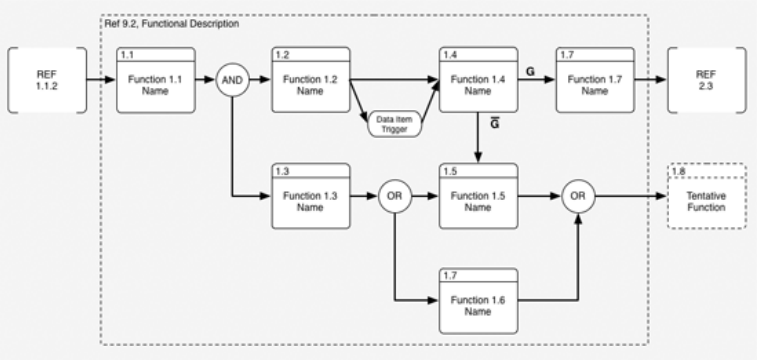
LABORATORIO MATEMATICA

DIPARTIMENTO DI MATEMATICA



ITIS V.Volterra
San Donà di Piave

Versione [09/2008.1][S-All]

BY NC ND

Parte I

NUMERI

1

INTERI

Il loro nome testimonia quanto essi siano stimati,
nei regni del pensiero puro e dell'estetica,
rispetto ai loro fratelli minori, cioè i numeri complessi e i numeri reali -
dai cui nomi virtualmente trasuda
il loro sciocco venire a patti con la complessa realtà della vita di tutti i giorni!

Manfred R. Schroeder *La teoria dei Numeri*

2 REALI

Parte II

FUNZIONI

Parte III

MODELLI

Parte IV

PROGRAMMAZIONE

3 IL PARADIGMA DI PROGRAMMAZIONE FUNZIONALE

3.1 PREMESSA

Il *Paradigma di Programmazione Funzionale*[1] si affianca ad altri, forse più noti, paradigmi di programmazione come, ad esempio, quelli Logico, Imperativo, Dichiarativo, orientato ad Oggetti.

Il significato del termine *Paradigma* (dal greco paràdeigma) è modello, progetto, esempio, prototipo, mentre il termine *Funzionale* associato a *Programmazione* deriva dall'applicazione del concetto matematico di funzione; pertanto si può dire che il *Paradigma di Programmazione Funzionale* rappresenta uno dei possibili approcci nell'analisi di un problema e nella ricerca del suo algoritmo risolutivo, il quale sarà espresso in una forma compatibile con il linguaggio matematico.

Quindi, il programmatore che implementa la programmazione funzionale non si occupa di costruire programmi, bensì di applicare funzioni, siano esse predefinite (come ad esempio le funzioni matematiche *sqrt*, *sin*, *abs* presenti in tutti gli ambienti di programmazione) o di nuova e originale definizione.

Tutte le caratteristiche del Paradigma Funzionale ruotano attorno ai concetti di *espressione* (aspetto sintattico, cioè relativo alle regole per la costruzione delle frasi del linguaggio funzionale) e di *valutazione di una espressione* (aspetto semantico, cioè relativo all'attribuzione del significato delle frasi sintatticamente corrette).

Ad esempio, $3 + \texttt{sqrt}(4)$ è un'espressione scritta seguendo ben precise regole algerbriche e 5 è il risultato della valutazione di tale espressione, determinato sulla base del significato attribuito, in particolare, all'operatore $+$ e alla funzione `sqrt`.

Da ciò risulta ancor più evidente come la *Programmazione Funzionale* confonda le sue origini con quelle della Matematica e come questo stile di programmazione si presti meglio di ogni altro ad essere utilizzato nella risoluzione di problemi matematici.

Quindi, lo studio di tale paradigma parte da forti e concreti agganci con la tradizione matematica e si prefigge di sviluppare nei programmatori che lo attuano una mentalità matematica, contrapponendosi invece ad un uso utilitaristico degli strumenti informatici messi a disposizione dagli ambienti di programmazione, ad esempio per la costruzione di grafici e per *fare conti*.

La scelta di applicare il *Paradigma Funzionale* comporta la completa rinuncia a tre concetti caratteristici della Programmazione Imperativa: ciclo, assegnazione di variabili e stato della memoria, comando. Queste rinunce non creano in genere grande disagio e anzi spesso l'eleganza e il livello di astrazione consentiti dalla programmazione funzionale affascinano chi vi si addentra.

E' necessario sottolineare che alcuni ambienti di programmazione consentono l'interazione tra diversi paradigmi di programmazione, ad esempio permettendo l'uso di cicli nella definizione di funzioni; ciò può fornire uno strumento molto potente, ma apprezzabile più da parte degli esperti programmatori; contrariamente, la contaminazione tra diversi paradigmi può creare confusione per i neofiti.

Negli esempi che seguiranno si farà riferimento all'ambiente per la *Programmazione Funzionale* Maxima, un Computer Algebra System (CAS) open-source in grado di eseguire calcoli numerici e simbolici, di costruire grafici di funzioni e che consente, con una certa chiarezza, di delimitare l'applicazione del paradigma di programmazione funzionale puro, pur prevedendo di poterne utilizzare anche altri.

1 Un particolare ringraziamento al Prof. Giuseppe Callegarin: i suoi testi sono stati fondamentali nella stesura di questo capitolo.

3.2 PROGRAMMAZIONE FUNZIONALE E MATEMATICA

3.2.1 *Concetto primitivo di funzione*

Una funzione è una regola di corrispondenza che associa a ogni elemento del suo dominio un unico elemento nel codominio. In generale:

$$\begin{aligned} f: \quad & D \longrightarrow C \\ & x \longmapsto y = f(x) \end{aligned}$$

Esempio 3.2.1.

$$\begin{aligned} f_k: \quad & \mathbb{Z} \longrightarrow \mathbb{Z} \\ & a \longmapsto a + k \quad \text{con } k \in \mathbb{N} \end{aligned}$$

Quindi in questa definizione matematica si possono individuare:

- il nome della funzione: f_k;
- il dominio $\mathbb{Z}$: l'insieme sul quale opera la funzione;
- il codominio $\mathbb{Z}$: l'insieme nel quale si troveranno le immagini, cioè i risultati ottenuti dall'applicazione della funzione;
- la legge (o regola): *a+k*, essa permette di associare alla variabile $a \in \mathbb{Z}$, al variare del parametro $k \in \mathbb{N}$, un valore nell'insieme $\mathbb{Z}$.

3.2.2 *Funzioni, Variabili e Parametri*

Nel *Paradigma Funzionale* le variabili ed i parametri sui quali si appoggiano le definizioni delle funzioni hanno lo stesso identico significato che viene loro attribuito nella Matematica, ma non è necessario esprimere di quale tipo essi debbano essere (operazione che comunemente viene denominata *tipizzare le variabili*), come ad esempio deve avvenire nella Programmazione Imperativa; pertanto il compito del programmatore che debba definire una funzione è semplificato.

Ad esempio, la funzione matematica f_k dell'esempio 3.2.1 potrà essere definita in un ambiente di *Programmazione Funzionale* come Maxima nel seguente modo:

$$f(a, k) := a + k$$

dove a e k si chiamano *parametri formali* della funzione e $a + k$ è l'espressione che la definisce; inoltre, l'assegnazione del risultato avviene mediante la valutazione dell'espressione $a + k$ in corrispondenza dei valori individuati per i parametri formali, tale assegnazione viene evidenziata dall'uso del simbolo di $:=$.

Si può osservare come nella definizione della funzione Maxima f non sia stato necessario specificare in quale insieme si debbano collocare sia i parametri formali a e k che il risultato della funzione, cioè, come già detto, senza bisogno di effettuare alcuna loro tipizzazione.

Ovviamente, una funzione viene *definita* per poi essere *applicata* ad un elemento del dominio in modo da restituire il risultato, si dirà più semplicemente che essa viene *valutata (o semplificata)*.

Negli ambienti come Maxima, è proprio nell'azione di *valutazione* di una funzione che avviene la sostituzione dei *parametri formali* con i cosiddetti *valori attuali*, cioè valori scelti nel dominio della funzione, ai quali sarà associato il risultato della funzione; sarà quindi in tale passaggio di assegnazione dei *valori attuali* ai *parametri formali* che si realizza implicitamente la tipizzazione delle variabili sulle quali la funzione opera.

In riferimento alla funzione dell'esempio 3.2.1:

- il significato della scrittura matematica $f_2(3) = 5$ è ovvio;
- nella *Programmazione Funzionale* si possono individuare due azioni:

  Azione 1: *chiamata della funzione* con assegnazione del valore attuale 3 al parametro formale a e del valore attuale 2 al parametro formale k:

  $$f(3,2)$$

  Azione 2: *valutazione (o semplificazione) della funzione* con restituzione del risultato:

  $$5$$

3.3 TRE CONCETTI CHIAVE

Oltre al concetto di *funzione*, che include anche quelli già visti di *definizione*, *chiamata* con assegnazione dei valori ai parametri e *semplificazione*, si possono identificare altri due concetti chiave propri del paradigma funzionale:

- l'*espressione condizionale*;
- la *ricorsione*.

