYTHON

La libreria standard di Python possiede un modulo apposito (`random`) che è più che adeguato per la maggior parte delle applicazioni. L'uso di base è semplicissimo:

https://bitbucket.org/lbaldini/computing/src/tip/examples/random1.py

```
import random
# Fix the seed to a conventional value for reproducibility.
random.seed(1)

# Extract a random number from a uniform distribution between 0 and 1.
a = random.random()
# Now a gaussian distribution with mean = 10 and sigma = 2.
b = random.gauss(10, 2)
# And, finally, simulate the roll of a die.
c = random.randint(1, 6)
print(a, b, c)
```

```
12
13 [Output]
14 (0.13436424411240122, 11.95248589854377, 2)
```

L'unica cosa da notare in questo frammento è il fatto che, per ottenere un risultato riproducibile, abbiamo fissato ad un valore convenzionale (1 nel nostro caso) il *seed* del generatore pseudo-casuale; se non l'avessimo fatto Python avrebbe usato un *seed casuale* basato sul tempo di sistema ed avremmo ottenuto numeri diversi invocando due volte lo stesso programma.

Il seguente frammento mostra come generare una lista con numeri random, il che ci sarà utile in seguito.

https://bitbucket.org/lbaldini/computing/src/tip/examples/random2.py

```
1 import random
2 random.seed(1)
3
4 # Fill a list with random numbers from a uniform pdf between 0 and 1.
5 l = [random.random() for i in range(3)]
6 print(l)
7
8 [Output]
9 [0.13436424411240122, 0.8474337369372327, 0.763774618976614]
```

9.3 ALCUNE SEMPLICI APPLICAZIONI

Vediamo di seguito alcune semplici applicazioni delle sequenze pseudo-casuali.

9.3.1 *Calcolo di π*

L'esempio seguente fornisce una stima numerica del valore di π utilizzando il generatore di numeri pseudo-casuali fornito dalla libreria standard di Python, sfruttando l'idea con cui abbiamo aperto il capitolo: generiamo N numeri casuali in un quadrato di lato 1 e contiamo il numero n di punti che cadono nel cerchio inscritto (di raggio 1/2). Il rapporto n/N tende, per N grande, al rapporto tra le aree del cerchio e del quadrato:

$$\lim_{N\to\infty} \frac{n}{N} = \frac{\pi}{4}. \tag{11}$$

https://bitbucket.org/lbaldini/computing/src/tip/examples/random_pi.py

```
1  import math
2  import random
3  random.seed(1)
4
5  # Generate a lot of points in a square of side 2 and count how many
6  # of them fall into the inscribed circle of radius one.
7  N = 0
8  n = 0
9  for i in range(100000):
10     x = random.uniform(-0.5, 0.5)
11     y = random.uniform(-0.5, 0.5)
12     N += 1
13     if (x**2 + y**2) < 0.25:
14         n += 1
15
16 # Calculate our estimate of pi and the relative error (mind in
17 # python 2x this works because 4*n is executed first and returns
```

```
# a float, while n/N would return 0).
pi = 4.*n/N
err = abs((pi - math.pi)/math.pi)
print(pi)
print('%.3e' % err)

[Output]
3.13944
6.852e-04
```

Con la nostra particolare realizzazione di 100000 lanci otteniamo un errore relativo di $\sim 7 \times 10^{-4}$ sulla stima di π. Aumentando N (nel limite in cui la nostra sequenza casuale ha buone proprietà statistiche e non esaurisce il suo periodo) si può ottenere una stima accurata a piacimento del valore cercato.

9.3.2 *Il random walk in due dimensioni*

L'esempio seguente illustra un *random walk* in due dimensioni. Il modulo `turtle`, parte della libreria standard di Python permette di visualizzare in tempo reale la simulazione, sia pure in modo piuttosto rudimentale.

https://bitbucket.org/lbaldini/computing/src/tip/examples/random_walk.py

```
import random
random.seed(1)

import math

# This is simple module of the python standard library for generating
# graphics.
import turtle
turtle.title('2-d random walk')
turtle.setup(350, 350, 0, 0)

# Loop and generate a random walk in two dimension with fixed step.
x = 0.
y = 0.
step = 5.
for i in range(1000):
    angle = random.uniform(0, 2*math.pi)
    x += step*math.cos(angle)
    y += step*math.sin(angle)
    turtle.setposition(x, y)

# Save the canvas to file.
screen = turtle.getscreen()
screen.getcanvas().postscript(file = "random_walk.eps")
```

L'esempio può essere modificato banalmente per calcolare quantità interessanti, come ad esempio la distanza dall'origine delle coordinate ad ogni passo della simulazione (come deve crescere, in media?). Per completezza l'output grafico del programma è mostrato in figura 2.

FIGURA 2. Esempio di *random walk* in due dimensioni generato con il programma riportato qui sopra e visualizzato con il modulo `turtle` della libreria standard di Python.

10

ALCUNI ESEMPI PRATICI

In questo capitolo discutiamo alcune applicazioni pratiche che possono essere utili nelle esperienze di laboratorio.

10.1 CALCOLO DI MEDIA E VARIANZA CAMPIONE

Passiamo ad un esempio mutuato direttamente dalla statistica, cioè il calcolo della media e della deviazione standard di un campione di dati (in questo caso una lista di 100 campionamenti di una distribuzione Gaussiana in forma standard). Nell'esempio seguente seguiamo pedissequamente le definizioni:

$$m = \frac{1}{n}\sum_{i=1}^{n} x_i, \quad s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - m)^2 \tag{12}$$

eseguendo due cicli separati per il calcolo della media e quello della varianza.

https://bitbucket.org/lbaldini/computing/src/tip/examples/sample_stat1.py

```
import math

def get_stat(sample):
    """ Return the sample mean and variance for a list of values.

    This is done in two separate steps, i.e., we calculate the mean
    first, then the average.
    """
    # Calculate the mean (do not initialize mean as an integer!).
    mean = 0.
    for value in sample:
        mean += value
    mean /= len(sample)
    # Calculate the variance
    variance = 0.
    for value in sample:
        variance += (value - mean)**2.0
    variance /= len(sample) - 1
    # Return the two numbers.
    return mean, variance


# Generate a random vector of data.
import random
random.seed(1)
sample = [random.gauss(0, 1) for i in range(10000)]

# Calculate the sample statistics and print out mean and stdev.
mean, variance = get_stat(sample)
print(mean, math.sqrt(variance))
```

```
31
32 [Output]
33 (0.0030604261498986744, 0.9922712982269504)
```

Notiamo che le variabili `mean` e `variance`, nel programma, non rappresentano la media e la varianza campione a tutti i passi del programma, ma solo alla fine, dopo la divisione per n e $n-1$ (cosa assolutamente non inusuale da vedere in un programma). Da notare anche che è di fondamentale importanza la variabile `mean` come uno zero in virgola mobile (e non un intero). Perché?

Possiamo utilizzare qualche semplice passaggio algebrico elementare

$$\sum_{i=1}^{n}(x_i - m)^2 = \sum_{i=1}^{n} x_i^2 + nm^2 - 2m\sum_{i=1}^{n} x_i = \sum_{i=1}^{n} x_i^2 - nm^2 \tag{13}$$

per eseguire il calcolo completo con un solo *loop*, in cui *accumuliamo* la somma degli x_i e degli x_i^2, come mostrato nel frammento qui di seguito.

https://bitbucket.org/lbaldini/computing/src/tip/examples/sample_stat2.py

```
1  import math
2
3  def get_stat(sample):
4      """ Return the sample mean and variance for a list of values.
5
6      This is done in a single loop.
7      """
8      # Again: do not initialize mean as an integer!
9      mean = 0.
10     variance = 0.
11     n = len(sample)
12     for value in sample:
13         mean += value
14         variance += value**2.0
15     mean /= n
16     variance = (variance - n*mean**2.0)/(n - 1)
17     return mean, variance
18
19
20 # Generate a random vector of data.
21 import random
22 random.seed(1)
23 sample = [random.gauss(0, 1) for i in range(10000)]
24
25 # Calculate the sample statistics and print out mean and stdev.
26 mean, variance = get_stat(sample)
27 print(mean, math.sqrt(variance))
28
29 [Output]
30 (0.0030604261498986744, 0.9922712982269474)
```

L'ultimo esempio è sicuramente più elegante e conciso del precedente—e, probabilmente, anche leggermente più veloce. Così come è scritto soffre però una patologia: alla linea 16 sottraiamo due numeri che, potenzialmente sono molto grandi e molto vicini tra loro (ad esempio quando la media del campione è molto elevata e la deviazione standard relativamente piccola). In alcuni casi questo può portare a seri problemi numerici di arrotondamento.

Fixme: Mettere la versione giusta di Knuth.

Fixme: Vedere cosa mette a disposizione scipy.

10.2 LA MEDIA PESATA

Il frammento seguente illustra il calcolo della media pesata per una serie di misure date, secondo le note formule

$$\hat{m} = \frac{\sum_{i=1}^{n} \frac{y_i}{\sigma_{y_i}^2}}{\sum_{i=1}^{n} \frac{1}{\sigma_{y_i}^2}}, \quad \sigma_{\hat{m}}^2 = \frac{1}{\sum_{i=1}^{n} \frac{1}{\sigma_{y_i}^2}}. \tag{14}$$

https://bitbucket.org/lbaldini/computing/src/tip/examples/weighted_average.py

```
import math

# Define the measurements.
values = [1.325, 1.36, 1.32, 1.338, 1.335]
errors = [0.012, 0.05, 0.01, 0.005, 0.006]

# Initialize a couple of variables (mind that they only represent what
# their name suggest at the end of the program---particularly
# average_err is used in the loop to accumulate the sum of weights).
average = 0.
average_err = 0.

# Note the zip function, that allows to loop over multiple lists
# simultaneously.
for value, error in zip(values, errors):
    weight = 1./(error**2)
    average += weight*value
    average_err += weight
average /= average_err
average_err = math.sqrt(1./average_err)

print(average, average_err)

[Output]
(1.3339492233389898, 0.003427508366343914)
```

10.3 FIT DEL MINIMO χ^2 NEL CASO LINEARE

Il frammenoto di codice che segue implementa un fit analitico del minimo χ^2 con una retta. Nella prima parte generiamo una serie di dati secondo la relazione $y = m_0 x + q_0$, aggiungendo ai valori di y un *errore di misura* gaussiano con deviazione standard dipendente dal punto secondo la linea 12. Nella seconda parte eseguiamo il fit vero e proprio.

https://bitbucket.org/lbaldini/computing/src/tip/examples/least_squares.py

```
import math
import random
random.seed(1)

# Define the measurements (make sure you do understand the following
# three lines and put in print statements if that is helpful).
# Note you can split long lines with a "\" character
n = 9
m0 = 1.5
q0 = 1.0
xvalues = [i + 1.0 for i in range(n)]
yerrors = [0.05 + 0.02*i for i in range(n)]
```

```
yvalues = [m0*x + q0 + random.gauss(0, dy) for \
               x, dy in zip(xvalues, yerrors)]

# Initialize some variables (note we do all at once).
s = sx = sxx = sy = sxy = 0.
# Loop over the data points.
for x, y, dy in zip(xvalues, yvalues, yerrors):
    w = 1./(dy**2)
    s += w
    sx += x*w
    sxx += (x**2)*w
    sy += y*w
    sxy += x*y*w
# Calculate the fit parameters.
D = (sxx*s - sx**2)
m = (sxy*s - sx*sy)/D
sigma_m = math.sqrt(s/D)
q = (sy*sxx - sxy*sx)/D
sigma_q = math.sqrt(sxx/D)

# And here a first example of formatted output.
print('m = %.3f +- %.3f, q = %.3f +- %.3f' % (m, sigma_m, q, sigma_q))

[Output]
m = 1.468 +- 0.015, q = 1.108 +- 0.052
```

Notiamo che i parametri restituiti dal fit sono a circa 2σ dai valori *veri* (cioè quelli che abbiamo usato per generare la serie di dati).

10.4 RICERCA BINARIA

Nell'esempio che segue calcoliamo numericamente lo zero di una funzione utilizzando una ricerca binaria.

https://bitbucket.org/lbaldini/computing/src/tip/examples/binary_search.py

```
import math

def f(x):
    # Evaluation of f(x).
    return math.exp(x) - x**2.0

# Variable initialization.
precision = 1e-6
x1 = -1
x2 = 1
x = (x1 + x2)/2.0
y = f(x)
n = 0

# While loop: we exit when the difference from zero of the function value
# is whithin the predefined precision.
while abs(y) > precision:
    if y > 0:
        x2 = x
    else:
        x1 = x
    x = (x1 + x2)/2.0
    y = f(x)
```

```
    n += 1

# Print out the numeber of iteration, x and f(x).
print(n, x, f(x))

[Output]
(20, -0.7034673690795898, 1.0159194241410319e-07)
```

Fixme: Spiegare un pochino e far notare che questo metodo è molto più efficiente di una ricerca su griglia a passo fisso.

11

LAVORARE CON GLI ARRAY: NUMPY

In questo capitolo ci occupiamo di *array* ossia di strutture dati per rappresentare vettori o matrici ad un numero arbitrario di indici. Il pacchetto `numpy` fornisce...

Fixme: Breve introduzione a `numpy`. Dire che va installato a parte. Servono istruzioni?

11.1 DIFFERENZE TRA ARRAY E LISTE

Un *array* unidimensionale è semplicemente una sequenza ordinata (diremmo un *vettore*) di numeri. In `numpy` si instanzia semplicemente come:

https://bitbucket.org/lbaldini/computing/src/tip/examples/array_init.py

```
import numpy

# Create a numpy array of 5 elements.
a = numpy.array([0., 1., 3., 4., 8.])

# And take a quick look.
print(a)
print(a[2])

[Output]
[ 0.  1.  3.  4.  8.]
3.0
```

A prima vista questo non è molto diverso dal frammento di codice seguente, che utilizza solo liste native di Python:

https://bitbucket.org/lbaldini/computing/src/tip/examples/array_init_list.py

```
# Do something similar with a python list of 5 elements.
a = [0., 1., 3., 4., 8.]
print(a)
print(a[2])

[Output]
[0.0, 1.0, 3.0, 4.0, 8.0]
3.0
```

In effetti, a parte piccoli dettagli di formattazione, gli output dei due programmi sono sostanzialmente identici. Allora uno può chiedersi che differenze ci siano tra una lista di Python ed un array di `numpy`, e se veramente questo ultimo tipo di struttura dati. Le risposte a queste due domande, in ordine, sono: "moltissime" e "si", come vedremo tra un attimo.

