Matematica e Informatica in Python

Massimiliano Virdis

versione 20.10.2019

Indice

1 Introduzione

1.1 Premessa

Caro lettore,

alla fine degli anni ’80 ho cominciato ad interessarmi di informatica. Alcuni dei testi allora in commercio erano dedicati a studenti delle superiori o dei primi anni dell’università. L’oggetto di studio era l’utilizzo consapevole e mirato dei calcolatori per la matematica e la fisica. Alcuni di questi testi, tra quelli che ho avuto per le mani, sono ancora oggi tra i miei preferiti. L’intento e gli argomenti affrontati in questi testi è ancora attuale. Quello che è ampiamente superato sono i linguaggi di programmazione utilizzati (prevalentemente il basic e qualche volta il pascal). Ancora più datata è la scrittura dei programmi utilizzati: si preferiva un codice compatto ed efficiente ad uno più prolisso, meno efficiente, ma più leggibile e comprensibile.

Da qui è nato il desiderio di scrivere un testo comprensibile ad un buon studente delle superiori che sia interessato a scoprire degli aspetti della matematica non facilmente affrontabili se non con l’ausilio di un computer. L’interesse per l’informatica accompagnata tutto il discorso.

Spero che quanto riportato in quest’opera sia se non di aiuto almeno non dannoso. Per migliorare quanto scritto e evidenziare qualsiasi errore non esitate a scrivermi.

email: prof.virdis@tiscali.it

1.2 Licenza e Copyright

2 Numeri primi

2.1 Definizione

Un numero intero è detto primo se è divisibile solo per uno e per se stesso; uno non è considerato numero primo. Fin dal ’700 ci si pose il problema di come fossero distribuiti questi numeri, cioè quale relazione ci fosse tra lo scorrere dei numeri interi e la presenza di numeri primi. Ancor oggi non è stata trovata una precisa relazione, ma soltanto delle linee di tendenza. Il problema è stato affrontato in innumerevoli modi fino ai nostri giorni ed è ancora un intenso campo di studio.

Vogliamo anche noi studiare questo problema in maniera empirica. Ci proponiamo quindi:

1. di scrivere una lista di numeri primi;
2. di rappresentare nel piano cartesiano l’insieme di numeri trovato.

Per quanto riguarda il primo aspetto possiamo procedere in due modi.

1. Trovare quali sono i numeri primi in un certo intervallo di numeri interi.
2. Trovare un modo per controllare se un numero è primo oppure no. Poi possiamo applicare questo test all’insieme di numeri interi che ci interessa.

Le due tecniche sembrano del tutto analoghe. Tuttavia l’approccio è molto differente. Con la prima tecnica possiamo avere i numeri primi solo alla fine della procedura. Con la seconda abbiamo subito una risposta. La prima tecnica è spesso usata in quelli che si chiamano crivelli, il più famoso dei quali è quello di Eratostene. La seconda tecnica è usata soprattutto per sapere se un grande numero è primo o no ed è una procedura di grande importanza nelle tecniche di criptografia.

Proviamo ad applicare il secondo approccio, applicando la definizione di numero primo per tutti i numeri in un certo intervallo.

2.2 Test di primalità

Per scoprire se un numero è primo applichiamo la definizione. Prendiamo un numero iniziale n e lo dividiamo per tutti i numeri interi (diversi da uno) ad esso inferiori. Se una di questa divisioni dà come resto 0 allora il numero non è primo; altrimenti è primo. Queste divisioni saranno più frequenti con divisori piccoli: per questo conviene cominciare la ricerca con i numeri più piccoli.

Creiamo una funzione che controlli se un numero è primo.

```
def test_primalita(n):
    # suppongo che n sia primo
    p=1
    for k in range(2,n):
        # divido n per tutti gli interi ad esso inferiori
        # e controllo se la divisione è possibile
        # cioé se il resto della divisione è zero
        if (n%k==0):
            # se avviene così il numero n non è primo
            p=0
            # ed esco dal loop
            break
    # altrimenti se faccio tutto il loop senza mai uscirne
    # il numero è veramente primo
    return p
```

