Introduzione alla programmazione

Iniziamo questo mese a parlare della programmazione in Mathematica. La varietà degli stili di programmazione possibili, se da un lato costituisce una fonte di indubbia ricchezza che permette all'esperto di trovare soluzioni eleganti e/o efficienti ai suoi problemi, d'altra parte mette pesantemente in imbarazzo il neofita che resta interdetto dalle molte differenti possibilità di soluzione dello stesso problema. La trattazione che segue, essenzialmente basata su esempi, non ha nessuna pretesa di completezza e tratta solo gli aspetti più immediati della programmazione in Mathematica.

di Francesco Romani

Introduzione

I linguaggi di programmazione degli ultimi quaranta anni formano una "fauna" che per varietà e sviluppi evolutivi non ha niente da invidiare a qualche affollata specie animale. I momenti più significativi di questa evoluzione sono alcuni linguaggi (BASIC, Cobol, Fortran, Algol 60 e poi LISP, Pascal e PROLOG) che, per le idee innovative alla base della loro progettazione, hanno costituito delle tappe essenziali nello sviluppo dell'informatica. Molti altri linguaggi sono stati progettati per soddisfare esigenze specifiche e alcuni di essi hanno grandissima diffusione (per esempio il C e le ultime versioni del Fortran). In *Mathematica*, con una scelta sicuramente non da tutti condivisa, si è deciso di implementare un linguaggio di programmazione che riunisce le caratteristiche di molti dei linguaggi più usati. è quindi possibile programmare in stile C, Pascal, LISP o mediante regole di riscrittura basate sul *pattern-matching*. Ogni problema presenta svariate strade di risoluzione e non sempre le più eleganti sono le più efficienti. Per di più esiste un sistema di abbreviazioni che permettono di scrivere programmi più compatti ma meno leggibili di quelli che usano le forme estese. Nel nostro percorso alla scoperta di *Mathematica* cercheremo di partire dagli esempi più semplici e immediati, dando quando possibile un'idea delle soluzioni più involute. Nel riquadro della bibliografia sono citate le pubblicazioni specialistiche dove il pubblico seriamente interessato può trovare risposta a molti dei suoi problemi.

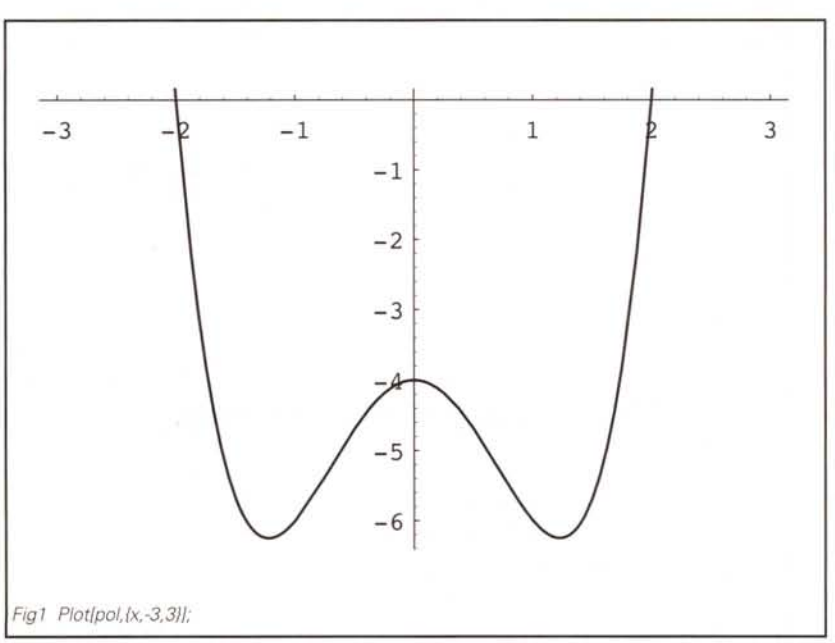


Fig1 Plot[pol,{x,-3,3}];