Vediamo allora un esempio appena più avanzato, in cui creaiamo una lista ed un array con lo stesso contenuto e, immediatamente dopo, proviamo ad inserire un numero in virgola mobile.

https://bitbucket.org/lbaldini/computing/src/tip/examples/array_list1.py

```
import numpy

# Create a python list and a numpy array with the same content.
l = [0., 1., 3., 4., 8.]
a = numpy.array([0., 1., 3., 4., 8.])

# Append a floating point number to both.
# (Note the syntax is slightly different in the two cases.)
l.append(16.)
a = numpy.insert(a, len(a), 16.)
print(l)
print(a)

[Output]
[0.0, 1.0, 3.0, 4.0, 8.0, 16.0]
[  0.   1.   3.   4.   8.  16.]
```

Fino a qui tutto funziona come atteso, a parte il fatto che gli array non hanno il metodo `append()` e siamo costretti ad utilizzare una sintassi un pochino più laboriosa.

Ma qui è più o meno dove il parallelo finisce. Se adesso proviamo ad aggiungere una stringa, l'operazione funziona meraviglia sulla lista ma fallisce miseramente sull'array.

https://bitbucket.org/lbaldini/computing/src/tip/examples/array_list2.py

```
import numpy

# Create a python list and a numpy array with the same content.
l = [0., 1., 3., 4., 8.]
a = numpy.array([0., 1., 3., 4., 8.])

# Now try and append a string. Ouch...
l.append('howdy?')
a = numpy.insert(a, len(a), 'howdy?')
print(l)
print(a)

[Output]
Traceback (most recent call last):
  File "/data/work/teaching/computing/examples/array_list2.py", line 9, in <module>
    a = numpy.insert(a, len(a), 'howdy?')
  File "/usr/lib/python2.7/site-packages/numpy/lib/function_base.py", line 3452, in insert
    new[slobj] = values
ValueError: could not convert string to float: howdy?
```

Il motivo è semplice: come vedremo nella prossima sezione gli *array* di `numpy` forniscono tutta una serie di funzionalità avanzate (la, somma, tanto per dirne una) che semplicemente non avrebbero senso se la struttura dati di base permettesse di mischiare tipi di variabile diversa. (Vi immaginate, ad esempio, se provassimo a sommare un numero in virgola mobile ed una stringa?)

Così ogni *array* di `numpy` ha un suo tipo bene definito, che viene fissato, implicitamente o esplicitamente, al momento della creazione.

https://bitbucket.org/lbaldini/computing/src/tip/examples/array_types.py

```
import numpy

# Instantiate three different arrays (of length 1).
a = numpy.array([0])
b = numpy.array([0.])
c = numpy.array([0.], 'float32')
```

```
7
8  print(a.dtype)
9  print(b.dtype)
10 print(c.dtype)
11
12 [Output]
13 int32
14 float64
15 float32
```

In questo caso `a` contiene numeri interi a 64 bit, e `b` e `c` numeri in virgola mobile a 32 e 64 bit, rispettivamente. Va da sé che, in questo modo, rientriamo immediatamente in un mondo in cui tutti i problemi legati alla rappresentazione dei numeri interi, appena accennati nella sezione 7.2, tornano ad essere rilevanti.

https://bitbucket.org/lbaldini/computing/src/tip/examples/array_overflow.py

```
1  import numpy
2
3  # Create a 1-dimensional array of 8-bit integers.
4  a = numpy.array([0, 0, 0], 'int8')
5
6  # Oops... this is only holding integers from -128 to 127!
7  a[0] = 100
8  a[1] = 200
9  a[2] = 300
10
11 print(a)
12
13 [Output]
14 [100 -56  44]
```

11.2 OPERAZIONI (PIÙ O MENO) ELEMENTARI

Abbiamo capito che gli *array* di `numpy` sono oggetti profondamente diversi dalle liste native di Python. Cominciamo adesso a vederli all'opera!

Per prima cosa `nupmy` supporta tutte le operazioni elementari tra *array*, come mostrato nel frammento seguente.

https://bitbucket.org/lbaldini/computing/src/tip/examples/array_basics.py

```
1  import numpy
2
3  a = numpy.array([10., 20., 30.])
4  b = numpy.array([1., 2., 3.])
5
6  print(a + b)
7  print(a - b)
8  print(a*b)
9  print(a/b)
10 print(a**2)
11
12 [Output]
13 [ 11.  22.  33.]
14 [  9.  18.  27.]
15 [ 10.  40.  90.]
16 [ 10.  10.  10.]
17 [ 100.  400.  900.]
```

Così si possono sommare, sottrarre, moltiplicate, dividere ed elevare a potenza oggetti di tipo *array*. Tutte queste operazioni sono eseguite *elemento per elemento*, e nel seguito vedremo numerosi esempi di come questo sia utile nella vita reale. Notiamo, per inciso, che le liste native di Python, per buoni motivo, non offrono nessuna di queste funzionalità. Per eseguire le stesse operazioni su liste avremmo avuto bisogno di un ciclo `for`[1].

Le operazioni aritmetiche elementari non sono l'unico *piatto* del menù offerto da `numpy`. Mentre le funzioni matematiche della libreria standard `math` di Python non sono adatte, in generale, ad operare su *array* di lunghezza maggiore di 1

https://bitbucket.org/lbaldini/computing/src/tip/examples/array_funcs1.py

```
import numpy
import math

a = numpy.array([1., 2., 3.])
print math.exp(a)

[Output]
Traceback (most recent call last):
  File "/data/work/teaching/computing/examples/array_funcs1.py", line 5, in <module>
    print math.exp(a)
TypeError: only length-1 arrays can be converted to Python scalars
```

`numpy` offre un'implementazione alternativa di tali funzioni (e molte altre), disegnata esplicitamente per operare su *array*.

https://bitbucket.org/lbaldini/computing/src/tip/examples/array_funcs2.py

```
import numpy
import math

a = numpy.array([1., 2., 3.])
b = numpy.array([10., 100., 1000.])

print(numpy.exp(a))
print(numpy.log10(b))

[Output]
[  2.71828183   7.3890561   20.08553692]
[ 1.  2.  3.]
```

Il lettore è invitato a notare la sottile differenza tra `math.exp()` e `numpy.exp()`.

Vedremo nella prossima sezione che questo aumenta enormemente la potenza delle armi a nostra disposizione per affrontare i problemi tipici dell'analisi dati in laboratorio.

11.3 ALCUNE APPLICAZIONI

In questa sezione rivisitiamo, forti delle nostre nuove cononscenze, alcuni dei problemi che abbiamo visto nella prima parte di questo documento.

11.3.1 *Calcolo di media e varianza campione*

Il calcolo della media e della varianza campione visto nella sezione **??** può essere eseguito utilizzando il metodo `sum()` degli *array* di `numpy` per risparmiare un ciclo `for` (esplicito):

1 Ovviamente anche utilizzando *array* un ciclo `for` deve essere eseguito, in qualche modo. Ma questo ciclo è nascosto dietro le quinte. Per completezza, aggiungiamo anche che è eseguito in modo computazionalmente più efficiente di quanto sarebbe possibile in *puro* Python, ma l'ottimizzazione non è, a questo livello, il nostro scopo principale, per cui non insistiamo oltre.

https://bitbucket.org/lbaldini/computing/src/tip/examples/sample_stat_numpy.py

```
import numpy

def get_stat(sample):
    """ Return the sample mean and variance for a list of values.

    Note the input argument must be a numpy array.
    """
    mean = sample.sum()
    variance = (sample**2).sum()
    n = len(sample)
    mean /= n
    variance = (variance - n*mean**2.0)/(n - 1)
    return mean, variance


# Generate a random vector of data (this time we use numpy).
numpy.random.seed(1)
sample = numpy.random.normal(0., 1., 10000)

# Calculate the sample statistics and print out mean and stdev.
mean, variance = get_stat(sample)
print(mean, numpy.sqrt(variance))

[Output]
(0.0097726566991049712, 0.99883578672599183)
```

Notiamo, per inciso, che numpy offre anche funzioni predefinite per il calcolo delle statistiche campione:

https://bitbucket.org/lbaldini/computing/src/tip/examples/sample_stat_numpy2.py

```
import numpy

# Generate a random vector of data (this time we use numpy).
numpy.random.seed(1)
sample = numpy.random.normal(0., 1., 10000)

# Calculate the sample statistics and print out mean and stdev.
mean = numpy.mean(sample)
stdev = numpy.std(sample, ddof = 1)
print(mean, stdev)

[Output]
(0.0097726566991049712, 0.99883578672599183)
```

11.3.2 *La media pesata*

Veniamo adesso alla media pesata, che abbiamo già visto nelle sezione **??**. Utilizzando la macchineria di numpy il codice è decisamente più snello (ed il risultato, come potete facilmente verificare, non cambia).

https://bitbucket.org/lbaldini/computing/src/tip/examples/weighted_average_numpy.py

```
import numpy

# Define the measurements (note they are numpy array).
values = numpy.array([1.325, 1.36, 1.32, 1.338, 1.335])
errors = numpy.array([0.012, 0.05, 0.01, 0.005, 0.006])
```

```
6
7  # And use the numpy facilities to do the actual computation on the fly!
8  weights = 1./errors**2
9  average = (values*weights).sum()/weights.sum()
10 average_err = numpy.sqrt(1./weights.sum())
11
12 print(average, average_err)
13
14 [Output]
15 (1.3339492233389898, 0.0034275083663439141)
```

11.3.3 *Fit del minimo χ^2 nel caso lineare*

Ed ecco, infine, la traduzione nel linguaggio di numpy del nostro programma per il fit del minimo χ^2 nel caso lineare.

https://bitbucket.org/lbaldini/computing/src/tip/examples/least_squares_numpy.py

```
1  import numpy
2  import math
3  import random
4  random.seed(1)
5
6  # Define the measurements (this is identical to the plain example).
7  n = 9
8  m0 = 1.5
9  q0 = 1.0
10 xvalues = [i + 1.0 for i in range(n)]
11 yerrors = [0.05 + 0.02*i for i in range(n)]
12 yvalues = [m0*x + q0 + random.gauss(0, dy) for \
13                x, dy in zip(xvalues, yerrors)]
14
15 # Convert the measurements into numpy arrays.
16 x = numpy.array(xvalues)
17 y = numpy.array(yvalues)
18 dy = numpy.array(yerrors)
19
20 # Do the actual computation.
21 w = 1./(dy**2)
22 s = w.sum()
23 sx = (x*w).sum()
24 sxx = ((x**2)*w).sum()
25 sy = (y*w).sum()
26 sxy = (x*y*w).sum()
27 # Calculate the fit parameters.
28 D = (sxx*s - sx**2)
29 m = (sxy*s - sx*sy)/D
30 sigma_m = math.sqrt(s/D)
31 q = (sy*sxx - sxy*sx)/D
32 sigma_q = math.sqrt(sxx/D)
33
34 # And here a first example of formatted output.
35 print('m = %.3f +- %.3f, q = %.3f +- %.3f' % (m, sigma_m, q, sigma_q))
36
37 [Output]
38 m = 1.468 +- 0.015, q = 1.108 +- 0.052
```

12

ANCORA SULLA MATEMATICA: SCIPY

13

REALIZZARE GRAFICI

Questo capitolo costituisce una breve panoramica su `matplotlib`, una libreria grafica 2D per la realizzazione di grafici di alta qualità. Più precisamente utilizzeremo il modulo `pylab`, che combina le funzionalità grafiche di `matplotlib.pyplot` con le funzionalità di `numpy`. Al di là dei dettagli tecnici, assumendo che la singola istruzione

https://bitbucket.org/lbaldini/computing/src/tip/examples/pylab_import.py

```
import pylab
```

funzioni senza dare errori, siete pronti per avventurarvi nel resto del capitolo.

13.1 IL MIO PRIMO GRAFICO CARTESIANO

Supponiamo di avere un *file* di dati contenente la posizione, misurata ad intervalli di tempo regolari, di un corpo che si muove su una retta (ovviamente con l'errore associato). Il *file* è organizzato su tre colonne come mostrato di seguito.

https://bitbucket.org/lbaldini/computing/src/tip/examples/data/scatter_plot.txt

```
#t [s]  s [m]   Delta s [m]
1.0     2.08    0.60
2.0     4.44    0.61
3.0     6.09    0.62
4.0     6.88    0.64
5.0     8.61    0.65
6.0     9.79    0.67
7.0     11.96   0.69
8.0     12.06   0.72
9.0     15.01   0.74
```

Vogliamo realizzare un grafico cartesiano (ciò che in Inglese prende il nome di *scatter plot*) dei nostri dati sperimentali. Un programma minimale che funge allo scopo è riportato nel frammento di codice che segue (ed il grafico generato è mostrato in figura 3).

https://bitbucket.org/lbaldini/computing/src/tip/examples/pylab_scatter_plot.py

```
import pylab

# Load the data from the file.
t, s, ds = pylab.loadtxt('data/scatter_plot.txt', unpack = True)

# Create the scatter plot.
pylab.errorbar(t, s, ds)

# Save the plot to a pdf file for later use (maybe in a writeup?).
pylab.savefig('scatter_plot.pdf')

```

```
12 # Finally: show the plot on the screen.
13 pylab.show()
```

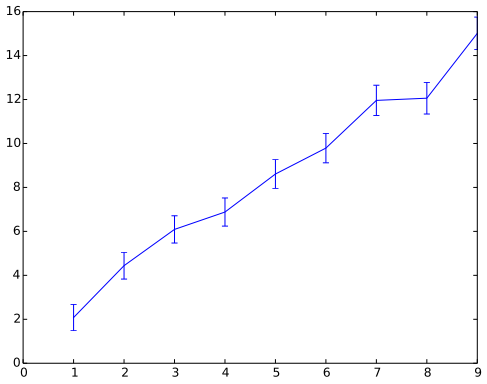

FIGURA 3. Grafico prodotto dal frammento di codice mostrato sopra. Vedi il testo per una lista dei difetti più importanti.