Per testare le prestazioni di questo algoritmo usiamolo per creare una lista di primi. Per tutti gli interi fino ad un dato valore eseguo il test di primalità. Se il numero trovato è primo lo accodo alla mia lista.

```
# creo una lista che contiene i primi due numeri primi
primi=[2,3]
def test_primalita(n):
    p=1
    for k in range(2,n):
        if (n%k==0):
            p=0; break
    return p
for i in range(5,1000):
    # per tutti gli interi successivi a 4 controllo se sono primi
    p=test_primalita(i)
    if p==1: # se lo sono
        primi.append(i) # gli aggiungo alla lista
# stampo quanti primi ho trovato
print(len(primi))
```

Questo programma impiega 1455 secondi per trovare i primi tra i naturali inferiori al milione. Questo algoritmo è però veramente brutto. Infatti, a una breve analisi, ci possiamo rendere conto che un numero primo non può che essere dispari (a parte 2). Quindi possiamo dimezzare la ricerca limitandoci ai numeri pari.

Cambiamo la riga 9 scrivendo:

`for i in range(5,1000,2):`

In questo modo cerchiamo tutti gli interi successivi a cinque, a passi di due, e possiamo evitare la metà dei numeri su cui effettuare il test.

Se applichiamo questa modifica il tempo di esecuzione scende a 1450 secondi! Come è possibile una differenza così piccola? In realtà abbiamo evitato di fare il test di primalità più immediato (al primo passaggio il numero è già divisibile per due) e non abbiamo tolto i test più onerosi (l'ultimo primo del nostro elenco deve essere diviso quasi un milione di volte per verificare che sia tale).

É dunqne proprio necessario fare tutte queste divisioni? In realtà no. Se infatti un numero non è primo lo si può scrivere come prodotto di due fattori $n = a \cdot b$. Il più piccolo di questi fattori non può essere inferiore a 2 (ad esempio $a = 2$) e l'altro non può essere superiore a $b = \frac{n}{2}$.

Quindi possiamo cambiare il test di primalità alla riga 5 effettuando il loop solo fino a n/2:

```
for k in range(2,n/2):
```

Con questa versione il tempo di esecuzione scende a 725 secondi: è un passo in avanti nella direzione giusta.

Questa modifica ha considerato la peggiore possibile delle ipotesi, ma se un numero è scomponibile in due fattori non è quello più grande che ci interessa, bensì quello più piccolo. Allora possiamo dire che il più piccolo dei due non può essere maggiore della radice quadrata del numero n: $n = \sqrt{n} \cdot \sqrt{n}$.

Con questa considerazione possiamo nuovamente modificare il test di primalità alla riga 5 e scrivere (dopo aver importato il modulo math):

```
for k in range(2,int(math.sqrt(n))+1):
```

Facciamo il test di divisibilità solo fino alla radice quadrata di n più uno. Con questa versione il tempo di esecuzione scende a 3,1 secondi: un miglioramento eccezionale.

2.3 Trovare i primi in un intervallo

Facciamo un'ultima considerazione. Qualsiasi numero intero è divisibile solo per i suoi fattori primi o combinazioni di questi. Questo significa anche che per sapere se un numero è primo posso limitarmi a dividerlo solo per i primi inferiori ad esso; gli altri eventuali divisori sono comunque multipli di questi primi. Per applicare questo algoritmo devo già avere, o costruire nel corso dell'elaborazione, una lista di primi. Non si tratta più di eseguire un semplice test di primalità tutti i numeri di un intervallo.

```
import math
primi=[2,3]
def test_primalita4(n):
    p=1 # suppongo che sia primo
    sup = int(math.sqrt(i))+1 # estremo superiore oltre il quale
                              # non cerchiamo divisori
    for k in primi:
        # per tutti i k contenuti tra i miei primi
        if (k>sup): # ma che non superino il sup
            break  # se k supera questo limite n è primo
        # altrimenti controllo se la divisione è possibile
        # cioé se il resto della divisione è zero
        if (n%k==0):
            p=0    # se avviene così il numero i non è primo
            break # ed esco dal loop
    return p
for i in range(5,1000000,2): # eseguio il test su 1000000 di interi
    p=test_primalita4(i)
    if p==1:
        primi.append(i)
```

```
print(len(primi))
```

Con questa ultima modifica il tempo di esecuzione scende a 0,9 secondi.