L'*espressione condizionale*, o costrutto condizionale, si può così schematizzare:

```
if <condizione> then <espressione1> else <espressione2>
```

e si legge:

```
se la <condizione> è vera valuta <espressione1>, altrimenti <espressione2>
```

L'applicazione dell'*espressione condizionale* non si differenzia quindi sostanzialmente da quella propria della Programmazione Imperativa, se non per il fatto che nell'ambiente Maxima non è ammessa la forma ridotta:

```
if <condizione> then <espressione>
```

Per quanto riguarda il concetto di *ricorsione*, la struttura tipica di una funzione ricorsiva è la seguente:

```
nomefunz(parametri):= if <condizione di stop>
                         then <espressione base>
                         else <espressione ricorsiva>
```

dove

```
 <espressione ricorsiva>
```

contiene certamente la chiamata alla funzione stessa (ricorsiva):

```
nomefunz(nuovi valori ai parametri)
```

Esempio 3.3.1. *Definizione ricorsiva di potenza* $a^0 = 1,\ a^n = a \cdot a^{n-1},\ a \in \mathbb{R},\ n \in \mathbb{R}$*:*

```
pot(a,n):= if n=0 then 1
                  else a*pot(a,n-1)
```

Si intuisce quindi facilmente che mediante l'approccio ricorsivo è possibile realizzare un ciclo:

```
ciclo(i,n):= if i>n
                then "fine ciclo"
                else ciclo(i+1,n)
```

La chiamata a tale funzione potrebbe essere `ciclo(1,10)` e la frase *fine ciclo* verrà visualizzata dopo 10 passaggi ricorsivi.

Qualcuno obietterà che se esiste questa forma di corrispondenza rispettivamente tra ciclo e ricorsione e tra assegnazione e passaggio di parametri, non si vede quali vantaggi sostanziali fornisca la *Programmazione Funzionale*.

La semplicità dell'esempio non deve però trarre in inganno: si potrebbero citare degli esempi di funzioni non facilmente traducibili nel Paradigma Imperativo.

Tuttavia, la principale risposta a questa obiezione consiste nel fatto che nella programmazione funzionale pura vale il principio di *Trasparenza Referenziale* e la conseguenza più importante di questo principio è che ogni variabile ha sempre lo stesso valore nel suo campo di validità; questa proprietà rende i programmi funzionali più leggibili e facilmente verificabili.

In termini più semplici potremmo affermare che la Programmazione Funzionale ha un potere espressivo e un livello di verificabilità superiori a quelli consentiti, ad esempio, dalla Programmazione Imperativa.

3.4 UN PO' DI STORIA

Il primo linguaggio funzionale, introdotto già negli anni '30 da Alonzo Church, è stato il $\lambda - calcolo$.

Attualmente il tema della *Programmazione Funzionale* evoca più che altro linguaggi come il LISP (LISt Processor) sviluppato a partire dalla fine degli anni '50. Ma, secondo una moderna visione della programmazione, tale linguaggio non può più essere considerato funzionale a causa della presenza di molte caratteristiche tipicamente imperative. Alcuni esperti ritengono addirittura che l'aver associato per molti anni la *Programmazione Funzionale* con il LISP abbia provocato un ritardo di dieci anni nello sviluppo di questo paradigma. Di certo c'è che l'aspetto sintattico del LISP, così poco attraente, ha allontanato le persone dal *Paradigma Funzionale*.

Da qualche anno l'associazione corretta è con i linguaggi ML, Miranda e Haskell, cioè linguaggi praticamente sconosciuti al grande pubblico, ma piuttosto noti negli ambienti scientifici ed accademici. Occorre tuttavia considerare che anche strumenti molto popolari, come Derive o SQL interattivo, appartengono al *Paradigma Funzionale*; non solo, anche negli applicativi come i fogli di calcolo (Excel di Microsoft, Calc di OpenOffice) i valori contenuti nelle celle spesso provengono proprio dalla valutazione di formule (espressioni) nelle quali compaiono chiamate a funzioni e quindi in perfetto stile Funzionale.

Per quanto riguarda l'ambiente di *Programmazione Funzionale* Maxima, esso discende direttamente da *Macsyma*(contrazione di *MAC symbolic manipulation*), il leggendario progetto sviluppato negli anni Sessanta negli Stati Uniti presso il Massachusetts Institute of Technology con fondi del Department of Energy. Tale progetto vedeva impegnato un folto gruppo di ricercatori, ciascuno dei quali si dedicò allo sviluppo mediante il linguaggio LISP di una specifica parte del progetto.

Macsyma all'epoca rappresentò una vera rivoluzione, tanto da risultare incomprensibile da parte di coloro che si trovavano al di fuori dell'ambiente di ricerca, ma vale la pena sottolineare che gli attuali sistemi di calcolo simbolico usati negli ambienti accademici (Derive, Mathematica, Maple) sono ispirati a tale progetto.

Nel 1982 William Shelter scorporò da Macsyma un ramo che denominò Maxima; egli ottenne nel 1998 il rilascio del codice sorgente e proseguì nello sviluppo del sistema nello spirito del movimento open-source per permettere ad un numero sempre più elevato di utenti di poterlo conoscere.

Maxima venne quindi distribuito con la GNU General Public License.

Dopo la morte di Shelter, avvenuta nel 2001, si costituì un gruppo formato da utenti e da sviluppatori di software che continua tuttora con estrema dedizione nella manutenzione di Maxima affinchè gli sforzi di Shelter per mantere vivo il progetto Macsyma non andassero perduti.

Esistono versioni di Maxima supportate dai più comuni sistemi operativi e il download della versione desiderata è possibile dal sito

`http://maxima.sourceforge.net/`.

4

L'AMBIENTE DI PROGRAMMAZIONE FUNZIONALE MAXIMA

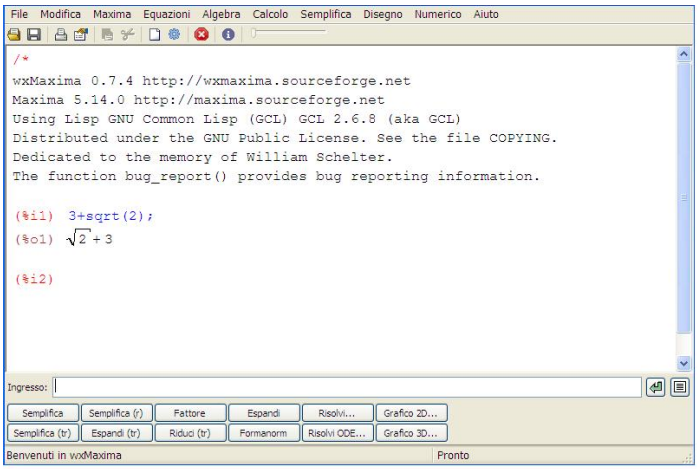
Questo capitolo non vuole rappresentare un *manuale d'uso di Maxima*: sulla rete Internet se ne possono agevolmente trovare ed, inoltre, la guida in linea di Maxima è spesso sufficientemente chiara.

Si preferisce piuttosto fornire alcune indicazioni di base per poter velocemente approcciarsi con tale ambiente di programmazione, indicazioni derivanti dall'esperienza di insegnamento e di programmazione durante le attività di Laboratorio di Matematica svolte negli ultimi tre anni.

4.0.1 *Linee di comando*

La versione base *xMaxima* si presenta con un'interfaccia grafica molto scarna: un menu con pochi comandi disposti sopra una finestra di dialogo.

E' possibile invece utilizzare la versione *wxMaxima* (quella alla quale faremo sempre riferimento di seguito) con interfaccia grafica decisamente più accattivante: oltre al menu di comandi (arricchito rispetto alla versione xMaxima) sono presenti diversi pulsanti per velocizzare alcune operazioni; inoltre, è presente una linea di ingresso nella quale scrivere le espressioni da semplificare; un'espressione scritta nella linea di ingresso viene riportata dopo la pressione del tasto *invio* nella finestra di algebra ed etichettata con (%i1) dove i sta proprio per input; il risultato ottenuto dalla sua semplificazione viene invece associato ad una linea di output etichettata con (%o1) dove o sta proprio per output.



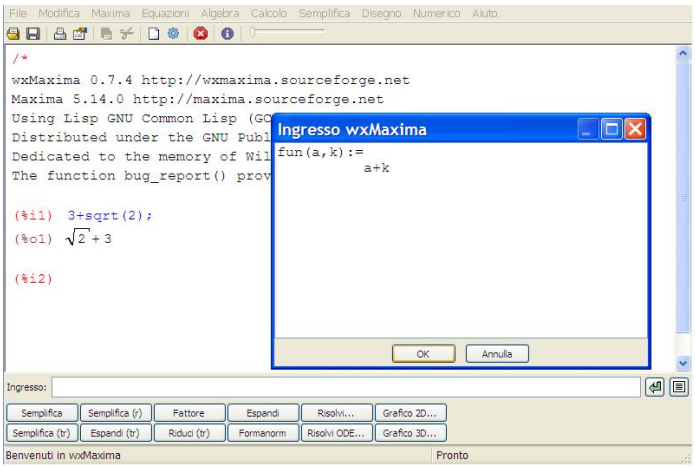
Le successive linee di input verranno individuate da etichette composte dai simboli (%i...) in ciascuna delle quali la lettera i è seguita da un numero.