Le istruzioni non hanno bisogno di particolari spiegazioni oltre ai commenti. Notiamo che, dopo che l'ultima istruzione (alla linea 13) è stata eseguita il flusso del programma si interrompe fino a che l'utente non chiude la finestra in cui il grafico viene mostrato.

Il risultato mostrato in figura 3, dobbiamo ammetterlo, non è particolarmente eccitante—difficilmente troverebbe collocazione in una pubblicazione scientifica. Passsiamo brevemente in rassegna i problemi più ovvi.

▷ Le grandezze rappresentate sugli assi (e le rispettive unità di misura) non sono indicate.

▷ I punti sperimentali sono uniti con una spezzata—cosa che, generalmente, non ha nessun significato quantitativo, per cui non è da considerarsi una buona pratica.

▷ I numeri sono troppo piccoli. (Tipicamente è buona regola assicurarsi che le dimensioni dei caratteri siano confrontabili con quelle del testo.)

▷ I *marker* (cioè gli elementi grafici che identificano il punto sperimentali) sono di fatto invisibili.

▷ L'estremo superiore dell'asse delle ascisse concide con la coordinata dell'ultimo punto sperimentale, per cui la barra d'errore di quest'ultimo si confonde con la cornice del grafico.

Aggiungeremmo anche che, visto che spesso gli articoli scientifici vengono stampati in bianco e nero, è buona regola non utilizzare i colori a meno che non sia strettamente necessario.

Il frammento seguente è un tentativo di riscrittura del nostro primo esempio sulla base delle considerazioni appena espresse (ed il grafico generato è mostrato in figura 4).

https://bitbucket.org/lbaldini/computing/src/tip/examples/pylab_scatter_plot_fancier.py

```
1  import pylab
2
3  # Load the data from the file.
4  t, s, ds = pylab.loadtxt('data/scatter_plot.txt', unpack = True)
5
6  # Format the plot.
7  pylab.rc('font', size = 18)
8  pylab.title('Law of motion', y = 1.02)
9  pylab.xlabel('t [s]')
10 pylab.ylabel('s [m]', labelpad = 25)
11 pylab.xlim(0, 10)
12 pylab.ylim(0, 18)
```

```
pylab.grid(color = 'gray')

# Create the scatter plot.
pylab.errorbar(t, s, ds, linestyle = '', color = 'black', marker = 'o')

# Save the plot to a pdf file for later use (maybe in a writeup?).
pylab.savefig('scatter_plot_fancier.pdf')

# Finally: show the plot on the screen.
pylab.show()
```

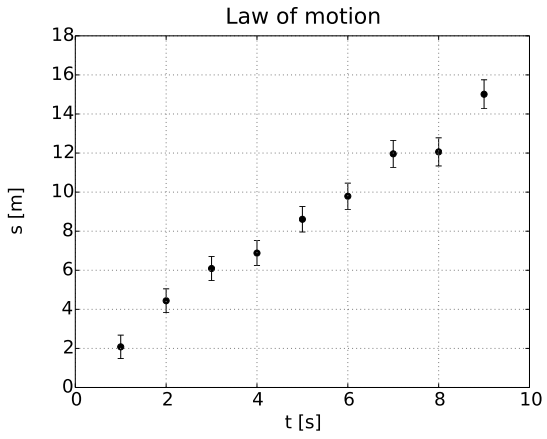

FIGURA 4. Versione migliorata del nostro primo grafico. Si noti che abbiamo anche aggiunto un titolo ed una griglia (che, in genere, aumenta la leggibilità).

13.2 ISTOGRAMMI

L'esempio seguente genera un campione di dati estratti da una distribuzione normale e crea un istogramma dei valori corrispondenti. Non ci dilunghiamo commentando i vari comandi, molti dei quali sono stati utilizzati negli esempi precedenti. Il risultato è mostrato in figura 5.

https://bitbucket.org/lbaldini/computing/src/tip/examples/pylab_histo.py

```
import pylab
import numpy
numpy.random.seed(1)

# Generate a sample of 100000 gaussian-distributed values.
sample = numpy.random.normal(0., 1., 100000)

# Format the plot.
pylab.rc('font', size = 18)
pylab.title('A histogram', y = 1.02)
pylab.xlabel('x')
pylab.ylabel('Counts', labelpad = -5)
pylab.ylim(0, 10000)
pylab.grid(color = 'gray')

# Create the histogram.
pylab.hist(sample, bins = 50, range = (-5, 5), histtype = 'step',
           color = 'black')

# Save the plot to a pdf file for later use (maybe in a writeup?).
pylab.savefig('histogram.pdf')

```

```
# Finally: show the plot on the screen.
pylab.show()
```

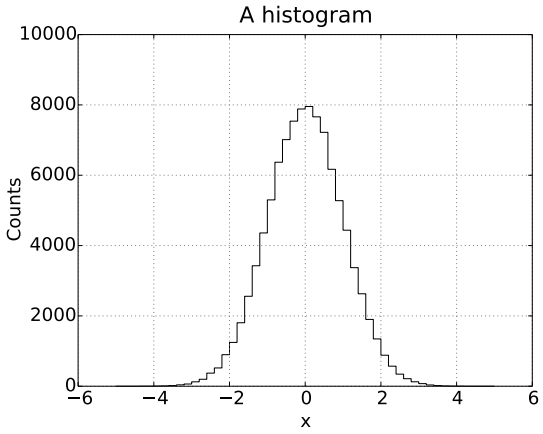

FIGURA 5. Esempio di istogramma.

13.3 GRAFICI A BARRE

Si tratta di un tipo di rappresentazione adatto per variabili intrinsecamente discrete, come ad esempio la somme delle uscite del lancio di un certo numero di dadi. Supponiamo di avere in ingresso il *file* seguente—che contiene il numero di occorrenze della somma delle uscite di due dadi lanciati 1000 volte.

https://bitbucket.org/lbaldini/computing/src/tip/examples/data/plasduino_dice.txt

```
#
# Written by plasduino.diceEventBuffer on Thu Jan 16 09:31:56 2014.
#
# Sample mean = 6.996000, sample standard deviation = 2.385304 (1000 entries)
#
2         24
3         51
4         91
5         115
6         138
7         165
8         136
9         123
10         76
11         54
12         27
```

La macro seguente mostra i dati di cui sopra nella forma di un grafico a barre (ed il risultato è mostrato in figura 6).

https://bitbucket.org/lbaldini/computing/src/tip/examples/pylab_bar.py

```
import pylab
import numpy
numpy.random.seed(1)

# Load the input data.
x, y = pylab.loadtxt('data/plasduino_dice.txt', unpack = True)

```

```
# Format the plot.
pylab.rc('font', size = 18)
pylab.title('A bar plot', y = 1.02)
pylab.xlabel('Sum')
pylab.ylabel('Counts')
pylab.xlim(0, 15)
pylab.ylim(0, 200)
pylab.grid(color = 'gray')

# Create the histogram. Note that we offset the x values by 1/2
# times the width of the bars.
w = 0.25
pylab.bar(x - w/2., y, width = w, color = 'white')

# Save the plot to a pdf file for later use (maybe in a writeup?).
pylab.savefig('bar.pdf')

# Finally: show the plot on the screen.
pylab.show()
```

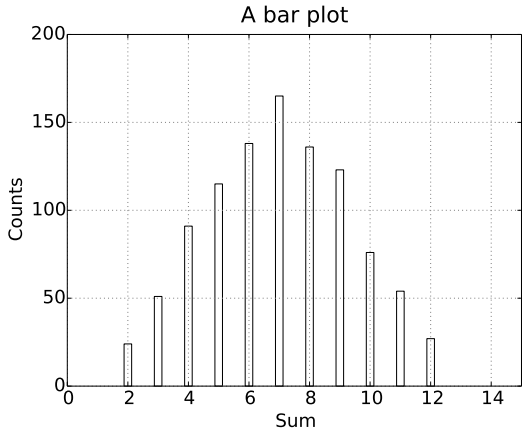

FIGURA 6. Esempio di grafico a barre.

13.4 GRAFICI DI FUNZIONI MATEMATICHE

Parlando di grafici, c'è almeno una cosa potenzialmente utile che non abbiamo ancora discusso: come fare, cioè, il grafico di una funzione. Concettualmente possiamo pensare di calcolare i valori y_i della funzione per una serie predefinita di valori x_i della variabile indipendente x ed unire le coppie (x_i, y_i) così ottenute con dei segmenti. Nel limite in cui i punti sono abbastanza vicini tra loro la spezzata risultante sarà di fatto indistinguibile dal grafico cartesiano della funzione di partenza.

Il frammento di codice che segue disegna il grafico della funzione

$$f(x) = x^2 \tag{15}$$

campionandone il valore in corrispondenza di 100 valori di x (cfr. figura 7).

https://bitbucket.org/lbaldini/computing/src/tip/examples/pylab_func_plot.py

```
import pylab
import numpy

# Create a grid of x-values (from 0 to 10 in 100 steps).
x = numpy.linspace(0, 10, 100)
```

```
6
7  # Calculate the y values. Note x and y are both arrays!
8  y = x**2
9
10 # Some formatting.
11 pylab.rc('font', size = 18)
12 pylab.title('A function', y = 1.02)
13 pylab.xlabel('x')
14 pylab.ylabel('f(x)')
15 pylab.grid(color = 'gray')
16
17 # Plot the graph.
18 pylab.plot(x, y, color = 'black')
19
20 # Save the plot to a pdf file for later use (maybe in a writeup?).
21 pylab.savefig('square.pdf')
22
23 # And show the plot.
24 pylab.show()
```

I comandi dovrebbero essere essenzialmente autoesplicativi, ma vale la pena sottolineare come `x` ed `y`, alle linee 5 e 8, siano *array* e non semplici scalari: il tutto funziona perché gli operatori matemamatici standard—ed in particolare quello di elevamento a potenza—funzionano con gli *array*. Interessante anche la funzione `linespace` di `numpy`, che genera una sequenza di punti equispaziati.

https://bitbucket.org/lbaldini/computing/src/tip/examples/linspace.py

```
1 import numpy
2
3 # Generate 5 equally-spaces points between 0. and 10.
4 print(numpy.linspace(0., 10., 5))
5
6 [Output]
7 [  0.    2.5   5.    7.5  10. ]
```

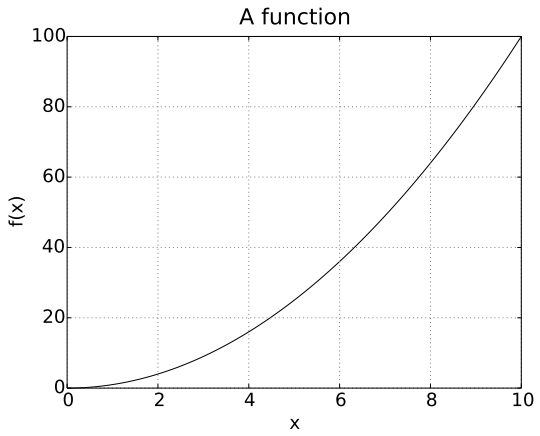

FIGURA 7. Grafico cartesiano della funzione $f(x) = x^2$.

`pylab` si può utilizzare in congiunzione con la libreria statistica di `scipy` per generare grafici di qualsiasi funzione abbia rilevanza pratica.

https://bitbucket.org/lbaldini/computing/src/tip/examples/pylab_gauss_plot.py

```
1 import pylab
2 import numpy
```

```
import scipy.stats

# Create a grid of x-values (from 0 to 10 in 100 steps).
x = numpy.linspace(-5, 5, 100)

# Grab the normal distribution from the scipy.stats module.
rv = scipy.stats.norm()

# Calculate the y values. Note x and y are both arrays (Mind here you
# want the probability density function of the distribution).
y = rv.pdf(x)

# Some formatting.
pylab.rc('font', size = 18)
pylab.title('A standard gaussian', y = 1.02)
pylab.xlabel('x')
pylab.ylabel('pdf(x)')
pylab.xlim(-5, 5)
pylab.ylim(0, 0.5)
pylab.grid(color = 'gray')

# Plot the graph.
pylab.plot(x, y, color = 'black')

# Save the plot to a pdf file for later use (maybe in a writeup?).
pylab.savefig('gauss.pdf')

# And show the plot.
pylab.show()
```

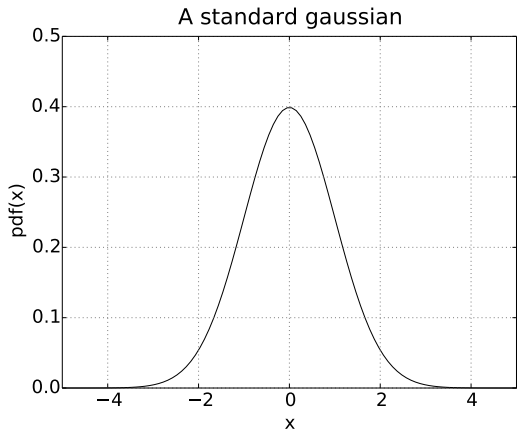

FIGURA 8. Grafico cartesiano della densità di probabilità di una variabile gaussiana in forma standard.

14

FIT DI TIPO NUMERICO

In questo capitolo illustriamo brevemente come utilizzare il modulo `scipy` per eseguire fit di tipo numerico.