A questo punto abbiamo un algoritmo accettabile per creare una lista di numeri primi su cui compiere le nostre osservazioni e ricerche. Osserviamo che, se cerchiamo su internet, possiamo trovare piccoli programmi, anche in python, centinaia di volte più veloci di quello che abbiamo appena scritto. Possiamo usare uno di quelli, ma il nostro codice ha il vantaggio di essere molto più facilmente comprensibile in ogni sua parte.

2.4 La distribuzione dei numeri primi

Ora che possiamo creare la nostra raccolta di numeri primi possiamo studiare come sono distribuiti. Proviamo a rappresentare sul piano cartesiano i nostri numeri, tracciando per ognuno di essi un punto le cui coordinate sono rispettivamente il numero d'ordine del primo e il suo valore. Aggiungiamo al codice precedente una riga iniziale per importare il modulo **pylab** che useremo per fare i grafici.

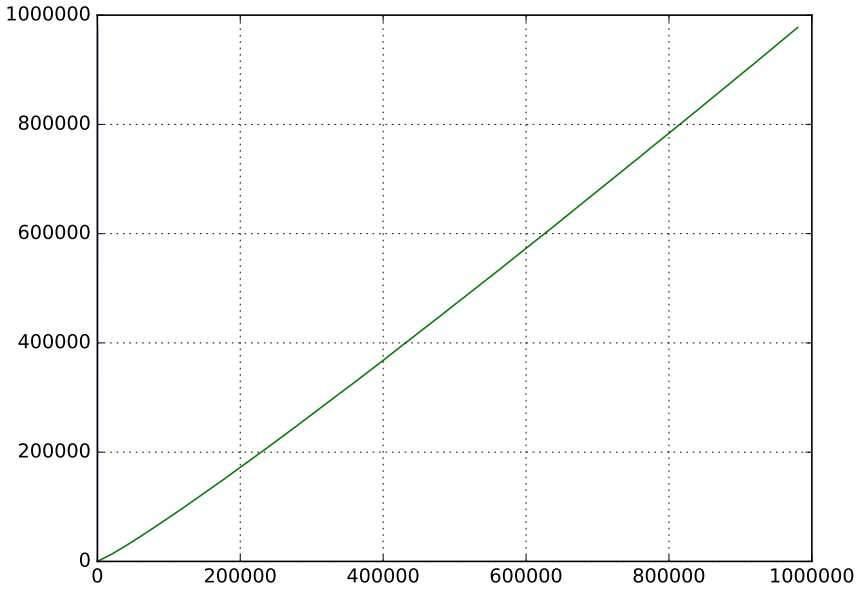
```
from pylab import *
```

La funzione **plot** visualizza un punto per tutti gli elementi della variabile passata (nel nostro caso la lista primi). Ogni punto ha per coordinata x il numero d'ordine della lista e per coordinata y il valore dell'elemento della lista.

Alla fine del file inseriamo invece le seguenti linee:

```
plot(primi,'.') # stampa un punto (.)
    # in corrispondenza di tutti i numeri
grid(True)      # visualizza una griglia nel grafico
show()          # accende il grafico
```

Otteniamo un grafico di questo tipo:



Questa linea ha proprio l'aspetto di una retta. Vogliamo confermare questa interpretazione con una valutazione quantitativa.

Ricordiamo che una retta è descrivibile da un equazione del tipo:

$$y = m \cdot x + c \tag{2.1}$$

dove m è il coefficiente angolare della retta (legato all'inclinazione della retta) e c è un parametro che ci da il punto di intersezione della retta con l'asse y.

Il metodo classico per affrontare il problema è quello che fa uso della "retta dei minimi quadrati". Con questo metodo possiamo trovare la retta che meglio approssima una serie di punti, con la possibilità di verificare quanto bene la retta trovata è in grado di passare vicino ai punti dati. Non abbiamo bisogno di conoscere tutti i dettagli tecnici del metodo: diversi moduli python hanno delle funzioni già pronte per questo scopo.

Importiamo il sottomodulo stats dal modulo scipy:

```
from scipy import stats
```

Usiamo la funzione linregress. In ingresso le possiamo dare o un array bidimensionale con le coordinate dei nostri punti, oppure due liste o array unidimensionali con le ascisse e ordinate. In output otteniamo cinque parametri. Nell'ordine abbiamo il parametro m e c della retta, un coefficiente detto di correlazione e altri due parametri. Il coefficiente di correlazione ha un valore compreso tra 1 e -1: più siamo prossimi a questi due estremi meglio la retta si adatta ai punti dati, se invece il valore tende a 0 i punti dati hanno poco a che fare con una retta.