Costruzione di un programma a partire da una seduta interattiva

Mathematica è un linguaggio interpretato e interattivo. L'utente può fare a meno di programmare, utilizzando il sistema come una grossa calcolatrice da tavolo, ma l'utilizzazione più naturale consiste nel risolvere un esempio in modo interattivo, passo dopo passo, e poi mettere insieme i vari passi in un programma che può essere utilizzato per risolvere i problemi più generali. Nel libro di Maeder intitolato "Programming in *Mathematica*." molti capitoli sono dedicati alla scrittura del package di grafica per le funzioni complesse ComplexMap di cui abbiamo presentato due esempi di funzionamento nel precedente articolo. L'autore inizia da una sessione interattiva per disegnare una mappa e arriva fino alla stesura di un completo Package, con il trattamento di tutte (o quasi) le possibili situazioni, da includere in una libreria di programmi. I nostri obiettivi sono meno ambiziosi (e lo spazio a disposizione più ridotto); vediamo di mettere insieme un programma

che traccia nel piano complesso le posizioni delle n radici di un polinomio di grado n.

Innanzi tutto definiamo un polinomio con le 4 radici 2, -2, i, -i, lo plottiamo e ne calcoliamo le radici con la funzione **NSolve**. Si noti il simbolo "==" che denota il predicato di uguaglianza (l'equivalente Fortran è ".EQ." e l'equivalente Pascal è "=").

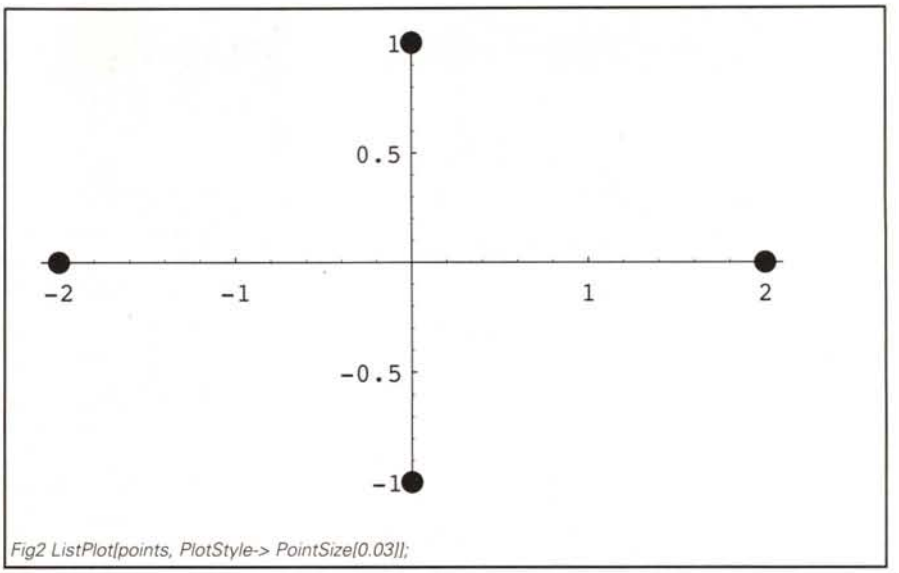


Fig2 ListPlot[points, PlotStyle-> PointSize[0.03]];

```
In[1]:=
    pol=Expand[(x^2-4) (x^2+1)]

Out[1]=
    -4 - 3 x^2 + x^4

In[2]:=
    NSolve[pol==0,x]

Out[2]=
      {{x -> -2.}, {x -> 7.88861 10^-31- 1. I},

       {x -> 7.88861 10^-31 + 1. I}, {x -> 2.}}
```