14.1 UN PRIMO ESEMPIO DI FIT LINEARE

Come primo esempio eseguiamo un fit del minino χ^2 con una funzione lineare dei dati utilizzati nel capitolo precedente per le figure 3 e 4. Un esempio di *script* è mostrato di seguito ed il risultato grafico è mostrato in figura 9.

https://bitbucket.org/lbaldini/computing/src/tip/examples/pylab_fit_scatter.py

```
import pylab
import numpy
from scipy.optimize import curve_fit


def fit_function(x, m, q):
    """ Definition of the fit function.

    This is a simple straight line of the form y = m*x + q.
    Mind that the x argument, here, is a numpy array, and the whole thing
    works because you can multiply an array by a scalar and you can add a
    scalar to an array.
    """
    return m*x + q


# Load the data from the file.
x, y, dy = pylab.loadtxt('data/scatter_plot.txt', unpack = True)

# Set the initial values for the fit parameters and fit the data.
initial_values = (1., 1.)
pars, covm = curve_fit(fit_function, x, y, initial_values, dy)

# Retrieve the best-fit parameters and related quantities, and print
# them on the terminal.
m0, q0 = pars
dm, dq = numpy.sqrt(covm.diagonal())
chisq = (((y - fit_function(x, m0, q0))/dy)**2).sum()
ndof = len(x) - 2
print('m = %f +- %f' % (m0, dm))
print('q = %f +- %f' % (q0, dq))
print('Chisquare/ndof = %f/%d' % (chisq, ndof))

# Format the plot.
pylab.rc('font', size = 18)
pylab.title('Law of motion', y = 1.02)
```

```
37 pylab.xlabel('t [s]')
38 pylab.ylabel('s [m]', labelpad = 25)
39 pylab.xlim(0, 10)
40 pylab.ylim(0, 18)
41 pylab.grid(color = 'gray')
42
43 # Create the scatter plot.
44 pylab.errorbar(x, y, dy, linestyle = '', color = 'black', marker = 'o')
45
46 # Create a one-dimensional grid and draw the fit function.
47 func_grid = numpy.linspace(0, 10, 100)
48 pylab.plot(func_grid, fit_function(func_grid, m0, q0), color = 'black')
49
50 # Save the plot to a pdf file for later use (maybe in a writeup?).
51 pylab.savefig('fit_scatter.pdf')
52
53 # Finally: show the plot on the screen.
54 pylab.show()
55
56 [Output]
57 m = 1.488891 +- 0.069653
58 q = 1.105806 +- 0.369456
59 Chisquare/ndof = 4.623582/7
```

Si tratta di gran lunga dell'esempio più complesso che abbiamo incontrato fino a questo commento, per cui spenderemo il resto di questa sezione commentandolo debitamente.

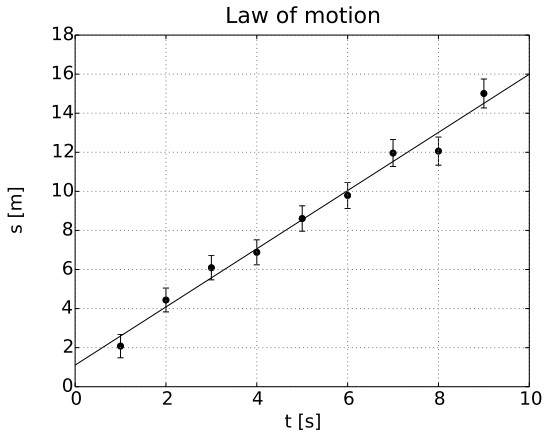

FIGURA 9. Esempio di fit ad una serie di dati con una funzione lineare.

Nelle linee 6–14 definiamo la funzione di *fit*—in questo caso una semplice retta, definita dal coefficiente angolare m e dall'intercetta q

$$y = mx + q. \tag{16}$$

Come vedremo tra un attimo, questa funzione sarà usata dal metodo `curve_fit` del modulo `scipy.optimize` alla linea 22. Non ci addentreremo nella descrizione di questo algoritmo, per maggiori informazioni rimandiamo il lettore alla documentazione online[1], per avere un'idea delle sue capacità e limitazioni è sufficiente dire che può essere considerato l'equivalente numerico del metodo dei minimi quadrati studiato a lezione. Notiamo che la nostra funzione di *fit* è una funzione nel senso di Python (cfr. il capitolo 4). Anche se non è ovvio dalla sintassi, la funzione accetta come primo argomento `x` un *array* (nel senso di `numpy`) di valori della variabile indipendente x e restituisce

1 `http://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.curve_fit.html`

l'*array* dei corripondenti valori della variabile dipendente y, per due generici valori dei parametri m e q (sarà compito del metodo `curve_fit` variare opportunamente i valori di m e q nella procedura iterativa di *fit*). Un semplice *script* vale più di mille parole.

https://bitbucket.org/lbaldini/computing/src/tip/examples/pylab_fit_func.py

```
1  import numpy
2
3
4  def straight_line(x, m, q):
5      """ Definition of a simple straight line of the form y = m*x + q.
6      """
7      return m*x + q
8
9
10 # Create an array of number, from 0 to 10 in 5 steps.
11 x0 = numpy.linspace(0., 10., 5)
12 # Set the straight-line parameters.
13 m0 = 2.
14 q0 = 1.
15
16 # Print the values of the independent and dependent variable.
17 print(x0)
18 print(straight_line(x0, m0, q0))
19
20 [Output]
21 [  0.    2.5   5.    7.5  10. ]
22 [  1.   6.  11.  16.  21.]
```

Torniamo all'esempio completo mostrato a pagina 55. Dopo aver *caricato* i dati in ingresso, alla linea 21 inizializziamo i valori dei parametri ed alla linea 22 eseguiamo il *fit* vero e proprio. La funzione `curve_fit` del modulo `scipy.optimize` richiede (nell'ordine)

1) la funzione di *fit*;
2) l'array di valori della variabile indipendente x;
3) l'array di valori della variabile dipendente y;
4) i valori iniziali dei parametri (in questo caso m e q);
5) l'array (opzionale) degli errori sulla variabile dipendente y,

e ritorna (nell'ordine)

1) l'array dei valori di *best-fit* dei parametri;
2) la matrice di covarianza tra i parametri.

Alle linee 26–29 estraiamo tutte le informazioni utili dai prodotti del *fit*. Gli errori sui parametri sono dati dalla radice quadrata dei corrispondenti elementi diagonali della matrice di covarianza (linea 27). Il χ^2 del fit si può calcolare direttamente come

$$\chi^2 = \sum_{i=1}^{n} \left(\frac{y_i - f(x_i; m, q)}{\sigma_{y_i}} \right)^2, \tag{17}$$

che è esattamente quanto scritto (nel linguaggio *sintetico* di `numpy`) alla linea 28. Infine, il numero di gradi di libertà è dato dal numero di misure meno il numero dei parametri (in questo caso 2)

$$\nu = n - 2 \tag{18}$$

(cfr. linea 29). Il parallelo tra ciò che avete studiato sul libro di statistica e questo esempio di codice è molto stretto.

14.2 IL PROBLEMA DELLA STIMA DEI VALORI INIZIALI

In generale è impossibile scrivere in forma chiusa le espressioni per i parametri di *best-fit* (e.g. con un fit dei minimi quadrati, pesato o non pesato) per un modello arbitrario. Per questo motivo la funzione `curve_fit` del modulo `scipy.optimize` utilizza un metodo numerico *iterativo* per la stima dei parametri: si parte da un insieme di valori iniziali e si fanno variare in modo opportuno fino a che l'algoritmo non converge, secondo un qualche criterio, al minimo del χ^2.

Non si tratta semplicemente di una difficoltà di tipo pratico: al di là dei casi più semplici non è garantito che il χ^2 abbia un solo minimo al variare dei parametri. In tal caso l'assegnazione dei valori iniziali dei parametri (cfr. la linea 21 nell'esempio con cui abbiamo aperto la sezione) diviene fondamentale, poiché *valori iniziali diversi per i parametri possono far convergere il fit a soluzioni diverse*—che non possono, ovviamente essere tutte giuste. Ciò che accade, tecnicamente, è che, quando il χ^2, in funzione dei parametri, ha più di un minimo, le procedure di fit iterative possono convergere, a seconda della scelta dei valori iniziali, ad uno dei minimi locali, anziché il minimo globale cercato.

Illustreremo il problema con un semplice esempio unidimensionale. Consideriamo la serie di dati nella tabella seguente, e rappresentati nel grafico cartesiano in figura 10. Si tratta della misura (simulata) della proiezione s su uno dei due assi ortogonali del moto di un punto materiale su una circonferenza di raggio unitario.

https://bitbucket.org/lbaldini/computing/src/tip/examples/data/fit_initial_val.txt

```
0.000000        -0.044256        0.050000
0.992082        0.937443        0.050000
1.984164        0.710725        0.050000
2.976246        -0.419388        0.050000
3.968328        -0.989147        0.050000
4.960409        -0.357852        0.050000
5.952491        0.735501        0.050000
6.944573        0.888280        0.050000
7.936655        -0.072213        0.050000
8.928737        -0.877147        0.050000
9.920819        -0.595930        0.050000
10.912901        0.513773        0.050000
11.904983        1.083195        0.050000
12.897065        0.215203        0.050000
13.889146        -0.849774        0.050000
14.881228        -0.878043        0.050000
15.873310        0.143627        0.050000
16.865392        0.913651        0.050000
17.857474        0.587699        0.050000
18.849556        -0.556872        0.050000
```

Assumiamo di non conoscere la velocità angolare e cerchiamo di fittare i nostri dati con il modello

$$s(t; \omega) = \sin(\omega t) \tag{19}$$

al variare del parametro ω. (In generale includeremmo nel nostro modello altri due parametri—uno per l'ampiezza A ed uno per la fase ϕ—ma in questo caso i dati sono stati generati con $A = 1$ e $\phi = 0$ per mantenere l'esempio il più semplice possibile.) Notiamo, per inciso, che i punti sperimentali sembrano compiere tre oscillazioni complete in ≈ 16 s, cioè con un periodo $T \approx 5.3$ s, per cui ci aspettiamo che $\omega = 2\pi/T \approx 1.2$ s. Questo potrebbe essere un valore iniziale ragionevole per il nostro fit.

Proviamo allora ad eseguire il fit provando a variare i valori iniziali dei parametri per vedere che cosa succede...

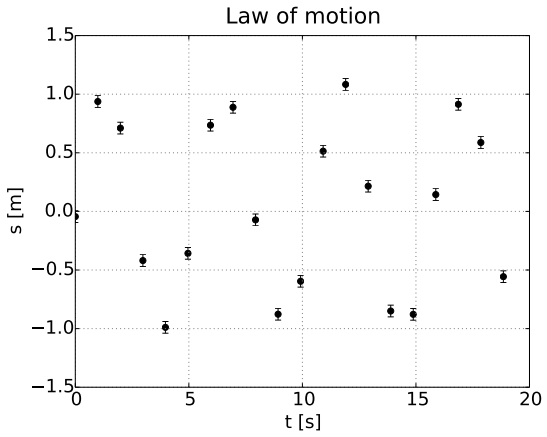

FIGURA 10. Misura (simulata) della proiezione s su uno dei due assi ortogonali del moto di un punto materiale su una circonferenza di raggio unitario. Gli errori sulla misura di s sono costanti e pari a 10 cm, mentre gli errori sulla misura dei tempi sono trascurabili.

https://bitbucket.org/lbaldini/computing/src/tip/examples/fit_initial_val_fit.py

```
import pylab
from scipy.optimize import curve_fit

def fit_function(x, omega):
    """ Definition of the fit function.
    """
    return pylab.sin(omega*x)

# Load the data from the file.
t, s, ds = pylab.loadtxt('data/fit_initial_val.txt', unpack = True)

# Pick a few different initial values and fit.
for initial_values in [(0.8,), (1.2,), (1.5,), (2.0,)]:
    pars, covm = curve_fit(fit_function, t, s, initial_values, ds)
    print(initial_values[0], pars[0])

[Output]
(0.8, 0.80112807399552199)
(1.2, 1.1977618985146361)
(1.5, 1.6164614506615154)
(2.0, 1.9491168598261097)
```

Interessante: quattro valori iniziali diversi per ω fanno convergere il fit a quattro valori finali completamente diversi. A dir poco spiazzante. (Prima di proseguire notiamo che la sintassi `(0.8,)` è permette di specificare in Python una tupla di lunghezza unitaria: anche nel caso si abbia un solo parametro, la funzione `curve_fit` di `scipy.optimize` si aspetta una tupla come quarto argomento.) Lo *script* che abbiamo appena visto ci suggerisce un modo ovvio per cercare di capire cosa sta succedendo: perché non calcoliamo il valore del χ^2 del nostro modello su una griglia predeterminata di valori di ω?

https://bitbucket.org/lbaldini/computing/src/tip/examples/fit_initial_val_chi2.py

```
import pylab
import numpy
from scipy.optimize import curve_fit

def fit_function(x, omega):
    """ Definition of the fit function.
    """
    return pylab.sin(omega*x)

```

```
# Load the data from the file.
t, s, ds = pylab.loadtxt('data/fit_initial_val.txt', unpack = True)

# Define a grid and calculate the values of the chisquare on the grid
# (note we are not fitting, here).
omega_grid = numpy.linspace(0, 10, 500)
chi2_values = []
for omega in omega_grid:
    chi2 = (((s - fit_function(t, omega))/ds)**2).sum()
    chi2_values.append(chi2)

# Make a plot of the chisquare as a function of omega.
pylab.rc('font', size = 18)
pylab.title('Chisquare scan', y = 1.02)
pylab.xlabel('omega [1/s]')
pylab.ylabel('chisquare', labelpad = 0)
pylab.grid(color = 'gray')
pylab.plot(omega_grid, chi2_values, color = 'black')

# Save the plot to a pdf file for later use (maybe in a writeup?).
pylab.savefig('fit_initial_val_chi2.pdf')

# Finally: show the plot on the screen.
pylab.show()
```

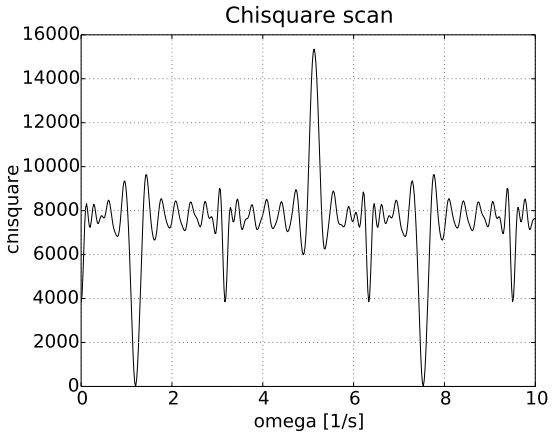

FIGURA 11. Valori del χ^2 del nostro modello (19), calcolati su una griglia a passo costante di valori di ω. Il minimo globale a $\omega \sim 1.198$ è la soluzione cercata.

Il risultato della nostra ricerca su griglia è mostrato in figura 11, che dovrebbe rendere abbastanza chiaro quel che sta accadendo: il χ^2 in funzione di ω ha moltissimi minimi locali, più o meno prominenti, e quando inizializziamo il nostro parametro ad un dato valore, il *fit* converge al valore più vicino a quel valore. Il minimo globale a $\omega \sim 1.198$ è la soluzione cercata al nostro problema[2], e nella pratica dobbiamo far attenzione ad inizializzare i parametri a valori *ragionevoli*. (In questo caso la stima iniziale di 1.2 che avevamo dato *ad occhio* è sufficiente.)