Quindi inseriamo le seguenti due righe al termine del nostro programma:

```
m,c, r_value, p_value, std_err = stats.linregress(interi,primi)
print 'm=',m,'c=',c,'coef. corr.=',r_value
```

Troviamo i parametri della nostra retta dei minimi quadrati sul primo milione di interi: otteniamo:

m= 12.9169400329 c= -28609.262976 coef. corr.= 0.99960564776

Proviamo con dieci milioni:

m= 15.2259019732 c= -239318.163789 coef. corr.= 0.999722558251

Ed infine con 100 milioni:

m= 17.5354894929 c= -2053277.72057 coef. corr.= 0.999794706277

In nostri dati stanno con ottima approssimazione su una retta, tanto meglio quanto più aumentiamo il numero di primi. Come mai allora questo fatto non è riportato negli usuali libri di matematica?

Se osserviamo meglio i dati precedenti possiamo notare che è si vero che il coefficiente di correlazione tende sempre più a uno, ma il coefficiente angolare della retta è sempre più crescente. I dati stanno su una retta, ma la retta non è univocamente definita. Noi vorremmo invece una retta a cui tendano definitivamente (asintoticamente) tutti i numeri primi, ma questo non è possibile.

2.5 La distribuzione dei numeri primi (parte seconda)

Un altro aspetto interessante relativo alla distribuzione dei numeri primi è trovare una qualche regolarità nella distanza tra essi, in particolare nella differenza tra un primo e il suo precedente. Costruiamo un programma che ci permetta di calcolare queste distanze.

Partiamo da una lista con i nostri numeri primi (prendiamo uno dei codici precedenti). Costruiamo una nuova lista delle stesse dimensioni di quella dei numeri primi. Per ogni primo nella lista la distanza i-esima è la differenza tra il numero primo in posizione i-esima e il successivo (devo quindi scorrere la lista fino al penultimo elemento).

Possiamo aggiungere quindi le seguenti linee di codice a uno dei programmi precedenti:

```
distanza = [0]* len(primi)
k=0
for i in primi[:-1]:
    distanza[k]=primi[k+1]-primi[k]
    k=k+1
```

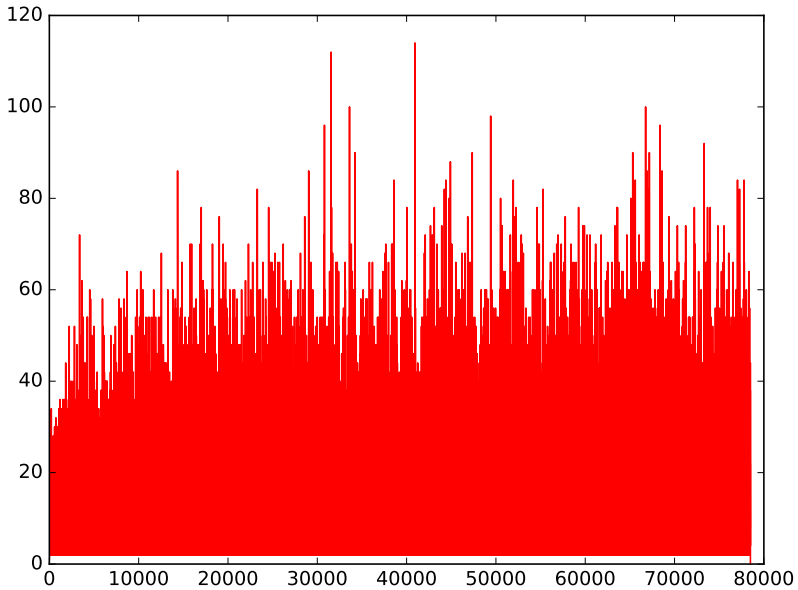
[1, 2, 2, 4, 2, 4, 2, 4, 6, 2, 6, 4, 2, 4, 6, 6, 2, 6, 4, 2, 6, 4, 6, 8, 4, 2, 4, 2, 4, 14, 4, 6, 2, 10, 2, 6, 6, 4, 6, 6, 2, 10, 2, 4, 2, 12, 12, 4, 2, 4, 6, 2, 10, ...