In una linea di input è possibile scrivere un'espressione da semplificare (o la definizione di una funzione) e il risultato verrà posto in una corrispondente linea di output individuata dai simboli (%o...); come per le linee di input, il valore numerico che segue la lettera o indica la posizione ordinale della linea.

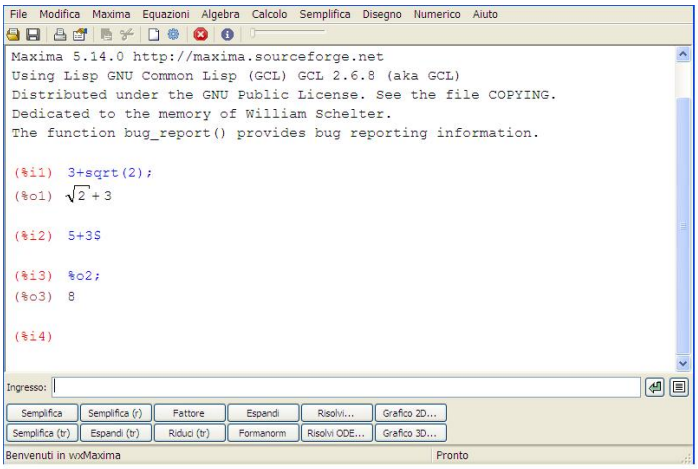
Ogni linea di input avrà quindi una corrispondente linea di output e saranno presenti coppie di linee del tipo:

```
(%i1) ...
(%o1) ...
```

Nelle linee di input le espressioni potranno essere scritte anche utilizzando più righe di testo aprendo una finestra di ingresso mediante la pressione del bottone che si trova a destra della linea di ingresso.



In qualche caso, ad esempio nella definizione di una funzione, potrebbe non essere necessario visualizzare la linea di output; sarà allora sufficiente concludere la scrittura con il simbolo $, tenendo presente però che nella memoria di Maxima rimane comunque associata alla linea di input appena scritta la sua corrispondente linea di output, alla quale sarà sempre possibile far riferimento tramite la sua etichetta.



4.1 LISTE

La *lista* è l'unica struttura dati di cui si possa disporre nella programmazione funzionale; si tratta di una collezione di oggetti, detti elementi della *lista*, che possono essere anche non omogenei.

La *lista* viene delimitata dalle parentesi quadre *[ ]* e di seguito sono riportati alcuni esempi:

- $[1, 2, 3]$ lista costituita da un sottoinsieme dei numeri naturali ;
- $[y = 3x, y = 5x]$ lista contenente le equazioni di due rette ;

- [] lista vuota ;
- $[[1,2],[3,4]]$ lista contenente due liste, ciascuna formata da due elementi che potrebbero rappresentare le coordinate di due punti ;
- $[1, y = x, [1,2,3]]$ lista contenente 3 elementi di diversa natura .

Ad una lista può essere associato un nome mediante l'operatore :

```
l:[1,2,3,4,5]
```

e quindi per individuare, ad esempio, il terzo elemento della lista `l` si scrive:

```
l[3]
```

Se ogni elemento della Lista è legato agli altri da una relazione, allora la lista può essere costruita mediante la funzione predefinita *Makelist*:

```
Makelist( generico elemento, indice, val min, val max);
```

dove:

- *generico elemento*: legge che lega i diversi elementi della lista;
- *indice*: parametro che compare nella legge suddetta e che varia (nei casi più semplici rappresenta la posizione assunta dagli elementi all'interno della lista);
- *val min*: valore minimo assunto dall'indice;
- *val max*: valore massimo assunto dall'indice.

L'indice varia a partire dal valore minimo con passo costante, pari a 1, fino a raggiungere il valore massimo.
Esempio:

```
(%i1) Makelist(3+k,k,-2,2);
(%o1)                       [1, 2, 3, 4, 5]
```

Esempio:

```
(%i1) Makelist(y=m*x,m,1,5);
(%o1)                       [y=x, y=2x, y=3x, y=4x, y=5x]
```

4.2 FUNZIONE PREDEFINITA EV

La funzione predefinita di Maxima *ev* è una delle funzioni più potenti. Essa prevede molteplici applicazioni e di seguito ne vengono proposte solo alcune.

- Per valutare un'espressione assegnando ad una variabile un valore:

```
(%i18) f:x+1$
(%i19) ev(f,x=1);
(%o19) 2
```

- Per sostituire una variabile in una espressione:

```
(%i20) ev(a+3*b-c,c=a+2);
(%o20) 3 b - 2
```

- Per applicare un operatore in notazione post-fissa, ad esempio la funzione predefinita *float* per approssimare e restituire la notazione decimale del risultato di una espressione numerica:

```
(%i2) ev(1+1/4,float);
(%o2)                  1.25
```

4.3 GRAFICI

Per rappresentare graficamente una funzione di cui sia nota l'equazioni nella sua forma esplicita $y = f(x)$ si usa la funzione predefinita *plot2d* con la seguente sintassi:

```
plot2d(f(x),[range x],opzioni grafiche);
```

E' importante ricordare che il primo parametro di *plot2d* deve rappresentare solo il secondo membro $f(x)$ dell'equazione e che *[range x]* è una lista di tre elementi che deve contenere la variabile indipendente x e gli estremi dell'intervallo nel quale far variare x per visualizzare il grafico della funzione, cioè *[x,min,max]*.

L'indicazione del range per la variabile dipendente y è opzionale e la sintassi per definirlo è la stessa di *[range x]*:

```
plot2d(f(x),[range x],[range y],opzioni grafiche);
```

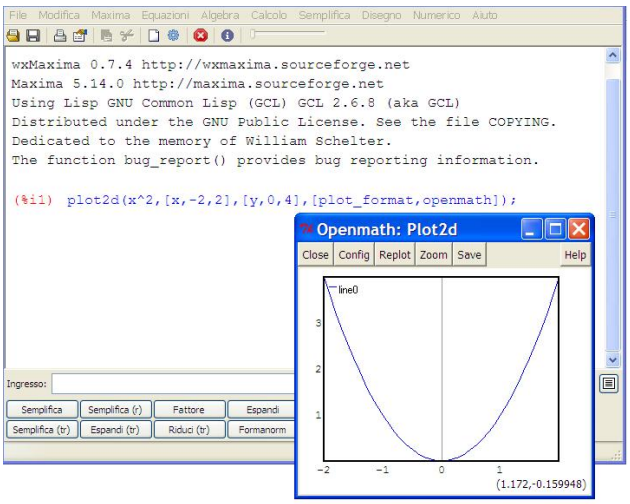
La scelta di *[range x] e [range y]* permette di individuare quindi un rettangolo sul piano cartesiano e, se la scelta è coerente con l'andamento della funzione, in tale porzione di piano verrà visualizzato il grafico della funzione stessa.

Per quanto riguarda le *opzioni grafiche* si consiglia di consultare il manuale in linea relativo alla versione di wxMaxima che si sta utilizzando poichè il loro utilizzo può essere molto vario.

Ad esempio, nella versione wxMaxima 0.7.4 l'*opzione grafica*

```
[plot_format, openmath]
```

viene utilizzata per rappresentare il grafico della funzione in una finestra separata da quella che contiene la sessione Maxima.



Nel caso in cui sia nota l'equazione della funzione in forma implicita $g(x, y) = 0$ sarà necessario usare la funzione predefinita *solve* per esplicitare la variabile y:

Esempio:

```
(%i10) solve(2*x+y-1=0,y);
(%o10)                        [y=1-2*x]
```

Si osservi che il risultato della funzione *solve* viene restituito in una lista che, nel nostro esempio, contiene un solo elemento e cioè l'equazione in forma esplicita. Ora è possibile utilizzare un'altra funzione predefinita, la funzione *rhs* (Right Side) per isolare il secondo membro dell'equazione ottenuta, ricordando sempre che l'equazione in forma esplicita ottenuta con *solve* è contenuta in una lista di un solo elemento:

```
(%i11) rhs(%o10[1]);
(%o11)                        1-2*x
```

Finalmente, il contenuto della linea di output %o11 potrà rappresentare il primo parametro della funzione *plot2d*:

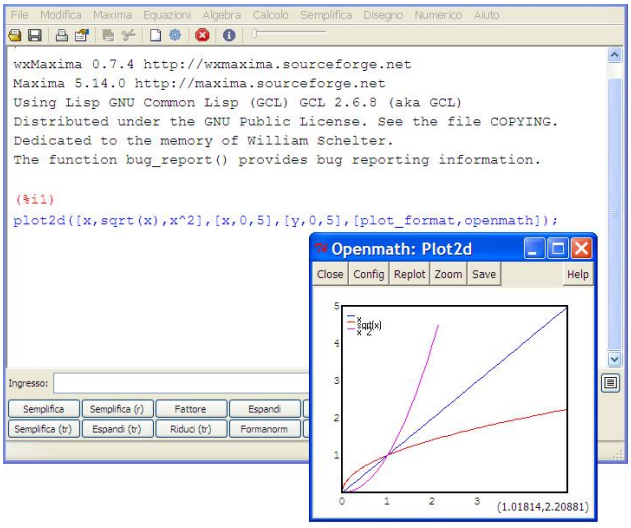
```
(%i12) plot2d(%o11,[x,-2,2],[plot_format, openmath]);
```