Il risultato viene reso sotto forme di regole di sostituzione e presenta dei fattori spuri dovuti alla aritmetica finita usata per i calcoli. La funzione **Chop** taglia queste imprecisioni e l' operatore "**/.**" permette di sostituire una sottoespressione in una espressione. Nel nostro caso l'espressione da sostituire è la **x** e siccome abbiamo una lista di regole si ottiene una lista di espressioni sostituite, ovvero il vettore delle soluzioni cercate.

```
In[3]:=
    rules=Chop[%]
Out[3]=
    {{x -> -2.}, {x -> -1. I}, {x -> 1. I},

     {x -> 2.}}
In[4]:=
    sol = x /. rules
Out[4]=
    {-2., -1. I, 1. I, 2.}
```

Gli operatori **Re** e **Im** estraggono rispettivamente la parte reale e quella immaginaria che possono venire unite insieme per formare una lista di coppie, ovvero le coppie ascissa-ordinata dei punti da disegnare nel piano complesso.

```
In[5]:=
    realpart = Re[sol]
Out[5]=
    {-2., 0, 0, 2.}
In[6]:=
    impart = Im[sol]
Out[6]=
    {0, -1., 1., 0}
In[7]:=
```

Definizioni in *Mathematica*

Esistono due modi principali per definire una variabile:

a = <espressione>

significa che tutte le volte che il *Kernel* deve valutare a viene sostituita <espressione> come è stata valutata al momento dell'assegnamento. ("**=**" è la abbreviazione della funzione **Set**, la forma interna di "**x=a**" è "**Set[x,a]**").

a := <espressione>

significa che tutte le volte che il *Kernel* deve valutare a viene valutata e sostituita <espressione>. ("**:=**" è la abbreviazione della funzione **SetDelayed**, la forma interna di "**x:=a**" è "**SetDelayed[x,a]**").

```
In[1]:=
Clear[x]
a  =  x;
b :=  x;
x  = 10;
In[2]:=
a
Out[2]=
x
In[3]:=
b
Out[3]=
10
```

```
points = Transpose[{realpart,impart}]
```
Out[7]=
```
{{-2., 0}, {0, -1.}, {0, 1.}, {2., 0}}
```

A questo punto basta usare la **ListPlot** e il gioco è fatto.

Una volta sperimentato istruzione per istruzione cosa si voleva fare, viene il desiderio di raccogliere quel gruppo di istruzioni in un "programma" che possa essere richiamato quando lo si desidera. Il modo più semplice è quello di raccogliere le istruzioni (separate dal simbolo "**;**") in un unica cella del *Notebook*. L'intera cella può essere mandata in esecuzione con un solo comando (la pressione del tasto ENTER).

```
solve     = NSolve[pol==0,x];
rules     =  Chop[solve];
sol       = x /. rules;
realpart  = Re[sol];
impart    = Im[sol];
list      = Transpose[{realpart,impart}];
ListPlot[list,
         PlotStyle-> PointSize[0.03]]);
```

Questo approccio è l'equivalente di un programma BASIC che viene fatto eseguire con il comando RUN.

Il passo successivo consiste nel raccogliere le istruzioni tra una coppia di parentesi tonde e assegnare la sequenza ad una variabile (ad esempio **RootPol1**) con il simbolo "**:=**" (**SetDelayed**,). Tanto per compattare il programma si è anche provveduto ad eliminare qualche variabile temporanea. Ogni volta che il *Kernel* dovrà valutare **RootPol1** eseguirà la sequenza di istruzioni tra parentesi.

```
RootPol1:=
(solve    = Chop[NSolve[pol==0,x]];
sol       = x /. rules;
realpart  = Re[sol];
impart    = Im[sol];
ListPlot[Transpose[{realpart,impart}],
         PlotStyle-> PointSize[0.03]]);
```

Questo approccio è l'equivalente di una procedura Pascal senza argomenti e senza dichiarazioni di variabili, infatti, il polinomio da trattare si deve chiamare "**pol**" deve essere nella variabile "**x**" e deve essere definito nell'ambiente globale e tutte le variabili che vengono usate esistono e sono modificate nell'ambiente globale.