Di seguito un esempio completo di fit, con un valore iniziale corretto del parametro, che produce il grafico in figura 12.

https://bitbucket.org/lbaldini/computing/src/tip/examples/fit_initial_val_fitplot.py

```
import pylab
import numpy
from scipy.optimize import curve_fit

```

2 Sembra che ci sia un minimo altrettanto profondo a $\omega \sim 7.5$. Qualcuno ha un'idea di cosa potrebbe essere?

```
def fit_function(x, omega):
    """ Definition of the fit function.
    """
    return pylab.sin(omega*x)

# Load the data from the file.
t, s, ds = pylab.loadtxt('data/fit_initial_val.txt', unpack = True)

# Do the fit.
initial_values = (1.2,)
pars, covm = curve_fit(fit_function, t, s, initial_values, ds)
omega0 = pars[0]

# Plot the data points and the fit function.
pylab.rc('font', size = 18)
pylab.title('Law of motion fit', y = 1.02)
pylab.xlabel('t [s]')
pylab.ylabel('s [m]', labelpad = 5)
pylab.xlim(0, 20)
pylab.ylim(-1.5, 1.5)
pylab.grid(color = 'gray')
x = numpy.linspace(0, 20, 200)
y = fit_function(x, omega0)
pylab.errorbar(t, s, ds, linestyle = '', color = 'black', marker = 'o')
pylab.plot(x, y, color = 'black')

# Save the plot to a pdf file for later use (maybe in a writeup?).
pylab.savefig('fit_initial_val_fitplot.pdf')

# Finally: show the plot on the screen.
pylab.show()
```

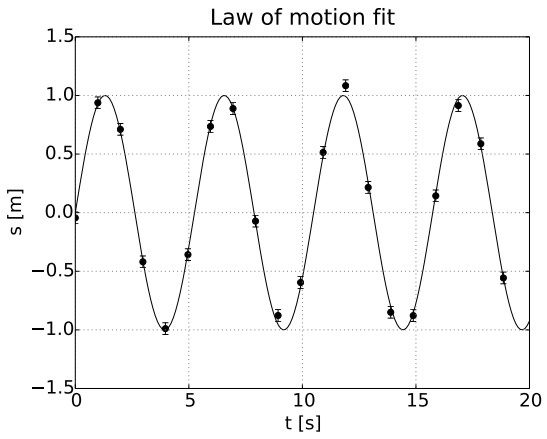

FIGURA 12. Fit ai nostri dati con un valore iniziale appropriato ($\omega = 1.2$).

15

SCRIVERE UNA RELAZIONE CON LATEX

LaTeX è un programma per la formattazione di documenti di testo, fortemente orientato alla matematica, che gode di una notevole popolarità nella comunità scientifica e tra i fisici in particolare. Quando si deve affrontare la redazione di un documento complesso e voluminoso, LaTeX ha, tra gli altri, l'indubbio vantaggio di rendere difficile il realizzarlo in modo tipograficamente mal strutturato. Un motivo sufficiente, quanto meno, per provarlo. In questo capitolo impareremo come si scrive e compila un documento LaTeX e come si introducono, all'interno del testo, liste, tabelle, equazioni e figure.

15.1 INTRODUZIONE

Un *file* LaTeX è un misto di testo e comandi che sostanzialmente *non* contiene informazioni di formattazione. Quest'ultima avviene invece in un secondo momento, quando il *file* stesso viene processato con un apposito programma. La realizzazione del prodotto finale, pronto per la stampa, passa attraverso le operazioni fondamentali elencate di seguito.

1) *Stesura del documento*. Qualsiasi editor di testo va bene allo scopo, anche se vale la pena notare che alcuni, sia sotto GNU/Linux che sotto Windowspossiedono funzionalità specifiche molto utili[1]. Per fissare le idee, se vogliamo creare un nuovo *file* LaTeX chiamato `relazione.tex` usando gedit sotto GNU/Linux, digiteremo nella *shell*:

```
> gedit relazione.tex &
```

2) *Compilazione*. Si esegue dal terminale[2] attraverso il comando `pdflatex`. Ad esempio per compilare il *file* `relazione.tex` digiteremo:

```
> pdflatex relazione.tex
```

In questo modo, a meno che non vi siano errori di sintassi all'interno del documento LaTeX (nel qual caso il compilatore si lamenterà e potrete tornare al terminale premendo `<CTRL + C>`), verrà prodotto un certo numero di *file* con diverse estensioni, tra cui il file in uscits `relazione.pdf`.

3) *Visualizzazione e stampa*. Il file prodotto da `pdflatex` è un file pdf pronto per la viasualizzazione e la stampa. Sotto GNU/Linux si può utilizzare, tra gli altri, il programma `evince` per entrambi:

```
> evince relazione.pdf
```

1 Tra questi ricordiamo Kile sotto GNU/Linux e TeXniCenter sotto Windows.

2 Per terminale intendiamo qui sia la *shell* di GNU/Linux che il *prompt* di DOS nel caso in cui si lavori sotto Windows. Notiamo anche esplicitamente che, nel caso in cui si utilizzino editor orientati a LaTeX come quelli citati prima, vi sono in generale appositi bottoni nelle rispettive interfacce grafiche per l'utente che permettono di eseguire questa operazione automaticamente.

15.2 IL PRIMO DOCUMENTO LATEX

Cominciamo dunque con il più semplice documento che si possa immaginare e scriviamo un *file* di testo contente le 4 linee seguenti:

```
\documentclass[11pt, a4paper]{article}
\begin{document}
Il mio primo documento\ldots
\end{document}
```

Il risultato della compilazione, essenzialmente pronto per essere stampato, è mostrato in figura 13.

Non è niente di esaltante, bisogna ammetterlo. Ma prima di proseguire e familiarizzare con funzionalità più avanzate, cerchiamo di capire il significato di ciascuna delle quattro righe del nostro documento. Una per una.

```
\documentclass[11pt, a4paper]{article}
```

Qui dichiariamo sostanzialmente il tipo di documento. Diciamo che si tratta di un documento del tipo *article*[3], che la dimensione dei caratteri del testo ordinario è 11 punti[4], e che la dimensione della pagina è quella di un foglio A4[5] (che è la scelta tipica in Europa).

```
\begin{document}
```

Segna l'inizio del documento vero e proprio. In generale questo comando separa quello che è chiamato *preambolo* (nel nostro caso è sostanzialmente la linea precedente) dal *corpo* vero e proprio del documento, che è *sempre* contenuto tra un comando di inizio ed un comando di fine documento.

```
Il mio primo documento\ldots
```

Si tratta del corpo del nostro documento, ed in effetti è la parte che compare nella versione pronta per la stampa, come si può vedere in figura 13.

```
\end{document}
```

Questa linea chiude il documento ed è sempre richiesta alla fine di un *file* LaTeX valido.

Nel prossimo paragrafo vedremo un esempio di documento più realistico e cominceremo ad apprezzare alcune funzionalità più utili ed avanzate.

15.3 UN DOCUMENTO REALISTICO

Consideriamo attentamente il documento LaTeX che segue e vediamo come appare l'uscita del compilatore, pronta per la stampa, che è riportata in figura 14.

```
\documentclass[11pt, a4paper]{article}
\title{Il mio primo documento}
\author{Luca Baldini}
```

3 Esistono moltissimi tipi diversi di documenti predefiniti in LaTeX tra cui *book*, *report*, *letter* e *slides*; altri possono essere aggiunti, anche se non si tratta esattamente della cosa più semplice del mondo. Il tipo *article* si presta piuttosto bene alla stesura di una relazione di media lunghezza su una esperienza didattica ed è questo il motivo per cui lo abbiamo scelto per i nostri esempi.

4 Esistono anche le varianti 10 e 12 punti.

5 Altri formati validi, solo per citarne un paio, sono *a5paper* e *letterpaper*.

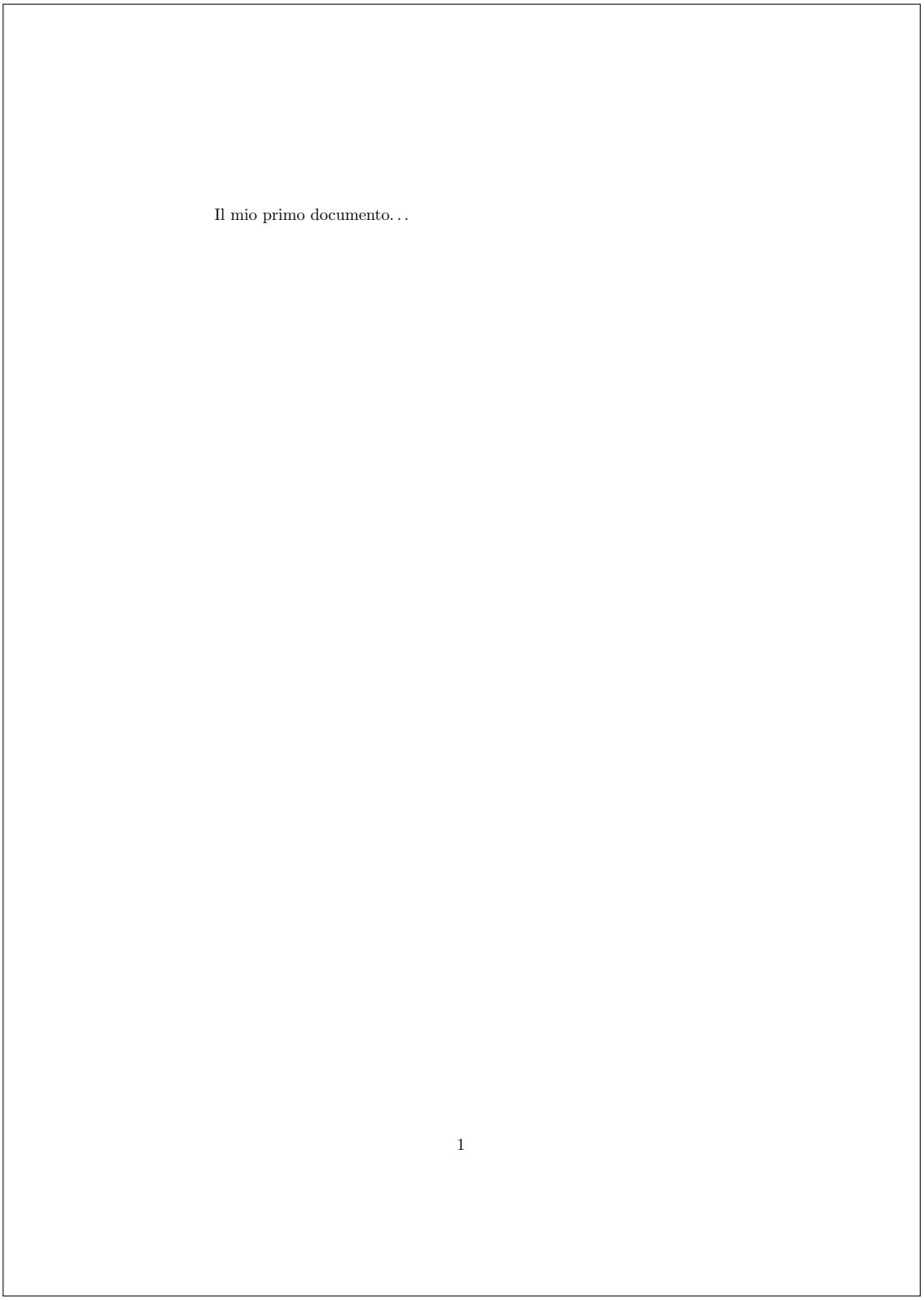

FIGURA 13. Risultato della compilazione del documento LaTeX riportato a pagina 64.

```
4 \date{20 Giugno 2006}
```

```
5
6  \begin{document}
7  \maketitle
8
9  \section{Introduzione}
10 Questo vuole essere un esempio di documento realistico, con lo scopo
11 di mostrare alcune tra le funzionali\'a di base di \LaTeX.
12
13 \section{Un nuovo capitolo}
14 Una prima cosa da notare \'e che, una volta dichiarate le sezioni del
15 documento, \LaTeX\ si occupa automaticamente della numerazione. Questo
16 pu\'o risultare estremamente utile nel caso in cui si voglia aggiungere
17 una nuova sezione in mezzo ad un documento in stato avanzato di stesura
18 (non \'e necessario rinumerare ci\'o che viene dopo, \LaTeX\ fa
19 tutto da solo).
20
21 \subsection{Una sotto-sezione}
22 Come vedete esiste anche un livello pi\'u \emph{basso} di sezionamento
23 del documento.
24
25 \subsection{Ancora una sottosezione}
26 Anche a questo livello la numerazione \'e automatica. Questo consente
27 anche, in documenti pi\'u complessi, la generazione automatica dell'indice.
28
29 \section{Conclusioni}
30 Per il momento ci fermiamo qui, abbiamo abbastanza di cui discutere\ldots
31
32 \end{document}
```

Cominciamo con alcuni commenti di carattere generale per poi passare all'analisi di alcuni tra i nuovi comandi che abbiamo appena incontrato.

Tutti i comandi LaTeX cominciano con il carattere "\" (*backslash*). Alcuni di essi accettano (o richiedono, a seconda del caso) *opzioni* (che sono sempre contenute tra parentesi quadre) e *argomenti* (racchiusi tra parentesi graffe). I comandi ed il testo sono, come è evidente, mischiati insieme all'interno del documento. È importante notare come gli spazi multipli siano trattati come spazi singoli e come i ritorni a capo siano semplicemente ignorati, se non che una linea vuota segna l'inizio di un nuovo paragrafo. Esiste ovviamente un comando per andare a capo ed è "\\" (doppio *backslash*).

Tra i comandi rivestono particolare importanza quelli di sezionamento, che sono utilizzati molto frequentemente. I documenti di tipo *article* ammettono i due comandi di sezionamento:

```
1 \section{}
2 \subsection{}
```

che accettano come argomento il titolo della sezione o della sotto-sezione, rispettivamente.