Esercizio:
Rappresentare il grafico di $x^2 + y^2 = 1$.
Per visualizzare il grafico di più funzioni è necessario costruire una lista che contenga tutti i secondi membri delle loro equazioni in forma esplicita e tale lista costituirà il primo parametro della funzione *plot2d*:

```
plot2d(lista funzioni,[range x], opzioni grafiche);
```

Esempio:

```
plot2d([x,sqrt(x),x^2], [x,0,10],[plot_format, openmath]);
```



Quando è necessario rappresentare insiemi di punti di cui siano note le coordinate (memorizzate in una lista), la funzione *plot2d* deve essere usata con una diversa sintassi:

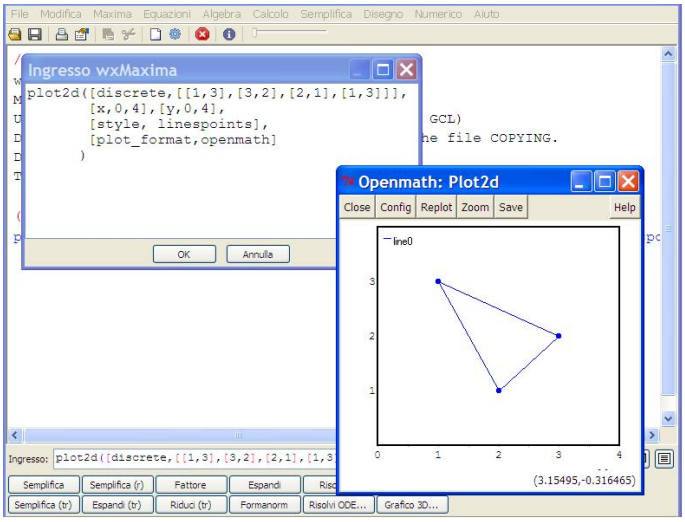
```
plot2d([discrete,listapunti],[range x], opzioni grafiche);
```

che, per default, e cioè senza specificare opzioni grafiche particolari, produce la spezzata i cui vertici sono individuati dai punti contenuti nella

`listapunti`

.

N.B. *[range x]* è un parametro obbligatorio e quindi va sempre indicato, anche nella rappresentazione grafica discreta; *[range y]*continua ad essere un parametro opzionale.



Si presti particolare attenzione al modo in cui è stata indentata la scrittura della funzione *plot2d*, alle opzioni relative allo stile di rappresentazione grafica (*linespoints* significa che i punti vengono rappresentati con dei pallini e uniti da linee), alle coppie di parentesi quadre usate per scrivere la lista dei punti da rappresentare:

- ogni punto è rappresentato da una lista di due elementi [x,y];
- i punti devono essere raccolti in una lista

`[[ , ],[ , ],...,[ , ]]`

e infine al fatto che nella lista dei punti è stato riscritto alla fine il primo punto in modo che il triangolo rappresentato risultasse chiuso.

4.4 ARCHIVIO DI FUNZIONI DEFINITE DALL'UTENTE

Le funzioni definite dall'utente possono essere salvate in un file di testo (con estensione .txt) allo scopo di creare un archivio di funzioni che potranno essere caricate in memoria e utilizzate nelle successive sessioni di lavoro con Maxima come fossero predefinite.

La procedura consigliata per la costruzione e l'utilizzo di tale archivio è dunque la seguente:

- aprire una sessione di Maxima e definire una funzione;
- verificare la correttezza della funzione costruita prevedendo adeguate prove di collaudo;
- copiare solo la definizione della funzione mediante la consueta procedura Copia-Incolla in un file di testo (è possibile usare un qualunque text editor, anche blocco-note);
- salvare tale file prestando attenzione a scegliere il formato testo (estensione del file .txt).

Le funzioni contenute nel file di testo creato potranno essere copiate e caricate in memoria in una nuova sessione di Maxima, rendendo possibile il loro utilizzo proprio come avviene per le funzioni predefinite.

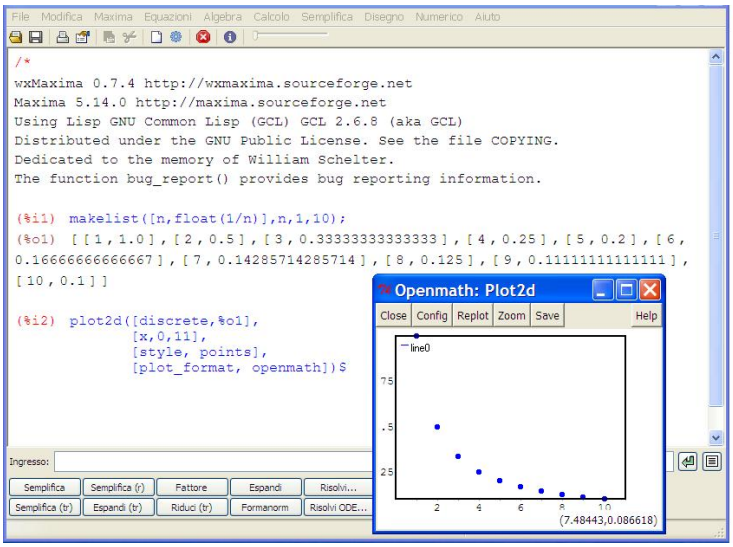
4.4.1 *Applicazioni ed esempi*

Per studiare una successione numerica reale di termine generale a_n è possibile costruire la lista dei punti $[n, a_n]$ e analizzare la successione da due punti di vista: quello numerico, osserrvando le ordinate dei punti; quello grafico, osservando l'andamento grafico dei termini della successione.

Esempio 4.4.1.

$$a_n : \begin{array}{rcl} \mathbb{N} & \longrightarrow & \mathbb{R} \\ n & \longmapsto & \dfrac{1}{n} \end{array}$$

Con Maxima:



Parte V

ZIBALDONE

5 PROGETTO *MULTIPLI, DIVISORI E NUMERI PRIMI*

5.1 MULTIPLI E DIVISORI

I multipli naturali del numero A sono

$$0, A, 2A, 3A, 4A, \ldots$$

cioè nA con $n \in N$.

Nell'ambiente di programmazione funzionale *wxMaxima* si possono disporre i multipli del numero A all'interno di una struttura *Lista*, applicando la funzione *Makelist* per la sua costruzione, ad esempio:

```
Makelist(A*i,i,0,5);
```

produce

```
[0, A, 2*A, 3*A, 4*A, 5*A]
```

E' possibile definire una funzione che restituisca la lista suddetta, dati A, numero di cui determinare i multipli, ed n, ricordando che $n+1$ è la quantità di multipli compresi 0 e A stesso:

```
multipli(A,n):= Makelist(A*i,i,0,n);
```

Di seguito sono riportati alcuni esempi di chiamata e valutazione della funzione *multipli*:

```
multipli(2,4)    produce    $[0,2,4,6,8]$
multipli(10,3)   produce    $[0,10,20,30]$
```

Posto che il numero B è divisore del numero A, ovvero A è multiplo di B, quando il resto della divisione intera tra A e B è uguale a zero, si evince che la ricerca dei divisori si riconduce al calcolo del resto della divisione intera:

$$A : B = Q \quad \text{con resto} \quad r \quad \Rightarrow A = B * Q + r \Rightarrow r = A - B * Q$$

E' evidente che r è una quantità minore di B non negativa:

$$0 \leqslant r < B$$

.

5.1.1 *Resto della divisione intera*

Supponendo di non disporre dell'operazione di divisione intera tra due numeri, come capita in molti ambienti di programmazione, per determinare il resto sarà sufficiente ritornare con la mente alle scuole elementari.

La domanda che la mestra ci poneva era la seguente: *quante volte* B *sta in* A*?* ovvero *dati* A *elementi, quanti gruppi di* B *elementi si possono formare?*

Una volta formati tutti i possibili gruppi, è possibile che rimangano degli elementi isolati in quantità certamente inferiore a B (altrimenti sarebbe possibile costituire un altro gruppo di B elementi).

La quantità degli elementi rimasti rappresenta il resto r della divisione intera tra A e B, mentre il numero di gruppi contenenti B elementi rappresenta il quoziente Q.

Se B sta in A un numero esatto di volte non rimangono elementi isolati e quindi il resto della divisione intera tra A e B è nullo; in tal caso si può dire che B *è divisore di* A, ovvero che A *è multiplo di* B.