Una soluzione più pulita consiste nell'incapsulare la sequen-

Pattern matching

I *pattern* sono un modo per rappresentare classi di espressioni. La tecnica del *pattern-matching* permette di effettuare trasformazioni selettive ed è uno dei mezzi con cui si effettuano le elaborazioni simboliche. Ecco (dal manuale di S.Wolfram) alcuni esempi di *pattern* con il loro significato:

Pattern	*Significato*	*Esempio*
f[n_]	f con un qualunque argomento	**f[23]**
f[m_, n_]	f con due qualunque argomenti	**f[2,a]**
x^n_	x elevato ad un qualunque esponente	**x^(2/3)**
x_^n_	un espressione elevata ad un esponente	**(a+b)^(c+d)**
a_+b_+c_	una somma di tre espressioni	**2+5+(t^e)**

Quando una definizione di funzione contiene un *pattern* essa viene attivata solo se la chiamata soddisfa il *pattern*, se più di un *pattern* può venire soddisfatto viene usato il meno generale (l'ordine delle scelte è un punto delicato che influenza pesantemente la semantica dei programmi di *Mathematica* ma il discorso si farebbe troppo lungo ...).

Definiamo una funzione f con tre diversi tipi di argomenti (in ogni caso viene resa un stringa che ci dirà quale definizione è stata sostituita). Se f ha 2 argomenti viene usata la definizione 1. Se f ha l'argomento speciale y viene usata la definizione 3. Se f ha un qualunque argomento viene usata la definizione 2.

In[1]:=
```
f[a_,b_]:= "uso la prima definizione"
f[a_]:= "uso la seconda definizione"
f[y]:= "uso la terza definizione"
```
In[2]:=
```
f[x,y]
```
Out[2]=
```
uso la prima definizione
```
In[3]:=
```
f[y]
```
Out[3]=
```
uso la terza definizione
```
In[4]:=
```
f[x]
```
Out[4]=
```
uso la seconda definizione
```

Nel seguito vedremo numerosi esempi più raffinati di uso dei *pattern*

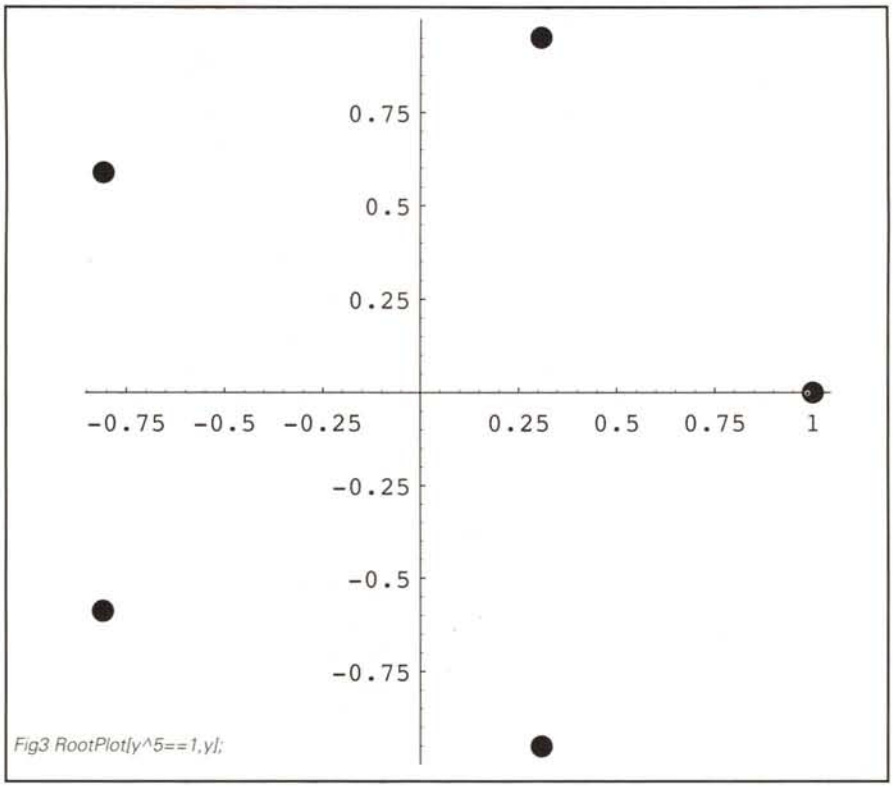


Fig3 RootPlot[y^5==1,y];

za in un modulo che permette di definire delle variabili locali senza interferire con l'ambiente esterno e di passare al programma degli argomenti (come nella maggioranza dei linguaggi che si rispettino). Questo si ottiene col costrutto