Una volta definito il tipo di documento, LaTeX si occupa automaticamente sia di numerare le sezioni e le sotto-sezioni che di fissare le dimensioni e le altre caratteristiche dei caratteri per il testo ordinario e per i titoli. Si tratta di un punto qualificante della filosofia di LaTeX: il documento non contiene, al suo interno, alcuna informazione riguardante la formattazione. Ovviamente è possibile fare ogni genere di cambiamento, ma questo avviene ordinariamente nel preambolo attraverso comandi specifici. Rimandiamo ai riferimenti bibliografici per questi argomenti, che sono, in un certo senso, più avanzati.

Veniamo adesso alla descrizione dei nuovi comandi che abbiamo introdotto.

Il mio primo documento

Luca Baldini

20 Giugno 2006

1 Introduzione

Questo vuole essere un esempio di documento realistico, con lo scopo di mostrare alcune tra le funzionalià di base di LaTeX.

2 Un nuovo capitolo

Una prima cosa da notare è che, una volta dichiarate le sezioni del documento, LaTeX si occupa atomaticamente della numerazione. Questo può risultare estrememente utile nel caso in cui si voglia aggiungere una nuova sezione in mezzo ad un documento in stato avanzato di stesura (non è necessario rinumerare ciò che viene dopo, LaTeX fa tutto da solo).

2.1 Una sotto-sezione

Come vedete esiste anche un livello più *basso* di sezionamento del documento.

2.2 Ancora una sottosezione

Anche a questo livello la numerazione è automatica. Questo consente anche, in documenti più complessi, la generazione automatica dell'indice.

3 Conclusioni

Per il momento ci fermiamo qui, abbiamo abbastanza di cui discutere...

1

FIGURA 14. Risultato della compilazione del documento LaTeX riportato a pagina 66.

```
\title{Il mio primo documento}
\author{Luca Baldini}
\date{20 Giugno 2006}
```

Questo blocco definisce, come è naturale aspettarsi, il titolo, l'autore e la data di composizione del documento, rispettivamente. Se il comando date non viene dato esplicitamente, LaTeX scrive automaticamente la data del giorno, mutuata dal tempo di sistema del calcolatore. Di *default* questa data viene scritta in lingua inglese, ma si può ovviare a questo inconveniente mettendo nel preambolo il comando "`\usepackage[italian]{babel}`". Se non si vuole la data è sufficiente scrivere "`\date{}`".

Vale la pena notare che, di per sé, questi comandi non hanno alcun effetto a meno che non siano seguiti, nel corpo del documento, dal comando:

```
\maketitle
```

I più attenti avranno notato come sono state *prodotte* le lettere accentate. Ad esempio i due comandi "`\'e`" e "`\'e`" generano, dopo la compilazione, i due caratteri "è" ed "é" (cogliamo l'occasione per ricordare che la lingua Italiana possiede due accenti distinti: quello grave e quello acuto). LaTeX offre supporto completo, previa dichiarazione di un apposito pacchetto nel preambolo, per la tastiera Italiana e, quindi, per le ordinarie lettere accentate. Il lettore interessato può trovare le informazioni relative nella documentazione specifica.

15.4 ELENCHI

Molte delle esigenze più avanzate del semplice testo vengono gestite in LaTeX attraverso specifici *ambienti* predefiniti. Tecnicamente un ambiente è una porzione di documento inclusa tra un comando "`\begin{nome_ambiente}`" ed un comando "`\end{nome_ambiente}`".

Esistono *ambienti*, ad esempio, per la creazione di elenchi puntati e numerati. Le seguenti linee:

```
\begin{itemize}
\item Punto primo.
\item Punto secondo.
\end{itemize}
```

producono, dopo la compilazione:

- Punto primo.
- Punto secondo.

D'altra parte questo frammento:

```
\begin{enumerate}
\item Punto primo.
\item Punto secondo.
\end{enumerate}
```

genera:

1. Punto primo.
2. Punto secondo.

15.5 TABELLE

Le tabelle si costruiscono, secondo la filosofia di LaTeX, a partire da elementi *semplici*: essenzialmente linee e caratteri come mostrato nel frammento di codice che segue:

```
\begin{tabular}{cc}
\hline
Colonna 1 & Colonna 2 \\
\hline
\hline
a & b \\
c & d \\
\hline
\end{tabular}
```

e che, compilato, genera:

Colonna 1	Colonna 2
a	b
c	d

Analizziamo in dettaglio queste poche righe una alla volta. La prima:

```
\begin{tabular}{cc}
```

è piuttosto densa di significato. Sostanzialmente segna l'inizio dell'ambiente `tabular` e dichiara una tabella di due colonne. Ognuna delle due "`c`" sta qui per *center* e sta ad indicare che il contenuto della colonna corrispondente sarà centrato. È possibile allineare a destra o a sinistra il contenuto delle colonne usando "`r`" (*right*) o "`l`" (*left*), rispettivamente.

Le linee orizzontali debbono essere esplicitamente dichiarate tramite il comando "`\hline`" ed il contenuto vero e proprio della tabella è strutturato come nella riga seguente:

```
a & b \\
```

Il carattere "`&`" serve da separatore tra celle contigue appartenenti alla stessa linea, mentre il doppio *backslash* ("`\\`") funge da terminatore di linea.

Vale la pena, trovandoci in argomento, notare che con poche righe in più:

```
\begin{table}[!htb]
\begin{center}
\begin{tabular}{cc}
\hline
Colonna 1 & Colonna 2 \\
\hline
\hline
a & b \\
c & d \\
\hline
\end{tabular}
\caption{Questa \‘e una tabella di esempio.}
\end{center}
\end{table}
```

Colonna 1	Colonna 2
a	b
c	d

TABELLA 1. Questa è una tabella di esempio.

è possibile ottenere un risultato estremamente più appagante da un punto di vista estetico (cfr. tabella 1). Esaminiamo le differenze una per una. La più evidente è che adesso la tabella è centrata. Questo si ottiene banalmente includendola all'interno dei comandi:

```
\begin{center}
\end{center}
```

I più attenti avranno anche notato che abbiamo introdotto un ulteriore nuovo ambiente attraverso le due linee:

```
\begin{table}[!htb]
\end{table}
```

Questo ha due (benefici) effetti distinti. Il primo è che diventa possibile aggiungere una didascalia (automaticamente numerata) alla tabella attraverso il comando:

```
\caption{}
```

Il secondo (e più importante) è che la tabella diventa un oggetto *flottante*, cioè la sua posizione all'interno del documento pronto per la stampa (dopo la compilazione) non è più fissata. L'ambiente `table` è in effetti il primo esempio che incontriamo di ambiente flottante; ne vedremo almeno un altro (`figure`) nel seguito.

Adesso esaminiamo di nuovo la linea:

```
\begin{table}[!htb]
```

e cerchiamo di capire il significato dell'opzione "`!htb`" che abbiamo passato al comando. Le tre lettere "`h`" (*here*), "`t`" (*top*) e "`b`" (*bottom*), in ordine, dicono a LaTeX dov'è che vorremmo il nostro oggetto flottante; in questo caso LaTeX proverà ad inserirlo esattamente dove esso è definito nel corpo del testo (*here*) e, se questo non fosse possibile, lo inserirà all'inizio (*top*) o alla fine (*bottom*) della prima pagina successiva disponibile. Il punto esclamativo ("`!`") rafforza l'opzione. L'opzione "`!htb`" dovrebbe essere sufficiente per la maggior parte delle applicazioni non troppo avanzate—per quello che può contare tutte le figure di queste dispense sono state inserite in questo modo.

15.6 LATEX E LA MATEMATICA

Veniamo adesso ad uno dei motivi per cui LaTeX ha riscosso tanto successo all'interno della comunità scientifica: l'estensivo supporto che fornisce alla scrittura di simboli ed equazioni, che lo rende particolarmente indicato per la scrittura di articoli scientifici.

LaTeX offre numerosi ambienti per la matematica; ne citeremo qui solamente due: `math` ed `equation`. Il primo permette di inserire simboli matematici all'interno del testo e si apre e chiude essenzialmente con un simbolo "`$`". Ad esempio:

```
$\sigma_{x} = \sqrt{\sigma_{x}^{2}}$
```

diviene, dopo la compilazione, $\sigma_x = \sqrt{\sigma_x^2}$. Notiamo, per inciso, il comando "`\sqrt{}`" che permette di inserire radici quadrate di espressioni. L'esempio illustra anche come inserire lettere greche e caratteri in apice e pedice.

L'ambiente equation permette di scrivere vere e proprie equazioni (numerate), separate dal corpo del testo[6]. Ad esempio:

```
\begin{equation}\label{osc_armonico}
m \frac{d^{2}x}{dt^{2}} = m \ddot x = -k x
\end{equation}
```

diviene:

$$m\frac{d^2x}{dt^2} = m\ddot{x} = -kx \tag{20}$$

Il lettore potrà essere incuriosito dal comando "`\label{}`"; sostanzialmente esso permette di mettere un'etichetta in corrispondenza di un oggetto[7], o in altre parole di assegnargli un nome comprensibile di modo che in seguito, nel corpo del testo, ci si possa riferire ad esso in modo univoco con l'apposito comando "`\ref{}`". Se, ad esempio, adesso scriviamo:

```
la (\ref{osc_armonico}) \'e l'equazione di moto di un oscillatore armonico.
```

dopo la compilazione ciò che otteniamo è: la (20) è l'equazione di moto di un oscillatore armonico. I due comandi "`\label{}`" e "`\ref{}`" sono incredibilmente utili.

Gli spazi all'interno degli ambienti matematici vengono ignorati. LaTeX ha un suo algoritmo per determinare automaticamente le spaziature necessarie tra i vari oggetti in modalità matematica.

Chiudiamo il paragrafo con un'ultima equazione, che mostra un certo numero di comandi utili:

```
\begin{equation}
\left( \int_{-\infty}^{\infty} e^{-x^{2}} dx \right)^{2} =
\sum_{n=0}^{\infty}\frac{(-1)^{n}}{2n + 1} =
\prod_{n=1}^{\infty} \left( \frac{n + 1}{n} \right)^{(-1)^{n-1}} =
\pi
\end{equation}
```

che appare come:

$$\left(\int_{-\infty}^{\infty} e^{-x^2}\,dx\right)^2 = \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} = \prod_{n=1}^{\infty} \left(\frac{n+1}{n}\right)^{(-1)^{n-1}} = \pi \tag{21}$$

Rimandiamo il lettore alla documentazione specifica per una trattazione esaustiva delle funzionalità offerte da LaTeX in modalità matematica.

15.7 INSERIRE LE FIGURE

Eccoci all'ultimo argomento di questa breve rassegna sulle funzionalità di base di LaTeX: l'inserimento delle figure. La prima cosa che dobbiamo fare è includere un pacchetto specifico, e questo si fa inserendo all'interno del preambolo il comando:

```
\usepackage{graphicx}
```

6 Non è permesso (provare per credere) lasciare linee vuote all'interno dell'ambiente *equation*.

7 In questo caso l'oggetto in questione è un'equazione, ma il tutto funziona altrettanto bene con tabelle, figure, titoli di sezioni e paragrafi etc.

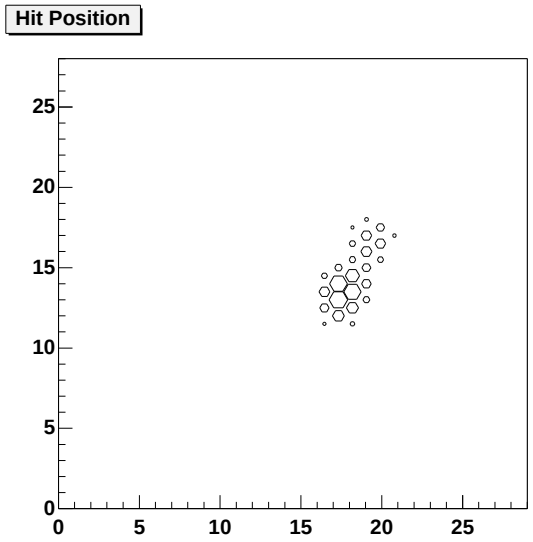

FIGURA 15. Questa è una figura di esempio.

Fatto questo, grafici ed immagini in formato pdf o png possono essere inseriti con poche linee (il risultato della compilazione appare in figura 15):

```
\begin{figure}[!htb]
\begin{center}
\includegraphics[width=0.5\textwidth]{test}
\end{center}
\caption{Questa \‘e una figura di esempio.}
\label{figura_di_esempio}
\end{figure}
```

Sostanzialmente non vi è niente di nuovo se non l'ambiente (flottante) `figure`, per cui vale quanto detto a proposito di `table`, ed il comando:

```
\includegraphics[width=0.5\textwidth]{test}
```

cui è passato il nome del *file* pdf (l'estensione non è necessaria) come argomento e la larghezza con cui l'immagine in esso contenuta deve apparire nel documento. Notiamo, per inciso, che "`\includegraphics`" è il comando che richiede il pacchetto `graphicx`.

A

LA SHELL DI GNU/LINUX E TUTTO IL RESTO

A.1 CONCETTI DI BASE

GNU/Linux è un sistema operativo *multi-utente* e *multi-tasking* cioè permette a più di un utente di accedere alla stessa macchina contemporaneamente—condividendo dati ed applicazioni—e supporta l'esecuzione concorrente di più *processi*.

A.1.1 *Il login*

Essendo GNU/Linuxun sistema operativo multi-utente, l'accesso al sistema è subordinato ad una procedura di identificazione che prende il nome di *login*. A tale scopo, tipicamente, l'utente è tenuto a fornire il proprio *username* e la propria *password* per eseguire il *login*) alla macchina.

GNU/Linux (al contrario di DOS e di Windows, come probabilmente qualcuno sa) distingue maiuscole e minuscole (cioè è case sensitive*) per cui il nome utente, la* password *ed i comandi in generale devono essere digitati esattamente come sono, con lettere maiuscole e minuscole al loro posto.*

A.1.2 *La shell ed i comandi*

La *shell*[1] , o terminale, è il luogo in cui si impartiscono i comandi; è, per certi aspetti, l'equivalente del *prompt* di DOS, per chi ha esperienza al proposito.

L'uso della *shell* consiste essenzialmente nello scrivere un comando GNU/Linux (valido) e nel premere il tasto di <INVIO>. Ogni comando può avere delle opzioni (che sono precedute da un carattere "-" e ne determinano l'esito) e può accettare dei parametri; in sostanza la struttura tipica di un comando GNU/Linux è del tipo[2]:

```
1 > comando -opzione argomento
2 >
```

Prima di andare avanti è utile ricordare che ci sono alcuni modi per riprendere il controllo della *shell* nel caso questo dovesse momentaneamente sfuggire (il che accade tipicamente quando si esegue un comando che si aspetta l'inserimento di dati da parte dell'utente oppure quando si avvia un programma la cui esecuzione impiega più tempo del previsto):

1) <CTRL + c>: solitamente elimina ogni programma in esecuzione e riporta alla linea di comando.