Esempio 5.1.1. 11 *è divisibile per* 3*? ovvero* 11 *è multiplo di* 3*?*
Consideriamo i seguenti 11 elementi (schematizzati con X)

```
X X X   X X X   X X X   X X
```

Dalla rappresentazione è evidente che si possono raggruppare in tre gruppi da 3 elementi e 2 di essi rimangono isolati.
E' possibile realizzare tali schieramenti nel modo seguente:

```
dati 11 elementi                  X X X X X X X X X X X

provo a formare 1 gruppo da 3     X X X     X X X X X X X X

oppure

provo a formare 2 gruppi da 3     X X X     X X X     X X X X X

oppure

provo a formare 3 gruppi da 3     X X X     X X X     X X X     X X
```

Ora ci si accorge che non è possibile formare più di tre gruppi da 3 elementi e che due elementi rimangono fuori dai gruppi.
In ciascuno di questi passaggi si parte sempre dall'insieme di 11 elementi e si cerca di raggrupparli per 3 in diverse maniere, fino a che si rimane con una quantità di elementi inferiore a 3.
Operando numericamente sulle quantità A *e* B*, la determinazione del resto della divisione intera equivale a calcolare le seguenti differenze:*

$$A - B$$
$$A - 2 * B$$
$$A - 3 * B$$
$$\vdots$$

e cosi' via fino a quando si arriva ad una differenza minore di B (nulla quando B *è divisore di* A*).*
Nell'esempio numerico si ottiene:

$$11 - 3 = 8$$
$$11 - 2 * 3 = 5$$
$$11 - 3 * 3 = 2$$

e ci si può fermare affermando che 3 non è divisore di 11 poiché il risultato dell'ultima differenza, e cioè il resto, è diverso da zero.
E' possibile realizzare in maniera semplice il calcolo delle suddette differenze mediante la struttura Lista e la funzione Makelist:

```
Makelist(A-i*B,i,1,n);
```

Esempio 5.1.2. *Se* $A = 11$ *e* $B = 3$ *la chiamata:*

```
Makelist(11-i*3,i,1,11);
```

produce il seguente risultato:

```
[8,5,2,-1,-4,-7,-10,-13,-16,-19,-22]
```

Osservando il risultato ottenuto nell'esempio si possono porre i seguenti quesiti:

1. perché al parametro n è stato assegnato il valore 11?
2. come si risponde alla domanda *'B è divisore di A?'*
3. come si può individuare il resto nel caso in cui $A < B$?

Le risposte possono essere le seguenti:

1. La scelta effettuata deriva dal fatto che B può assumere anche il valore 1 e in questo caso 1 sta nel numero 11 esattamente 11 volte; può sembrare privo di senso l'assegnazione del valore 1 a B, poiché è noto che 1 è divisore di qualunque numero; allora si potrebbe pensare di poter assegnare a B come minimo valore 2, così da non rendere necessario che l'indice i raggiunga il valore 11, bensì l'intero che precede la metà di 11 (dato che 11 è dispari) e cioè 5; ma questo implica avere delle competenze che alle scuole elementari certamente non si hanno e cade così l'ipotesi iniziale di riuscire a calcolare il resto della divisione intera pur non sapendo eseguire la divisione stessa. Quindi la chiamata alla funzione Makelist potrebbe diventare:

   ```
   Makelist( A - i*B, i ,  1, A);
   ```

   accettando il fatto di sovradimensionare la lista prodotta.

2. proprio a causa del sovradimensionamento della lista, il resto della divisione intera si può individuare all'interno della lista come quell'elemento che precede il primo negativo, nel nostro caso è il terzo elemento e vale 2; quindi la risposta alla domanda è: 3 non divide 11 poiché il resto è non nullo.

3. Se $A < B$, alla domanda *Quante volte B sta in A?* si risponde ovviamente *zero volte con resto A* (provare con i numeri 3 e 5!); ma la chiamata

   ```
   Makelist( 3 - i*5, i ,  1, A);
   ```

   produce

   ```
   [-2,-7,-12]
   ```

   dove evidentemente non compare alcun numero positivo o nullo (mentre è noto che il resto deve necessariamente esserlo!!!); quindi, per individuare il resto della divisione tra 3 e 5, è necessario far in modo che il primo elemento della lista sia proprio 3 (in generale A) e ciò si ottiene assegnando, nella chiamata alla funzione *makelist*, al parametro i il valore iniziale 0 .

Risulta quindi evidente che la chiamata alla funzione Makelist che tenga conto di tutti i possibili casi dovrà essere:

```
Makelist( A - i*B, i ,  0, A);
```

e la funzione che dati A e B produca la lista delle differenze suddette sarà:

```
lista_diff(A,B):= Makelist(A-i*B,i,0,A);
```

Di seguito vengono proposti alcuni esempi di chiamata e di valutazione della funzione *lista_diff*:

```
lista_diff(11,3)     produce     [11,8,5,2,-1,-4,-7,-10,-13,-16,-19,-22]
```

```
lista_diff(4,2)      produce     [4,2,0,-2,-4]

lista_diff(2,4)      produce     [2,-2]

lista_diff(6,5)      produce     [6,1,-4,-9,-14,-19,-24]
```

Si può contestare il fatto che il risultato prodotto dalla funzione *lista_diff* non è propriamente il resto, e cioè un numero da poter essere successivamente elaborato, bensì la lista delle differenze consecutive che consentono di giungere all'individuazione del resto mediante l'osservazione degli elementi della lista stessa.

Questo, ovviamente, risulta scomodo e poco pratico, perché quell'unico elemento della lista che rappresenta il resto si confonde tra tutti gli altri elementi e la sua individuazione implica un processo di osservazione successivo all'applicazione della funzione *lista_diff*.

E' quindi indispensabile pensare ad una strategia per metterlo in evidenza all'interno della lista.

L'idea potrebbe essere quella di sostituire tutti gli elementi della lista che non possono rappresentare il resto con il numero 0, cioè tutti gli elementi che non soddisfano alle condizioni

$$0 \leqslant elementolista < B$$

vengono sostituiti da 0.

Questo si realizza con Maxima mediante l'applicazione del costrutto condizionale *if then else*:

```
if  A-i*B<0 then 0
            else if A-i*B>=B then 0
                             else A-i*B
```

oppure, usando gli operatori logici or e and

```
if  (A-i*B<0) or (A-i*B>=B) then 0
                           else A-i*B
```

oppure

```
if  (A-i*B>=0) and (A-i*B< B) then A-i*B
                              else 0
```

Quindi la nuova definizione della funzione *lista_diff* è:

```
lista_diff(A,B):= Makelist(if (A-i*B>=0) and (A-i*B< B) then A-i*B
                                                        else 0,
                           i,
                           0,
                           A
                          );
```

dove è stata scelta una sola delle possibili scritture proposte.

N.B. Prestare attenzione al modo in cui sono stati indentati sia la scrittura della funzione Makelist che il costrutto *if then else*: risulta più semplice il controllo sulle coppie di parentesi *()*

La chiamata

```
lista_diff(11,3)
```

produce ora il seguente risultato:

```
[0,0,0,2,0,0,0,0,0,0,0,0]
```

il quale non lascia nessun dubbio circa il valore del resto nella divisione intera tra 11 e 3!

Rimane però un nodo da sciogliere: il risultato prodotto non è ancora *il resto*, ma una lista che lo contiene!

Ricordando comunque che 0 è l'elemento neutro rispetto all'operazione di somma viene spontaneo pensare di poter sommare tutti gli elementi della lista prodotta ottenendo quindi un numero che rappresenta proprio il resto cercato; questo si può realizzare mediante la funzione predefinita di Maxima *Sum*.

E' possibile quindi definire la funzione che associa ad ogni coppia di numeri naturali (A, B), con $B \neq 0$, il numero naturale r resto della divisione intera tra A e B:

$$\begin{aligned} \textit{resto}: \quad & \mathbb{N} \cdot \mathbb{N} \longrightarrow \mathbb{N} \\ & (A, B) \longmapsto r \end{aligned}$$

e la codifica Maxima di tale funzione è:

```
resto(A,B):= Sum(if (A-i*B>=0) and (A-i*B<B) then A-i*B
                                                 else 0,
                 i,
                 0,
                 A
                );
```

Quindi, la chiamata

```
resto(11,3)
```

produce finalmente 2.

5.1.2 *Resto della divisione intera: approccio ricorsivo*

Purtroppo la funzione *resto* risulta poco conveniente in molti casi.

Esempio 5.1.3. *la chiamata resto(100,99)*
pur non producendo la lista di 101 elementi $[0, 1, 0, ..., 0]$*, li calcola comunque tutti per poterli sommare e fornire così come risultato il numero* 1*; ma è possibile osservare che il resto è presente già nel secondo elemento della lista e gli altri 99 elementi sono stati calcolati inutilmente.*

Esempio 5.1.4. *Si voglia verificare se il numero* 149847 *è divisibile per* 3842*, il che equivale a determinare il resto della divisione intera tra i due numeri e verificare che non è uguale a zero:*
Semplificando la seguente scrittura in una sessione di Maxima

```
if resto(149847,3842)=0 then "ok"
                        else "ko"
```

si noterà che i tempi di esecuzione sono relativamente elevati ed è facile intuirne il motivo: la funzione resto deve sommare gli elementi di una lista di dimensione pari al dividendo della divisione intera più 1 e cioè 149848*!!!*

Si provi ad immaginare quali potrebbero essere i tempi di esecuzione di una funzione che individui tutti i numeri pari in una lista di 100 numeri!!!