Module[{<variabili locali>}, <corpo>**]**

Con un ulteriore compattamento il nostro programma diviene:

```
RootPlot[pol_, x_] := Module[{sol},
  sol = x /. Chop[NSolve[pol==0,x]];
  ListPlot[Transpose[{Re[sol],Im[sol]}],
           PlotStyle-> PointSize[0.03]]]
```

Esiste una sola variabile locale "sol" e due argomenti: il nome del polinomio e la variabile del medesimo. Ogni volta che il *Kernel* deve valutare una espressione del tipo **RootPlot[**<espressione 1>, <espressione 2>**]** viene chiamato il nostro modulo con le sostituzioni:

{pol -> <espressione 1>, **x ->** <espressione 2>**}**

Chiamando **RootPlot** con "**pol**" e "**x**" si ottiene ancora il grafico di figura 2. Chiamando **RootPlot** con il polinomio "y^5-1" e la variabile **y** si ottengono le 5 soluzioni della equazione "$y^5 = 1$" (ovvero le 5 radici quinte dell'unità che stanno ai vertici di un pentagono regolare nel piano complesso).

Costrutti di programmazione

Certamente non basta mettere una dopo l'altra le istruzioni per fare un programma, è presto sentita l'esigenza dei classici costrutti che possa legare le azioni da compiere. Accenniamo qui ai più comuni, che in base ai principi della programmazione strutturata sono sufficienti a scrivere qualunque programma.

If[<test>, <e1>, <e2>, <e3>**]**

se <test> è vero viene valutata <e1>, se <test> è falso viene valutata <e2>, se <test> è indefinito viene valutata <e3> (per esempio, l'espressione "**a<4**", se "a" non ha un valore, non è ne falsa ne vera ma indefinita).

Do[<corpo>, <iteratore>**]**

<corpo> viene valutata in accordo alla iteratore, si noti l'analogia tra le espressioni: "**Table[i, {i,1,5}]**", che rende la lista dei numeri da 1 a 5, e "**Do[Print[i], {i,1,5}]**", che stampa gli stessi valori.

While[<test>, <corpo>**]**

Finché <test> è vero viene valutato <corpo>.

Esempio: il calcolo del fattoriale

Per vedere alcuni esempi dell'uso dei costrutti, e dare un idea della infinite possibilità con cui uno stesso problema può essere risolta, vediamo alcuni modi alternativi di definire la funzione fattoriale: $n! = n \times (n-1) \times (n-2) \times \ldots \times 2$.

1) Iterativo, con **Do**

```
facl[n_]:=Module[{m},
                 m=1;
                 Do[m=i m,{i,n}];
                 m];
```

2) Iterativo, con **While**

Dove trovare di più

T.W. Gray and J. Glynn. Exploring Mathematics with *Mathematica*. Addison Wesley,1991.
R. Maeder. Programming in *Mathematica*. Addison Wesley,1991 (II Edition).
The *Mathematica* Journal. Rivista trimestrale edita dalla Miller Freeman Inc.
S. Wolfram. *Mathematica*. A System for Doing Mathematics by Computer. Addison Wesley,1991 (II Edition).

```
fac2[n_]:=Module[{m},
                 m=1;
                 While[i<=n, m=i m; i=i+1];
                 m];
```