2) <CTRL + z>: sospende il processo corrente mentre è in esecuzione e riporta alla linea di comando.

1 Esistono, in effetti, svariati tipi di *shell*, ma per l'uso di base che ne faremo noi le differenze sono sostanzialmente irrilevanti.
2 Notiamo per chiarezza che, qui e nel seguito del capitolo, i comandi da digitare sono costituiti da tutto (e solo) ciò che segue il carattere ">", la cui unica funzione è quella di segnalare che ci troviamo nella *shell* GNU/Linux.

A.1.3 *Il logout*

Il processo di *logout* chiude tutti i *file* che sono (per qualche ragione) aperti e termina ogni programma mandato in esecuzione dall'utente. È buona regola scollegarsi sempre, una volta che abbiamo finito di utilizzare il computer; questo permette agli altri utenti, eventualmente collegati da un terminale remoto, di utilizzare nel modo più efficiente possibile le risorse di sistema. Si può avviare la procedura di *logout* (ed eventualmente quella di spegnimento della macchina) accedendo all'apposito menu con il mouse.

A.2 IL *filesystem* GNU/LINUX

A differenza di ciò che accade sotto Windows (in cui le varie unità, interne o esterne, di memorizzazione permanente dei dati sono associate alle lettere A, B, C...), il *filesystem* di GNU/Linux ha un'unica *directory* radice (che si chiama *root* e che si indica con il simbolo "/"). La *directory root* può contenere *file* ed altre *directory* e le *directory* dei livelli sottostanti possono a loro volta fare altrettanto. La struttura fisica dell'*hardware* è in un certo senso *nascosta* all'utente: tutte le unità di archiviazione appaiono semplicemente come sotto-*directory* della *directory root*.

Ogni utente ha inoltre una *home directory*, che non ha niente di particolare a parte il fatto di essere quella in cui l'utente stesso si trova immediatamente dopo il *login*.

A.2.1 *File e permessi*

Sotto GNU/Linux i *file* sono caratterizzati da tre attributi fondamentali: "`r`", "`w`" ed "`x`" (che stanno per *read*, *write* ed *execute*); come dire che un *file* si può leggere, scrivere ed eseguire. O meglio che un determinato utente può avere (o non avere) i privilegi necessari per leggere, scrivere od eseguire un determinato *file*. Qualsiasi combinazione di questi tre attributi è ammissibile per il sistema operativo, anche se non tutte sono propriamente sensate.

Ogni *file* GNU/Linux ha un proprietario (che è un utente con un account valido sulla macchina) ed ogni utente appartiene ad uno o più gruppi di utenti. I permessi su ogni *file* sono settati su tre livelli: quello del proprietario del *file*, quello del gruppo a cui appartiene il proprietario e quello di tutti gli altri utenti del sistema; tanto per fare un esempio, è possibile che un *file* sia leggibile, scrivibile ed eseguibile dal proprietario, solamente leggibile dal gruppo del proprietario ed assolutamente inaccessibile a tutti gli altri. Vedremo tra poco come è possibile visualizzare (ed eventualmente cambiare) i permessi relativi ad un determinato *file* o ad una *directory*.

A.3 NAVIGARE IL FILESYSTEM

In questa sezione vedremo alcuni comandi dedicati alla navigazione nel *filesystem*—lo stretto necessario per sopravvivere.

A.3.1 `pwd`

È l'acronimo di *present work directory* e restituisce sullo schermo il nome della *directory* corrente. È sufficiente digitare `pwd` per sapere dove siamo; ad esempio:

```
> pwd
/data/work/teaching/computing
>
```

A.3.2 `ls`

Elenca semplicemente i *file* contenuti in una certa *directory* (più o meno l'equivalente di `dir` in DOS); se l'utente non passa alcun argomento al comando, elenca i *file* nella *directory* corrente. Ad esempio:

```
> ls
Ciao.doc
Temp
ClusterAxes.eps
>
```

Nel caso specifico la *directory* contiene tre oggetti. A questo livello non siamo in grado di dire se essi sono *file* o *directory*. Per far questo è necessario invocare il comando con l'opzione "`-l`":

```
> ls -l
total 5
-rw-rw-r--   1  lbaldini glast     36352  Dec 20 18:18 Ciao.doc
drwxrwxr-x   2  lbaldini glast      2048  Jul  7 2002  Temp
-rw-rw-r--   1  lbaldini glast     21847  Jul 24 2002  Cluster.ps
>
```

La prima riga dice quanti sono gli oggetti nella *directory* (in questo caso cinque... capiremo tra un attimo il motivo per cui ne sono elencati solamente tre), mentre le linee successive visualizzano alcuni dettagli sugli oggetti in questione. Il primo campo indica se si tratta di un *file* ("`-`"), di una *directory* ("`d`") o di un link ("`l`"); di seguito compaiono i permessi per il proprietario, per il gruppo del proprietario e per l'utente generico ("`r`", "`w`" ed "`x`" indicano rispettivamente che l'utente possiede il permesso di lettura, scrittura ed esecuzione, mentre il simbolo "`-`" indica che il corrispondente permesso è disabilitato). Nel caso in questione "`Ciao.doc`", come indicato dalla stringa "`-rw-rw-r--`", è un *file* che il proprietario ed il suo gruppo possono leggere e modificare, mentre tutti gli altri utenti possono solamente leggere. Invece "`Temp`" è una *directory* e così via...

Il campo successivo nella linea indica il numero di *hard link* all'oggetto in questione, ma per noi è poco importante. Seguono dunque il nome del proprietario (in questo caso "`lbaldini`") e del gruppo ("`glast`"). Infine vediamo la dimensione dell'oggetto su disco, la data della sua ultima modifica ed il nome; da notare che, nel caso delle *directory*, la dimensione non corrisponde allo spazio su disco occupato dal contenuto, ma a quello occupato dall'unità logica che controlla la *directory* stessa.

In generale GNU/Linux non mostra all'utente (a meno che esso non lo richieda esplicitamente) tutti i *file* contenuti in una *directory*. In particolare il semplice comando `ls` nasconde i *file* e le *directory* che cominciano con il carattere"`.`" (generalmente si tratta di *file* di configurazione). Per visualizzare tutti i *file* si può utilizzare l'opzione "`-a`" di `ls`:

```
> ls -a
.
..
Ciao.doc
Temp
ClusterAxes.eps
>
```

Adesso compaiono anche gli oggetti "`.`" (che in GNU/Linux sta per la *directory* corrente) e "`..`" che viceversa indica la *directory* che nel *filesystem* si trova al livello immediatamente superiore rispetto a quella corrente (a volte detta anche *directory* genitrice).

Notiamo che l'oggetto "`.`" esiste in tutte le *directory* e che "`..`" esiste in tutte le *directory* fatta eccezione per la *directory root* ("`/`"), che si trova in assoluto al livello più alto nel *filesystem*.

Il comando `ls` può accettare un argomento, che indica la *directory* della quale si vogliono elencare gli oggetti:

```
> ls -a Temp
.
..
Luca.jpg
>
```

In questo caso ci dice che la *directory* "`Temp`" contiene (oltre ovviamente a "`.`" e "`..`") un solo *file* che si chiama "`Luca.jpg`".

Sotto GNU/Linux il tasto `<TAB>` *funziona da auto completion, cioè completa automaticamente i nomi di comandi od oggetti del filesystem, a patto che l'utente ne abbia scritto una parte abbastanza lunga da rimuovere eventuali ambiguità. Una volta provato non si può più farne a meno!*

A.3.3 `cd`

Esattamente come in DOS, questo comando serve per cambiare *directory*; necessita, come argomento, del nome della *directory* di destinazione. Il percorso verso quest'ultima può essere quello assoluto (cioè partendo da *root*) oppure quello relativo (cioè partendo dalla *directory* corrente). L'esempio seguente, che unisce i pochi comandi visti sino ad adesso, dovrebbe chiarire le idee:

```
> pwd
/data/work/teaching/computing
> ls
Ciao.doc
Temp
ClusterAxes.eps
> cd Temp
> ls
Luca.jpg
> cd .
> ls
Luca.jpg
> cd ..
> ls
Ciao.doc
Temp
ClusterAxes.eps
> cd /data/work/teaching/computing/Temp
> ls
Luca.jpg
>
```

Notiamo che il comando "`cd .`" non fa niente (essendo "`.`" la *directory* corrente), mentre "`cd ..`" porta nella *directory* immediatamente superiore nel *filesystem*.

A.4 MODIFICARE IL FILESYSTEM

Ovviamente navigare non è l'unica cosa che si può fare col *filesystem*. Esistono comandi per creare, copiare, muovere *file* e così via. Al solito diamo una rapida occhiata a quelli indispensabili.

A.4.1 mkdir

Come si può intuire dal nome, crea una *directory* e richiede come argomento il nome della *directory* da creare. Al solito il percorso può essere assoluto o relativo; ovviamente se non si specifica niente, il nuovo oggetto viene creato nella *directory* corrente. Ad esempio:

```
> pwd
/data/work/teaching/computing
> ls
Ciao.doc
Temp
ClusterAxes.eps
> mkdir NewDirectory
> ls
Ciao.doc
Temp
ClusterAxes.eps
NewDirectory
>
```

Sotto GNU/Linux è buona norma evitare gli spazi all'interno dei nomi di file o directory*. Lo spazio è un carattere molto particolare poiché costituisce il separatore tra comandi, opzioni ed argomenti e, quando si lavora dalla shell, gestire spazi nei nomi dei file può essere scocciante.*

A.4.2 rmdir

È il contrario del comando precedente e semplicemente rimuove una *directory* esistente, a patto che sia vuota:

```
> ls
Ciao.doc
Temp
ClusterAxes.eps
NewDirectory
> rmdir NewDirectory
> ls
Ciao.doc
Temp
ClusterAxes.eps
>
```

Come vedremo tra un attimo, per cancellare una *directory* non vuota è necessario usare il comando `rm` con l'opzione "`-r`".

A.4.3 rm

Rimuove il *file* che gli viene passato come argomento e la sintassi è analoga a quella di `rmdir`. Ha alcune opzioni che vale la pena di conoscere:

1) "`-i`": il comando agisce in modo interattivo e chiede conferma prima di cancellare ogni *file*; caldamente consigliata quando si lavora a notte fonda!

2) "`-f`": il comando agisce in modo forzato senza chiedere conferma.

3) "`-r`": il comando agisce in modo ricorsivo; con questa può accettare il nome di una *directory* come argomento ed in tal caso cancella ricorsivamente tutto il contenuto della *directory* stessa.

Può essere buona abitudine utilizzare d'abitudine il comando `rm` in modo interattivo (con l'opzione "`-i`").

A.4.4 `cp`

È l'analogo del comando `copy` in DOS e serve per copiare un *file* di partenza (che viene passato come primo argomento) in un *file* o in una *directory* di destinazione (che costituisce viceversa il secondo argomento). Ad esempio il comando:

```
> cp *.doc Temp
>
```

copia tutti i *file* il cui nome termina per "`.doc`" nella *directory* "`Temp`".

Il simbolo "`*`" è un carattere speciale (detto wildcard) che sostanzialmente sta per "qualsiasi sequenza di lettere e numeri". È la shell stessa che si occupa di operare la sostituzione, sulla base del contenuto del filesystem nel momento in cui il comando viene impartito.

L'esempio successivo:

```
> cp Ciao.doc Salve.doc
>
```

crea una copia del *file* "`Ciao.doc`" nella *directory* corrente chiamandola "`Salve.doc`".

A.4.5 `mv`

Serve per rinominare un *file*:

```
> mv Ciao.doc Salve.doc
>
```

oppure per spostarlo in una *directory* di destinazione, se quest'ultima viene specificata:

```
> mv Ciao.doc Temp
>
```

Per inciso notiamo che, dopo il primo comando, il *file* "`Ciao.doc`" non esiste più.

A.5 VISUALIZZARE I *file* DI TESTO

Al di là dei *file* di testo propriamente detti, molte applicazioni GNU/Linux (ed il sistema stesso) utilizzano *file* di configurazione di tipo testuale, per cui è essenziale avere strumenti per visualizzare (ed eventualmente modificare) gli oggetti di questo tipo.

GNU/Linux offre un ampio insieme di comandi dedicati alla visualizzazione dei *file* di tipo testuale. Di seguito elenchiamo i principali.

A.5.1 `more`

Accetta come argomento un *file* e ne visualizza il contenuto sullo schermo; per esempio

```
> more Cluster.eps
>
```

visualizza il contenuto del *file* "`Cluster.eps`". La barra spaziatrice fa avanzare di una pagina, mentre il tasto "`b`" riporta indietro di una pagina; si torna alla *shell* premendo il tasto "`q`". Le pagine `man` dovrebbero illustrare l'elenco completo dei comandi che si possono usare all'interno di more.

A.5.2 `less`

La sintassi ed il funzionamento sono essenzialmente identici a quelli di more:

```
> less Cluster.eps
>
```

La differenza è che dovrebbe essere più facile muoversi all'interno del documento (in particolare si possono usare le frecce direzionali).

A.5.3 `head`

Visualizza le prime 10 righe (ma il numero può essere modificato utilizzando l'opportuna opzione) del *file* che gli viene passato come argomento. La sintassi è la solita:

```
> head Cluster.eps
>
```

È utile quando si vuole dare una sola occhiata ad un *file*.

A.5.4 `tail`

Al contrario del precedente visualizza le ultime righe di un *file*.

A.6 MODIFICARE I FILE DI TESTO

Un buon editor di testo è un ingrediente fondamentale per una sessione di lavoro produttiva. emacs e gedit sono due ottime alternative disponibili sulla maggior parte dei sistemi GNU/Linux. Si può aprire un *file* di testo (nuovo o esistente) con emacs (o gedit) semplicemente digitando dalla *shell*:

```
> emacs sample.txt
```

o

```
> gedit sample.txt
```

(ovviamente sostituite "`sample.txt`" con il nome del file di cui avete bisogno). In realtà, come vedremo tra un attimo, è comune far seguire il nome del file da un "`&`" per lanciare l'editor in *background*.