E' evidente che diventa necessario calcolare il resto mediante un nuovo algoritmo che permetta di interrompere il calcolo delle differenze consecutive A-i*B (al variare di i) non appena si verifica che il risultato di una di queste soddisfa alle condizioni proprie del resto e cioè $(A - i * B >= 0) and (A - i * B < B)$.

Osservando che i numeri A e $A - B$ hanno lo stesso resto nella divisione per B l'algoritmo potrebbe essere il seguente:

```
se $A<B$ allora il resto è A
                altrimenti si calcola A-B e
                           si riapplica l'algoritmo ai numeri A-B e B
```

Come si può osservare si tratta di un algoritmo ricorsivo e in tale procedimento si può riconoscere la *base della ricorsione* $A < B$, cioè la condizione che permette di interrompere il processo ricorsivo, e la riapplicazione di *resto* al caso più semplice $A - B$, B.

La traduzione di tale algoritmo in una funzione ricorsiva Maxima è quindi:

```
resto(a,b):= if a<b then a
                    else resto(a-b,b)$
```

Il confronto con la funzione *resto* elaborata mediante l'approccio non ricorsivo

```
resto(A,B):= Sum(if (A-i*B>=0) and (A-i*B<B) then A-i*B
                                             else 0,
               i,
               0,
               A
               );
```

mette in luce l'estrema semplicità ed eleganza della funzione ricorsiva, oltre che la già sottolineata efficienza.

Tecniche di programmazione

La funzione *resto*, sia nella prima versione proposta che in quella ricorsiva (anche se ovviamente è preferibile rifarsi sempre a quest'ultima), può essere utilizzata assieme al costrutto *if then else* ad esempio per determinare se un numero è pari o dispari nel seguente modo:

```
pari(n):= if resto(n,2)=0 then "ok"
                          else "ko"$
```

e ovviamente la chiamata

```
pari(11)
```

produce `ko`, mentre la chiamata

```
pari(8)
```

produce `ok`.

Per rendere più leggibili le funzioni costruite può essere conveniente utilizzare la seguente tecnica:

- dapprima si definisce una funzione di nome e_pari (si legge 'è pari', ma Maxima non accetta la lettere accentate nei nomi di funzioni!) che definisce la condizione che qualora sia verificata dà significato al nome stesso della funzione:

  ```
  e_pari(n):=resto(n,2)=0$
  ```

- di seguito si applica tale funzione all'interno di un costrutto *if then else*:

```
if e_pari(A) then "ok"
             else "ko"
```

Con questa tecnica il senso delle scritture è decisamente più chiaro e le frasi risultano simili al linguaggio comune; ne trova giovamento l'immediatezza della comprensione.

Finalmente si può iniziare a parlare di divisori!

Abbiamo già più volte sottolineato che il numero B è divisore di A se la divisione intera tra A e B produce resto nullo.

Ma quanti e quali sono tutti i divisori di A?

Intituitivamente si può pensare di provare la divisibilità di A per tutti i numeri che lo precedono, iniziando dal numero 2, poichè 1 è certamente divisore di tutti i numeri naturali:

Esempio 5.1.5. *Determinare i divisori di 8:*

```
          1  2  3  4  5  6  7  8

divisori: 1  2     4           8
```

Si può osservare che dalla lista di tutti i naturali da 1 a 8 sono stati estratti solo quelli che dividono 8.

L'esperienza acquisita nella costruzione di funzioni Maxima ci consente di definire facilmente una funzione che realizzi quanto visto nell'esempio mediante l'approccio non ricorsivo:

```
divisori(A):=makelist(if resto(A,i)=0 then i
                                      else 0,
                      i,1,A)

divisori(8)  produce   [1,2,0,4,0,0,0,8]
```

Ma, come più volte osservato, mediante questo approccio si costruisce una lista decisamente sovvradimensionata ed è quindi preferibile definire una funzione ricorsiva che costruisca una lista contenente solo quei numeri compresi tra 1 e A che siano divisori di A.

Per far ciò è necessario utilizzare una variabile di supporto i, che assuma come valore iniziale A, la quale permette di individuare la base della ricorsione e cioè $i = 1$ (infatti, in tal caso la lista dei divisori è $[1]$) e l'azione ricorsiva:

```
se resto(A,i)=0 allora aggiungi i alla lista dei divisori
                altrimenti riapplica l'algoritmo ai numeri A e i-1
```

La definizione Maxima della funzione *divis* è dunque la sseguente:

```
divis(A,i):= if i=1 then [1]
                     else append(divis(A,i-1),
                                 if resto(A,i)=0 then [i]
                                                 else []
                                 )
```

e la chiamata

```
divis(8,8)  produce   [1,2,4,8]
```

Tecniche di programmazione
Nella chiamata alla funzione *divis* così costruita è necessario che i due parametri a e i assumano lo stesso valore affinchè il risultato prodotto sia corretto. Può essere quindi conveniente definire una nuova funzione che preveda come unico parametro A e che si incarichi di richiamare *divis* con i parametri corretti:

```
divisori(A):=divis(A,A)
```

L'ultima funzione realizzata, pur applicando la definizione ricorsiva ed esprimendo tutta l'eleganza di tale approccio, non è ancora ottimale poichè è noto che i divisori propri di A possono essere ricercati tra 2 e $\frac{A}{2}$ e ai numeri così trovati si possono aggiungere 1 e A, detti *divisori impropri* di A.

Per giungere ad una versione ottimizzata si deve cambiare punto di vista, pensando che il parametro i debba assumere valore iniziale 1 anzichè A, e introducendo un ulteriore parametro d che rappresenti la lista, inizialmente vuota [], che alla fine del processo ricorsivo conterrà tutti i divisori di A; la definizione è quindi la seguente:

```
divis_ott(A,i,d):=if i>A/2 then append(d,[A])
                        else divis_ott(A,
                                       i+1,
                                       if resto(A,i)=0 then append(d,[i])
                                                       else d
                                      )
```

La chiamata:

```
divis_ott(8,1,[])    [1,2,4,8]
```

produce ovviamente ancora

```
[1,2,4,8]
```

ma il numero dei passaggi ricorsivi realizzati è decisamente inferiore rispetto alla versione precedente (provare a contarli!!!).

Tecniche di programmazione
Ancora una volta risulta conveniente definire una nuova funzione che preveda come unico parametro A e che si incarichi di richiamare *divis_ott* con i parametri corretti:

```
divisori_ott(A):=divis_ott(A,1,[])
```

Ricordando che nel caso in cui A sia un numero primo i suoi divisori sono solo 1 e A stesso, nella definizione di una funzione per individuare la primalità del numero A è sufficiente verificare se la lista restituita dalla funzione *divisori_ott(A)* contiene solo due elementi, nel qual caso saranno sicuramente 1 e A.

Utilizzando quindi una delle tecniche di programmazione suggerite si può definire:

```
e_primo(A):=length(divisori_ott(A))=2
```

e la verifica della primalità sarà realizzata mediante il costrutto *if then else*:

```
if e_primo(A) then "ok"
              else "ko"
```

6 ESERCIZI

6.1 ESERCIZIO 1

Obiettivo dell'esercizio: allenare gli allievi nell'uso della funzione Makelist per la costruzione di liste contenenti elementi appartenenti ad insiemi numerici.

Modalità di esecuzione:

- FASE 1: disporsi a coppie; ciascuna coppia elabora una legge che leghi gli elementi della lista prodotta dalla coppia (tempo assegnato 15 min.);
- FASE 2: ogni coppia propone alla classe gli elementi della propria lista; la coppia 'vince' se nessuno dei compagni riesce ad individuare la legge, altrimenti il vincitore è il compagno che ha individuato la legge.

Esempio:
la lista

$$[\frac{1}{2}, \frac{2}{3}, \frac{3}{4}, \frac{4}{5}]$$

proviene ovviamente dalla semplificazione di

$$Makelist(\frac{n}{n+1}, n, 1, 4).$$

Ovviamente la vincita può essere stabilita dal docente o concordata con la classe.

Il docente deve anche indirizzare gli allievi nella scelta della legge, ad esempio limitando le operazioni che possono essere utilizzate $(+, -, *, /, ^)$, perchè non risulti troppo vasto l'insieme delle possibili leggi.

6.2 ESERCIZIO 2

Obiettivo dell'esercizio: allenare gli allievi nella definizione di nuove funzioni Maxima e nell'uso del costrutto condizionale if then else.

Modalità di gioco:

- FASE 1: disporsi a coppie; ciascuna coppia elabora una funzione, che chiamerà genericamente f, nella quale si applichi il costrutto condizionale if then else;
- FASE 2: ogni coppia propone alla classe alcune chiamate alla funzione costruita e i risultati prodotti; la coppia 'vince' se nessuno dei compagni riesce ad individuare la definizione della funzione, altrimenti il vincitore è il compagno che la individua.