3) Ricorsivo, con **If**

```
fac3[n_]:=If[n==1, 1, n Fac4[n-1]];
```

4) Ricorsivo, basato sul *pattern-matching*

```
fac4[1]:=1;
fac4[n_]:=n Fac4[n-1];
```

5) Analitico, basato sulla proprietà che lega il fattoriale alla funzione speciale Γ(x): n! = Γ(n+1).

```
fac5[n_]:= Gamma[n+1]
```

Esempio: la torre di Hanoi

Sperimentiamo la potenza di programmazione di *Mathematica* risolvendo il problema della Torre di Hanoi. Si tratta di un noto rompicapo che consiste in tre pioli verticali e un numero qualsiasi di ciambelle di dimensione decrescente. L'obiettivo del gioco è spostare tutte le ciambelle dal primo al secondo piolo senza che mai una ciambella stia sotto una più grande. Il terzo piolo è usato come ausilio per i trasferimenti.

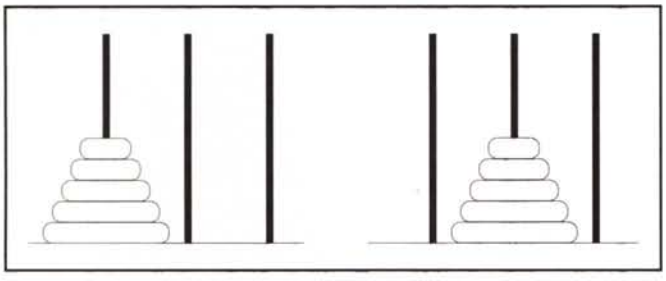
Si vede facilmente che l'unico modo per poter spostare la ciambella più grande dal primo al secondo piolo consiste nel porre prima tutte le altre ciambelle sul terzo piolo; in base alla regola enunciata precedentemente, queste devono essere ordinate in ordine crescente. Qualunque algoritmo di risoluzione del problema con n ciambelle deve, quindi, potersi suddividere nei seguenti tre passi:

a) sposta le prime n -1 ciambelle dal piolo 1 al piolo 3, rispettando l'ordine;

b) sposta l'ultima ciambella dal piolo 1 al piolo 2;

c) sposta le prime n -1 ciambelle dal piolo 3 al piolo 2, rispettando l'ordine.

I punti (a) e (c) implicano la risoluzione del problema di dimensione n -1 con una permutazione dell'ordine dei pioli. Questa osservazione permette una facile risoluzione ricorsiva del problema:



In[1]:=

```
Hanoi[n_,s_,d_,a_]:=(Hanoi[n-1,s,a,d];
                     Print[" muovi da ",s,"
a ",d];
                      Hanoi[n-1,a,d,s];) /;
n>0;
```

Il simbolo "**/;**" significa che la definizione va applicata solo se ciò che segue è vero. Quindi **Hanoi** è indefinita per n≤0 e la sua chiamata equivale a nessuna operazione. Il lettore esperto noterà la maggiore compattezza di questa implementazione rispetto al analoghe versioni ricorsive in C o Pascal. Per di più i nomi dati ai pioli sono trattati in modo simbolico e sono quindi arbitrari.

In[2]:=

```
Hanoi[3, bianco, rosso, verde]

 muovi da bianco a rosso
 muovi da bianco a verde
 muovi da rosso a verde
 muovi da bianco a rosso
 muovi da verde a bianco
 muovi da verde a rosso
 muovi da bianco a rosso
```

In[3]:=

```
Hanoi[4, 1,2,3]

 muovi da 1 a 3
 muovi da 1 a 2
 muovi da 3 a 2
 muovi da 1 a 3
 muovi da 2 a 1
 muovi da 2 a 3
 muovi da 1 a 3
 muovi da 1 a 2
 muovi da 3 a 2
 muovi da 3 a 1
 muovi da 2 a 1
 muovi da 3 a 2
 muovi da 1 a 3
 muovi da 1 a 2
 muovi da 3 a 2
```

MC

Francesco Romani è raggiungibile tramite Internet all'indirizzo romani@di.unipi.it