A.7 I PROCESSI

Ogni comando GNU/Linux crea un *processo* che il sistema operativo esegue fino a quando esso non termina o viene interrotto. Il punto fondamentale è che, essendo multitasking, GNU/Linux permette di mandare in esecuzione più di un processo contemporaneamente; permette inoltre di mandare in esecuzione processi in *background* (o, come si dice, dietro le quinte) semplicemente facendo seguire un carattere "`&`" al comando.

Un esempio può essere utile per chiarire le idee. Supponiamo di voler aprire con gedit il *file* "`Cluster.txt`"; possiamo farlo normalmente:

```
> gedit Cluster.txt
```

oppure possiamo farlo in *background*:

```
> gedit Cluster.txt &
[2] 12563
>
```

La differenza, come si vede, è che nel secondo caso il sistema torna alla linea di comando e la *shell* è di nuovo utilizzabile, mentre nel primo la *shell* è inutilizzabile fino a che non chiudiamo gedit (anche se, a dirla tutta, possiamo sempre aprire una nuova *shell*). Per completezza, i numeri "`[2] 12563`" identificano il processo all'interno del sistema operativo.

GNU/Linux offre un *set* di comandi per monitorare i processi ed eventualmente intervenire su di essi. Al solito vedremo i fondamentali.

A.7.1 ps

Visualizza l'elenco dei processi in memoria dei quali l'utente è proprietario. Non servono argomenti e la sintassi è banale:

```
> ps
  PID TTY          TIME CMD
 1657 pts/0    00:00:36 evince
 3011 pts/0    00:01:12 emacs
 3796 pts/0    00:00:00 ps
16765 pts/0    00:00:05 bash
>
```

Il risultato è una lista dei processi con relative informazioni.

A.7.2 top

Visualizza un elenco, aggiornato continuamente, di tutti i processi in memoria, fornendo inoltre alcune informazioni importanti come il numero di identificazione, il proprietario, il tempo di esecuzione, la priorità, etc... Anche qui non servono argomenti

```
>top
top - 19:53:50 up 7 days,  7:48,  3 users,  load average: 0.15, 0.16, 0.20
Tasks: 147 total,   1 running, 146 sleeping,   0 stopped,   0 zombie
libnuma: Warning: /sys not mounted or invalid. Assuming one node: No such file or directory
%Cpu(s):  2.0 us,  0.4 sy,  0.0 ni, 97.4 id,  0.1 wa,  0.0 hi,  0.0 si,  0.0 st

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 1331 lbaldini  20   0  793296 342256  44820 S   6.5 10.2  64:13.35 gnome-shell
    1 root      20   0    8316   5392   3604 S   0.0  0.2   0:07.21 systemd
    2 root      20   0       0      0      0 S   0.0  0.0   0:00.13 kthreadd
    3 root      20   0       0      0      0 S   0.0  0.0   0:01.81 ksoftirqd/0
...
```

Si torna alla *shell* digitando <CTRL + c>.

A.7.3 `kill`

Capita a volte, per qualche ragione, di voler interrompere forzatamente un processo; in tal caso si utilizza il comando `kill`, che accetta come parametro il numero di identificazione del processo da eliminare. Ad esempio per uccidere il processo avviato quando prima avevamo lanciato gedit in background dovremmo fare:

```
> kill -9 12563
[2]+  Killed     gedit
>
```

B

ALCUNI PROGRAMMI DA UTILIZZARE IN LABORATORIO

B.1 SIMULAZIONE DEL LANCIO DI DADI

Questo script di esempio legge direttamente il file di uscita del programma dice in plasuino e disegna un *bar plot*.

https://bitbucket.org/lbaldini/computing/src/tip/lab/plasduino_dice.py

```
import pylab

# Definizione di alcune variabili (da cambiare a piacimento).
n_dadi  = 2
n_lanci = 1000

# Lettura del file di dati in ingresso (da modificare con il percorso al
# vostro file in ingresso).
valori, occorrenze = pylab.loadtxt( 'data/plasduino_dice.txt', unpack = True)

# Calcolo della media e varianza campione.
media_campione = sum(occorrenze*valori)/n_lanci
varianza_campione = sum(occorrenze*((valori - media_campione)**2))/(n_lanci - 1)
dev_standard_campione = pylab.sqrt(varianza_campione)
print('m = %f, s = %f' % (media_campione, dev_standard_campione))

# Realizzazione e salvataggio del grafico.
pylab.title('%d lanci di %d dado/i' % (n_lanci, n_dadi))
pylab.xlabel('Somma delle uscite')
pylab.ylabel('Occorrenze')
pylab.grid()
pylab.bar(valori, occorrenze, 1)
pylab.savefig('dice_%d_%d.pdf' % (n_dadi, n_lanci))

pylab.show()

[Output]
m = 6.996000, s = 2.385304
```

B.2 PENDOLO QUADRIFILARE

Questo script legge il file di uscita del programma pendulum in plasuino e mostra un esempio di fit (velocità nel punto più basso in funzione del tempo) ed un esempio di grafico dei residui con un modello teorico atteso (periodo in funzione dell'ampiezza di oscillazione).

https://bitbucket.org/lbaldini/computing/src/tip/lab/plasduino_pendulum.py

```
import pylab
from scipy.optimize import curve_fit

# Lettura del file di dati in ingresso (da modificare con il percorso al
# vostro file in ingresso).
t, T, Tr = pylab.loadtxt('data/pendulum.txt', unpack = True)

# Definizione della geometria del pendolo [cm]
w = 2.4
l = 113.0
d = 115.0

# Calcolo di velocita' e angolo
v = (w/Tr)*(l/d)
Theta = pylab.arccos(1. - (v**2)/(2*980.*l))

# Funzioni di fit per velocita' e angolo
def f_v(x, v0, tau):
    return v0*pylab.exp(-x/tau)

def f_Theta(x, p1, p2):
    return 2*pylab.pi*pylab.sqrt(l/980.)*(1+ p1*(x**2) + p2*(x**4))

# Fit di V vs t
popt_v, pcov_v = curve_fit(f_v, t, v, pylab.array([500., 100.]))
v0_fit,   tau_fit = popt_v
dv0_fit, dtau_fit = pylab.sqrt(pcov_v.diagonal())
print('v0  = %f +- %f cm/s' % (v0_fit,  dv0_fit))
print('tau = %f +- %f s' % (tau_fit, dtau_fit))

# Plot di V vs t
pylab.figure(1)
pylab.xlabel('Tempo [s]')
pylab.ylabel('v [cm/s]')
pylab.grid(color = 'gray')
pylab.plot(t, v, '+', label='data')
pylab.plot(t, f_v(t, v0_fit, tau_fit), label='fit')
pylab.legend()
pylab.savefig('pendolo_VvsT.png')

# Plot di T vs Theta e dei residui rispetto alla funzione attesa
pylab.figure(2)
pylab.subplot(211)
pylab.ylabel('Periodo [s]')
pylab.grid(color = 'gray')
pylab.plot(Theta, T, 'o', label='data')
pylab.plot(Theta, f_Theta(Theta, 1./16, 11./3072), 'r', label='modello atteso')
pylab.legend(loc=2)
pylab.subplot(212)
pylab.xlabel('Angolo [rad]')
pylab.ylabel('Periodo data - modello [ms]')
pylab.plot(Theta, 1000*(T-f_Theta(Theta, 1./16, 11./3072)))
```

```
pylab.grid(color = 'gray')

pylab.savefig('pendolo_TvsTheta.png')

pylab.show()

[Output]
v0  = 147.869046 +- 0.068777 cm/s
tau = 462.492975 +- 0.555136 s
```

B.3 PENDOLO ANALOGICO

Questo script di esempio legge il file di uscita del programma pendulumview in plasuino e disegna il grafico dell'ampiezza in funzione del tempo e lo fitta in un intervallo.

https://bitbucket.org/lbaldini/computing/src/tip/lab/plasduino_pendulumview.py

```
import pylab
from scipy.optimize import curve_fit

# Lettura del file di dati in ingresso (da modificare con il percorso al
# vostro file in ingresso).
t, A, B = pylab.loadtxt('data/analog_pendulum.txt', unpack = True)

# Selezione dell'intervallo di tempo per il fit:
t_min = 100
t_max = 150

# Extrazione dei dati corrispondenti all'intervallo di tempo selezionato
t1 = t[t>t_min];  st = t1[t1<t_max];
A1 = A[t>t_min];  sA = A1[t1<t_max];
B1 = B[t>t_min];  sB = B1[t1<t_max];

# Per prima cosa guardiamo i grafici dei due pendoli
pylab.figure(1)
pylab.subplot(211)
pylab.ylabel('Pendolo A [u.a.]')
pylab.grid(color = 'gray')
pylab.plot(t, A)
pylab.plot(st, sA, 'r')
pylab.subplot(212)
pylab.xlabel('Tempo [s]')
pylab.ylabel('Pendolo B [u.a.]')
pylab.grid(color = 'gray')
pylab.plot(t, B, 'g')
pylab.plot(st, sB, 'r')
pylab.show()

# Funzione di fit: oscillatore armonico smorzato
def f(x, A, omega, tau, p, c):
    return A*pylab.exp(-x/tau)*pylab.cos(omega*x +p) + c

# Inizializzazione dei parametri per il pendolo A
# Attenzione: partire da un set di parametri appropriati e' fondamentale
#             per la convergenza del fit
initParamsA = pylab.array([500., 4.4, 70., 0., 480.])

# Fit del pendolo A nell'intervallo selezionato
popt_A, pcov_A = curve_fit(f, st, sA, initParamsA)
```

```
A_fit,   omega_fit_A,  tau_fit_A,  p_fit_A,  c_fit_A = popt_A
dA_fit, domega_fit_A, dtau_fit_A, dp_fit_A, dc_fit_A = pylab.sqrt(pcov_A.diagonal())
print('Omega = %f +- %f rad/s' % (omega_fit_A, domega_fit_A))
print('Tau   = %f +- %f s'     % (tau_fit_A,   dtau_fit_A))

# il fit del pendolo B si esegue allo stesso modo....

# Realizzazione e salvataggio del grafico A
pylab.figure(2)
pylab.xlabel('Tempo [s]')
pylab.ylabel('Pendolo A [u.a.]')
pylab.grid(color = 'gray')
pylab.plot(st, sA, '+', label='data')
pylab.plot(st, f(st, A_fit,   omega_fit_A,  tau_fit_A,  p_fit_A,  c_fit_A), label='fit')
pylab.legend()
pylab.savefig('pendoloA_fit.png')
pylab.show()

[Output]
Omega = 4.445431 +- 0.000028 rad/s
Tau   = 79.419239 +- 0.177359 s
```

B.4 INDICI DI RIFRAZIONE

Questo script di esempio utilizza due set di misure definite come *array* (relative all'esperienza sugli indici di rifrazione di acqua e plexiglass), disegna due *errorbar plot* ed esegue i fit con la funzione `curve_fit` del modulo `scipy.optimize`.

https://bitbucket.org/lbaldini/computing/src/tip/lab/refraction_index.py

```
import pylab
from scipy.optimize import curve_fit

# Definizione delle variabili con i dati in ingresso per il diottro
# (da modificare con i vostri dati).
p = pylab.array([45.6, 42.9, 41.0, 38.7, 35.8, 34.1], 'd')
q = pylab.array([43.5, 47.6, 51.2, 56.4, 68.1, 75.4], 'd')
Dp = pylab.array(len(p)*[0.5],'d')
Dq = pylab.array(len(q)*[1],'d')

# Plot di 1/q vs 1/p
pylab.figure(1)
pylab.title('Indice di rifrazione dell\'acqua')
pylab.xlabel('1/p [1/cm]')
pylab.ylabel('1/q [1/cm]')
pylab.grid(color = 'gray')
pylab.errorbar(1./p, 1/q, Dp/(p*p), Dq/(q*q), 'o', color='black' )

# Fit con una retta - nota che le incertezze sono ignorate!
def f(x, a, b):
    return a*x + b

popt, pcov = curve_fit(f, 1./p, 1/q, pylab.array([-1.,1.]))
a, b       = popt
da, db     = pylab.sqrt(pcov.diagonal())
print('Acqua:')
print('n = %f +- %f' % (a, da))
print('b = %f +- %f' % (b, db))
pylab.plot(1./p, f(1./p, a, b), color='black' )
```

```
pylab.savefig('rifrazione_acqua.png')

# Definizione delle variabili con i dati in ingresso per il plexiglass
# x = R*sin(theta_r), y = R*sin(theta_i) [cm]
x = pylab.array([1.25, 1.85, 2.9, 4.35, 0.5,  0.25, 0.8, 1.5, 4.8], 'd')
y = pylab.array([1.9,  2.5,  4.2, 6.65, 0.75, 0.45, 1.1, 2.4, 7. ], 'd')
Dx = pylab.array(len(x)*[0.1],'d')
Dy = pylab.array(len(y)*[0.1],'d')

# Plot di x vs y
pylab.figure(2)
pylab.title('Indice di rifrazione del plexiglass')
pylab.xlabel('R sin(theta_r) [cm]')
pylab.ylabel('R sin(theta_i) [cm]')
pylab.grid(color = 'gray')
pylab.errorbar(x, y, Dx, Dy, 'o', color='black' )

# Fit con una retta per essere sicuri che il termine noto
# sia compatibile con zero
popt, pcov = curve_fit(f, x, y, pylab.array([1.,0.]))
a, b       = popt
da, db     = pylab.sqrt(pcov.diagonal())
print('Plexiglass:')
print('a = %f +- %f' % (a, da))
print('b = %f +- %f compatibile con 0?' % (b, db))

# Fit con la legge di Snell
def f1(x, a):
    return a*x

popt, pcov = curve_fit(f1, x, y, pylab.array([1.]))
print('n = %f +- %f' % (popt, pylab.sqrt(pcov.diagonal())))
pylab.plot(x, f1(x, a), color='black' )
pylab.savefig('rifrazione_plexiglass.png')

pylab.show()

[Output]
Acqua:
n = -1.326998 +- 0.026800
b = 0.051979 +- 0.000686
Plexiglass:
a = 1.478984 +- 0.033340
b = 0.003611 +- 0.085045 compatibile con 0?
n = 1.480106 +- 0.019011
```