Esempio:

```
f(3,2)    produce    [2,3]

f(5,10)   produce    [5,10]

f(1,1)    produce    [1]
```

E' facile individuare che la funzione f può essere definita come segue:

```
$$f(a,b):= if a>b then [b,a]
                  else if a<b then [a,b]
                              else [a];$$
```

7 LE MATRICI

7.1 INTRODUZIONE

Si definisce matrice ...
Matrice riga, colonna, rettangolare, quadrata.
Dimensione di una matrice (oppure ordine).
Relazione d'uguaglianza.
Sottomatrici e minori complementari.

7.2 CARATTERISTICHE DI MATRICI QUADRATE

Diagonale principale e secondaria.
mMtrice quadrata: - simmetrica, diagonale, scalare, unitaria
- emisimmetrica
- triangolare: inferiore, superiore

7.3 OPERAZIONI TRA MATRICI

Somma: definizione e proprietà
Prodotto per uno scalare: definizione e proprietà
Prodotto righe per colonne: definizione e proprietà
Trasposizione

7.4 DETERMINANTE DI MATRICI QUADRATE

Definizione di determinante (come funzione)
Complemento algebrico
Sviluppo di Laplace

7.5 MATRICI QUADRATE INVERTIBILI

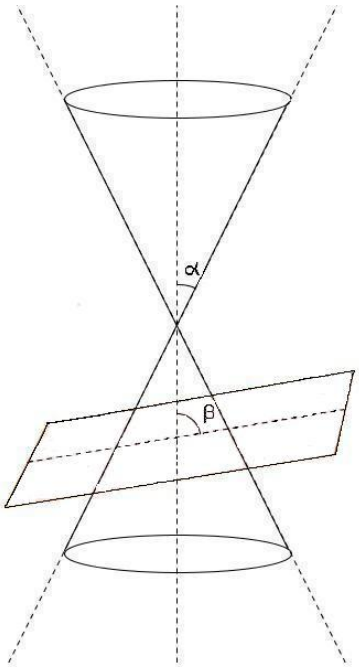
Def. matrice inversa
Matrici invertibili.

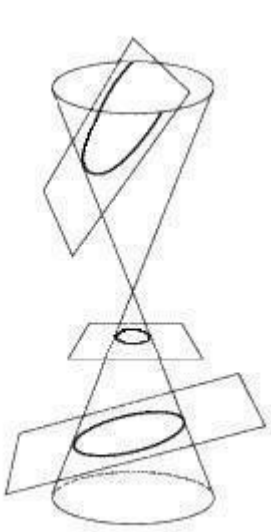
8

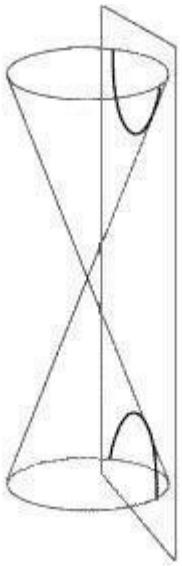
LE CONICHE IN FORMA MATRICIALE

8.1 SEZIONI CONICHE

Di seguito viene proposta la rappresentazione grafica delle sezioni coniche così come sono state classificate da Apollonio







8.2 EQUAZIONE E MATRICE DI UNA CONICA

Di seguito verranno esposti in modo sintetico riferimenti teorici relativi alla rappresentazione di coniche in forma matriciale.

L'equazione in forma canonica di una conica è la seguente:

$$ax^2 + by^2 + cxy + dx + ey + f = 0$$

Assegnando ai sei coefficienti a, b, c, d, e, f diversi valori si ottengono le equazioni delle diverse coniche.

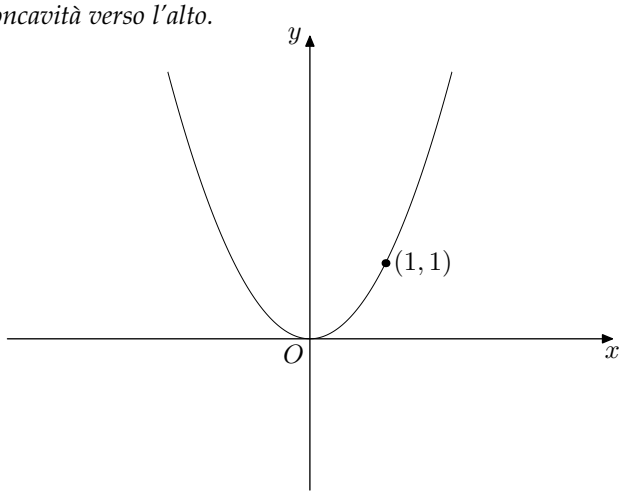
Esempio 8.2.1. *Se* $a = 1$, $b = 0$, $c = 0$, $d = 0$, $e = -1$, $f = 0$ *l'equazione (1.1) diventa*

$$x^2 - y = 0$$

cioè

$$y = x^2$$

nella quale è facile riconoscere l'equazione della parabola con vertice nell'origine O *e concavità verso l'alto.*



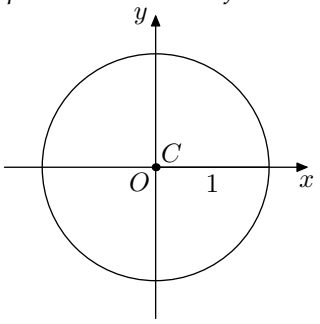
Esempio 8.2.2. *se* $a = 1$, $b = 1$, $c = 0$, $d = 0$, $e = 0$, $f = -1$ *l'equazione (1.1) diventa*

$$x^2 + y^2 - 1 = 0$$

cioè

$$x^2 + y^2 = 1$$

l'equazione della circonferenza con centro nell'origine O *e raggio* $r = 1$.



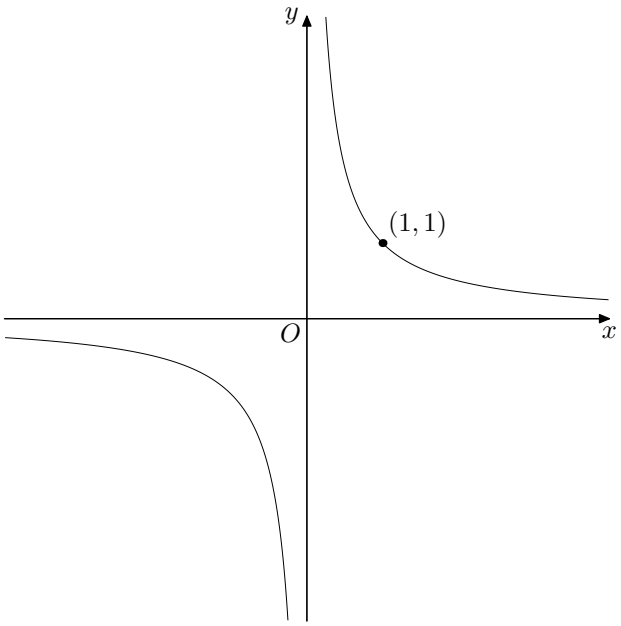
Esempio 8.2.3. *se* $a = 0$, $b = 0$, $c = 1$, $d = 0$, $e = 0$, $f = -1$ *l'equazione (1.1) diventa*

$$x * y - 1 = 0$$

o anche

$$x * y = 1$$

cioè l'equazione dell'iperbole equilatera riferita agli asintoti.



Ogni equazione come la (1.1) può essere scritta in forma matriciale nel seguente modo:

$$X^t.M.X = 0$$

dove:

- il simbolo . indica il prodotto *righe per colonne* tra matrici (si veda il capitolo sulle Matrici)
- X è la matrice

$$\begin{pmatrix} 1 \\ x \\ y \end{pmatrix}$$

- X^t è la matrice

$$\begin{pmatrix} 1 & x & y \end{pmatrix}$$

  cioè la matrice trasposta della matrice X;

  N.B. Si ricorda che la matrice trasposta è quella matrice che si ottiene scambiando le righe con le colonne di quella data; ad esempio:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}^t = \begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix}$$

- M è una matrice di ordine 3 (cioè quadrata con 3 righe e 3 colonne) contenente i coefficienti a, b, c, d, e, f dell'equazione (1.1) disposti nel seguente modo:

$$\begin{pmatrix} f & \frac{d}{2} & \frac{e}{2} \\ \frac{d}{2} & a & \frac{c}{2} \\ \frac{e}{2} & \frac{c}{2} & b \end{pmatrix}$$

- con 0 a secondo membro si intende ovviamente la matrice nulla di dimensioni 1x1.