Se scorriamo la lista di numeri che viene fuori non notiamo alcuna particolarità o regolarità evidente, se non il fatto che molto spesso la distanza tra i primi è solamente due e ovviamente la distanza è sempre pari perché tutti i primi sono dispari.

Provare a rappresentare graficamente questa lista.

```
plt.plot(distanza, 'r')
plt.show()
```

Per il primo milione di interi otteniamo un grafico di questo tipo:



Neanche questo grafico sembra particolarmente significativo. Proviamo allora a fare un istogramma dei dati. Per fare un istogramma dobbiamo contare quante volte compare ogni

numero nella nostra lista chiamata distanza. Poi possiamo un grafico ponendo in ascissa il singolo numero, in ordine crescente, e in ordinata quante volte compare, cioè la sua frequenza assoluta. Se i numeri distinti della lista sono più di qualche centinaia possiamo fare un grafico a colonne per intervalli. Ad esempio nella prima colonna riportiamo quante volte compaiono numeri compresi tra uno e dieci, nella seconda quante volte quelli compresi tra undici e venti, e così via.

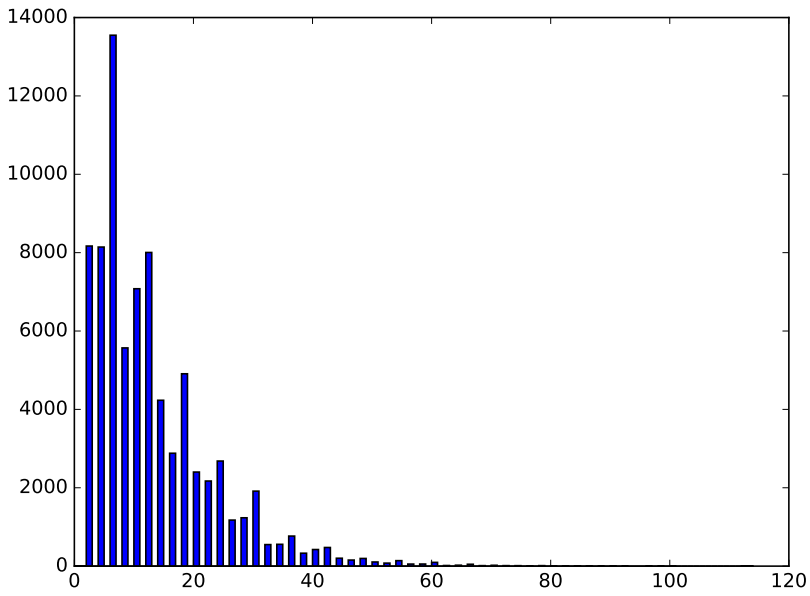
Nel nostro caso il numero più grande che compare nella lista distanza relativa al primo milione di interi vale solo 114 e non ci sono problemi.

```
print max(distanza)
```

Calcolare i valori da riportare nell'istogramma e disegnarlo è semplicissimo. Esiste infatti una funzione nella libreria matplotlib che svolge facilmente tutte le operazioni. Possiamo quindi aggiungere la seguente riga al nostro ultimo programma(il parametro bins specifica in quanti intervalli costruire il grafico):

```
plt.hist(distanza, bins=114)
```

Otteniamo questo grafico



3 Pi greco

3.1 Definizione

Il numero chiamato pi greco è noto fin dall'antichità. Questo nome tuttavia fu usato per la prima volta nel 1706 dal matematico inglese William Jones nel testo "A New Introduction to Mathematics". Questa costante numerica rappresenta il rapporto tra la lunghezza del perimetro e del diametro di una circonferenza.

Il problema iniziale è stato quello di avere un valore attendibile di tale numero. Nella Bibbia si scrive che esso vale semplicemente tre. Altri popoli approssimavano pi greco con una semplice frazione. Il primo che propose un metodo scientifico per trovare un valore quanto meglio approssimato fu Archimede. Egli approssimò una circonferenza con un poligono regolare inscritto e circoscritto alla stessa. Calcolando la lunghezza di questi poligoni trovò che:.

A partire dal seicento si svilupparono formule sempre più efficaci per determinare quante più cifre possibili. Infatti questo numero è irrazionale e non è rappresentabile con una frazione ovvero con un numero decimale periodico: le cifre dopo la virgola si succedono senza periodicità. Nell'ottocento si arrivò a calcolare a mano alcune centinaia di cifre.

In tempi moderni è il computer a farla da padrone. Nel 1949 L'Eniac impiegò alcuni giorni per calcolare due mila cifre. Per arrivare al milione dobbiamo andare agli anni settanta. A oggi siamo riusciti a calcolare circa cinquemilamiliardi di cifre.

Il nostro interesse in questa sede non è quello di implementare uno degli algoritmi per il calcolo di questo numero, ma uno studio e rappresentazione delle cifre. Il python ci viene incontro anche in questo caso con la libreria simpy. In essa ci viene data la possibilità di fare calcoli con precisione arbitraria. In particolare è presente anche una routine per il calcolo di pi greco con precisione voluta (compatibilmente con i tempi di calcolo).

3.2 Il calcolo di pi greco

I metodi per calcolare pi greco sono innumerevoli. Noi ne proponiamo solo uno, a titolo esemplificativo, compatibile con le conoscenze di uno studente delle superiori.

Dalla geometria elementare sappiamo che l'area di un cerchio è data dal prodotto di pi greco per il quadrato del raggio. Per cui possiamo anche definire pi greco come il rapporto tra quest'area e il quadrato del raggio del cerchio.

$$A = \pi \cdot r^2 \quad ; \qquad \pi = \frac{A}{r^2} \tag{3.1}$$

Cerchiamo un modo di calcolare l'area di un cerchio senza usare pi greco. Prendiamo una circonferenza di raggio unitario centrata nell'origine degli assi cartesiani. Affettiamo questa circonferenza approssimandola a dei rettangoli verticali alti quanto la circonferenza stessa. Il diametro di questa circonferenza ha lunghezza 2: lo dividiamo in n parti di lunghezza dx che è anche la misura della base di ognuno dei nostri rettangolini. L'altezza dei questi rettangoli, per il teorema di Pitagora, è il doppio della radice quadrata della differenza tra il quadrato dell'ipotenusa (semplicemente uno) e il quadrato del cateto alla base (x^2).

Se sommiamo tutti questi rettangoli otteniamo una buona approssimazione dell'area del cerchio (ogni rettangolino trascura una piccola area colorata in blu nella figura).

Possiamo scrivere quindi:

```
n=1000    # numero di rettangoli
ax=0      # area iniziale
dx=2.0/n # larghezza dei rettangoli
for i in xrange(n):
    x=dx*i-1
    ax = ax + dx*2*((1-x*x)**0.5)
print 'approssimato',ax
```

Confrontiamo il risultato ottenuto con un valore più corretto di pi greco. Questo lo possiamo ottenere semplicissimamente con una funzione apposita della libreria sympy. In questa libreria possiamo calcolare un'espressione, attraverso la funzione N(), con un numero di cifre significative arbitrariamente grande. In particolare possiamo valutare con precisione voluta il numero pi greco e il numero "e" di Eulero.

```
from sympy import *
a = N(pi, 20)
print 'pi greco =',a
```

Possiamo allora osservare che il nostro metodo iniziale con n=1000 ci dà solo cinque cifre significative corrette e la precisione cresce molto lentamente all'aumentare di n. Per scopi pratici è quasi inutile. D'altra parte con la funzione python possiamo ottenere un milione di cifre in circa un minuto.

3.3 La distribuzione delle cifre in pi greco

Proviamo a vedere se nelle cifre di pi greco esiste qualche tipo di regolarità. Dobbiamo quindi escogitare un metodo per estrarre queste cifre (decimali) dal nostro numero. Questa cosa non è immediata a causa del tipo di rappresentazione dei numeri utilizzata dal python e dal modulo sympy. Innanzi tutto la variabile a dell'ultimo codice è considerata di tipo float e non possiamo richiamare le sue singole cifre. Possiamo però trasformare il numero in una stringa e da questa estrarre le singole lettere, nel nostro caso le cifre di pi greco. Poi possiamo trasformare queste singole lettere di numero (sono infatti lettere di una stringa e non possono essere elaborate come numeri) in numeri interi che assegniamo ai singoli elementi di una opportuna lista. Notiamo infine che la seconda lettera di questa stringa è la virgola decimale e quindi non va presa.

Scriviamo quindi in seguente codice:

```
from sympy import *
a = N(pi, 2000001)
b=repr(a) # trasformiamo il numero in una stringa
data = [0]*(len(b)-1) # creiamo una matrice delle stesse dimensioni
data[0]=int(b[0])     # attribuiamole il primo elemento (k=0)
k=2                   # e saltiamo il secondo elemento (k=1)
for i in b[2:]:
    data[k-1]=int(b[k])
    k=k+1
# rappresentiamo l'istogramma dei dati ottenuti
import matplotlib.pyplot as plt
k=plt.hist(data, bins=10)
print (k[0]) # visualizziamo le frequenze assolute
plt.show()
```

Queste sono le frequenze ottenute con due milione di decimali:
[199792. 199535. 200077. 200142. 200083. 200521. 199403. 200310. 199447. 200691.]

Possiamo notare che le cifre compaiono tutte con una frequenza molto simile. Questo è un risultato inaspettato? No, tant'è che i numeri che hanno questo comportamento in una certa base (nel nostro caso 10) sono chiamati *semplicemente normali*. In particolare si presume che pi greco sia *normale* in qualsiasi base. Ma a tutt'oggi nessuno è riuscito a dimostrarlo.

4 E di Eulero

4.1 Definizione

Il numero di Eulero, indicato con la lettera **e**, è stato inizialmente usato nelle tavole logaritmiche pubblicate nel 1618 da John Napier. La prima definizione tuttavia si deve a Jakob Bernoulli:

$$e = \lim_{n \to \infty} \left(1 + \frac{1}{n}\right)^n \tag{4.1}$$

Il numero **e** è un numero irrazionale, quindi ha infinite cifre decimali senza un periodo che si ripeta. Calcolare queste cifre applicando la definizione appena vista non è facile. Proviamoci.

Per calcolare il limite dobbiamo tabulare il valore della funzione per valori di n sempre più grandi.

```
for j in xrange(1,20):
    i=10**j
    print (1.0+1.0/i)**i
```

Questo numero converge con difficoltà, a causa delle eccessive approssimazioni. Addirittura, con numeri n superiori a venti il calcolo perde di significato.

Il valore corretto, entro venti cifre decimali, ci può essere dato dalla libreria sympy:

```
from sympy import *
a = N(E,22)
print a
```

($e = 2,718281828459045235360$)

Una definizione più utile ai fini pratici è data dal seguente sviluppo in serie:

$$e = \sum_{i=1}^{\infty} \frac{1}{n!} \tag{4.2}$$

Questa è effettivamente la formula di riferimento usata dalle calcolatrici per calcolare l'esponenziale di un numero.

```
import math
e=0
for j in xrange(20):
    e=e+1.0/math.factorial(j)
    print '{:02.15}'.format(e)
```

Questo codice converge a venti cifre decimali in circa venti iterazioni.

4.2 La distribuzione delle cifre in "e"

Proviamo a vedere se nelle cifre di **e**, analogamente a quanto visto per pi greco ,esiste qualche tipo di regolarità.

Scriviamo quindi in seguente codice:

```
from sympy import *
a = N(E, 20001)
b=repr(a) # trasformiamo il numero in una stringa
data = [0]*(len(b)-1) # creiamo una matrice delle stesse dimensioni
data[0]=int(b[0])     # attribuiamole il primo elemento (k=0)
k=2                   # e saltiamo il secondo elemento (k=1)
for i in b[2:]:
    data[k-1]=int(b[k])
    k=k+1
# rappresentiamo l'istogramma dei dati ottenuti
import matplotlib.pyplot as plt
k=plt.hist(data, bins=10)
print (k[0]) # visualizziamo le frequenze assolute
plt.show()
```

Queste sono le frequenze ottenute con due milione di decimali:

[199093. 200171. 199472. 200360. 199923. 200285. 200395. 199789. 200098. 200414.]

Il comportamento è del tutto analogo a quello mostrato da pi greco: le cifre compaiono tutte con una frequenza molto simile. Siamo di fronte ad un numero normale. Ma a tutt'oggi nessuno è riuscito a dimostrarlo.

Indice analitico