Quindi si può scrivere:

$$\begin{pmatrix} 1 & x & y \end{pmatrix} \cdot \begin{pmatrix} f & \frac{d}{2} & \frac{e}{2} \\ \frac{d}{2} & a & \frac{c}{2} \\ \frac{e}{2} & \frac{c}{2} & b \end{pmatrix} \cdot \begin{pmatrix} 1 \\ x \\ y \end{pmatrix} = 0$$

Eseguendo il prodotto *righe per colonne* iniziando da quello della seconda matrice per la terza si ottiene:

$$\begin{pmatrix} 1 & x & y \end{pmatrix} \cdot \begin{pmatrix} f + \frac{d}{2} * x + \frac{e}{2} * y \\ \frac{d}{2} + a * x + \frac{c}{2} * y \\ \frac{e}{2} + \frac{c}{2} * x + b * y \end{pmatrix} = 0$$

E successivamente:

$$\begin{pmatrix} f + \frac{d}{2} * x + \frac{e}{2} * y + \frac{d}{2} * x + a * x^2 + \frac{c}{2} * x * y + \frac{e}{2} * y + \frac{c}{2} * x * y + b * y^2 \end{pmatrix} = 0$$

Riordinando i termini del polinomio contenuto nella matrice risultato e poi eguagliando tale elemento a zero, cioè l'unico elemento della matrice nulla che si trova a secondo membro dell'equazione matriciale, si ottiene:

$$ax^2 + by^2 + (\frac{c}{2} + \frac{c}{2})xy + (\frac{d}{2} + \frac{d}{2})x + (\frac{e}{2} + \frac{e}{2})y + f = 0$$

cioè proprio l'equazione (1.1).

E' facile riconoscere che la matrice M associata ad una conica ha una caratteristica particolare: è una matrice simmetrica.

N.B. Si ricorda che una matrice quadrata di ordine n in cui sia detto $a_{i,j}$ il suo generico elemento, risulta simmatrica se e solo se $a_{i,j} = a_{j,i} \forall i = 1, ..., n, j = 1, ..., n$.

Si può quindi affermare che una qualunque matrice simmetrica possa essere matrice di una conica?
(Si lascia al lettore tale dimostrazione.)
E' certo che una matrice di ordine 3 non simmetrica non può certamente rappresentare la matrice di una conica.

8.3 CLASSIFICAZIONE DI CONICHE

Le coniche vengono classificate secondo tre tipologie: *iperboliche*, *paraboliche*, *ellittiche*.

La classe di appartenenza di una conica è legata ai coefficienti dei termini di secondo grado dell'equazione (1.1), i quali sono collocati nella matrice M della conica precisamente nel minore complementare dell'elemento di indici 1,1, cioè la seguente matrice di ordine 2:

$$M_{1,1} = \begin{pmatrix} a & \frac{c}{2} \\ \frac{c}{2} & b \end{pmatrix}$$

Per ulteriori informazioni sui minori e sul calcolo del determinante di una matrice si veda il capitolo sulle Matrici.

La classe di appartenenza di una conica viene individuata quindi come segue:

$$\det\begin{pmatrix} a & \frac{c}{2} \\ \frac{c}{2} & b \end{pmatrix} = a*b-(\frac{c}{2})^2 \quad \begin{matrix} <0 & \text{tipo iperbolico} \\ =0 & \text{tipo parabolico} \\ >0 & \text{tipo ellittico} \end{matrix}$$

N.B. Si ricorda che le circonferenze rientrano nelle coniche di tipo ellittico.

Esempio 8.3.1. *In riferimento all'esempio 1.2.1:*
L'equazione della parabola $y = x^2$ *è associata alla matrice*

$$\begin{pmatrix} 0 & 0 & -\frac{1}{2} \\ 0 & 1 & 0 \\ -\frac{1}{2} & 0 & 0 \end{pmatrix}$$

Il determinante del minore complemetare $M_{1,1}$ *vale:* $1*0-0*0=0$*; essendo nullo vi è la conferma che la conica è di tipo parabolico.*

Esempio 8.3.2. *In riferimento all'esempio 1.2.2:*
L'equazione della circonferenza $x^2+y^2=1$ *è associata alla matrice*

$$\begin{pmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Il determinante del minore complemetare $M_{1,1}$ *vale:* $1*1-0*0=1$*; essendo positivo vi è la conferma che la conica è di tipo ellittico.*

Esempio 8.3.3. *In riferimento all'esempio 1.2.3:*
L'equazione dell'iperbole equilatera $x*y=1$ *è associata alla matrice*

$$\begin{pmatrix} -1 & 0 & 0 \\ 0 & 0 & \frac{1}{2} \\ 0 & \frac{1}{2} & 0 \end{pmatrix}$$

Il determinante del minore complemetare $M_{1,1}$ *vale:* $0*0-\frac{1}{2}*\frac{1}{2}=-\frac{1}{4}$*; essendo negativo vi è la conferma che la conica è di tipo iperbolico.*

Per ciascuna tipologia di conica è possibile individuare la cosiddetta *conica base* e di seguito vengono elencate le loro equazioni:

tipo parabolico	$y=x^2$	parabola base
tipo ellittico	$x^2+y^2=1$	circonferenza base
tipo iperbolico	$x^2-y^2=1$	iperbole base

8.4 MATRICI DI TRASFORMAZIONE

Anche le trasformazioni geometriche Traslazione, Omotetia e Rotazione possono essere rappresentate in forma matriciale mediante matrici di ordine 3.

Siano T la matrice di una trasfomazione geometrica e M la matrice di una conica; la matrice M' della conica trasformata secondo la trasformazione T si ottiene mediante la seguente relazione matriciale:

$$M' = T^t.M.T$$

Di seguito vengono presentate le matrici di alcune tra le trasformazioni più usate:

Traslazione:

$$\begin{pmatrix} 1 & 0 & 0 \\ -a & 1 & 0 \\ -b & 0 & 1 \end{pmatrix}$$

dove (a, b) è il vettore di traslazione:

- a è la componente orizzontale di traslazione
  - se $a < 0$ la traslazione avviene verso sinistra
  - se $a > 0$ la traslazione avviene verso destra
- b è la componente verticale di traslazione
  - se $b < 0$ la traslazione avviene verso il basso
  - se $b > 0$ la traslazione avviene verso l'alto

Cambio di scala:

$$\begin{pmatrix} 1 & 0 & 0 \\ 1 & \frac{1}{h} & 0 \\ 1 & 0 & \frac{1}{k} \end{pmatrix}$$

dove:

- $h \in \mathbb{R}^*$ è il rapporto di cambio di scala orizzontale, cioè applicato alle ascisse:
  - se $|h| < 1$ la conica subisce una contrazione in senso orizzontale
  - se $|h| > 1$ la conica subisce una dilatazione in senso orizzontale

  inoltre, se $h < 0$ viene applicata anche una simmetria rispetto all'asse y;
- $k \in \mathbb{R}^*$ è il rapporto di cambio di scala verticale, cioè applicato alle ordinate:
  - se $|k| < 1$ la conica subisce una contrazione in senso verticale
  - se $|k| > 1$ la conica subisce una dilatazione in senso verticale

  inoltre, se $k < 0$ viene applicata anche una simmetria rispetto all'asse x.

Rotazione:

$$\begin{pmatrix} 1 & 0 & 0 \\ 1 & \cos\alpha & sen\alpha \\ 1 & -sen\alpha & \cos\alpha \end{pmatrix}$$

dove α è l'angolo di rotazione della conica:

- se $\alpha < 0$ la rotazione avviene in senso orario

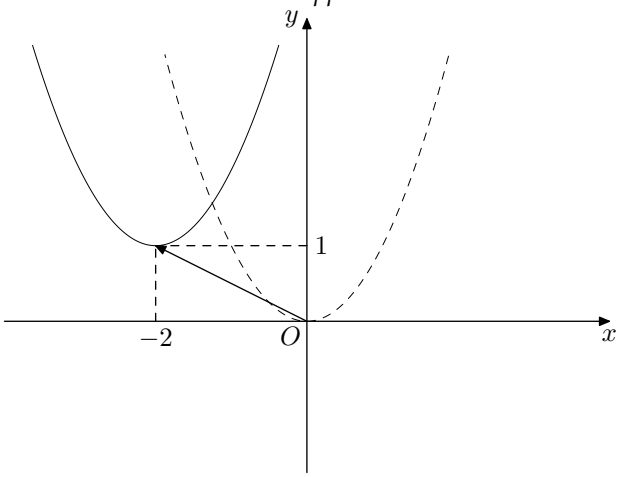
- se $\alpha > 0$ la rotazione avviene in senso antiorario

Esempio 8.4.1. *Applicare alla parabola di equazione* $y = x^2$ *una traslazione di vettore (-2,1).*

La matrice M' *della parabola trasformata viene calcolata nel seguente modo:*

$$\begin{pmatrix} 1 & 1 & -2 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 0 & 0 & -\frac{1}{2} \\ 0 & 1 & 0 \\ -\frac{1}{2} & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ -2 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 3 & 1 & -\frac{1}{2} \\ 1 & 1 & 0 \\ -\frac{1}{2} & 0 & 0 \end{pmatrix}$$

La matrice risultante è associata all'equazione $x^2 + 2x - y + 3 = 0$*, da cui* $y - 2 = (x+1)^2$*, equazione nella quale, come si evince anche dal seguente grafico, viene messa ben in evidenza la traslazione applicata:*

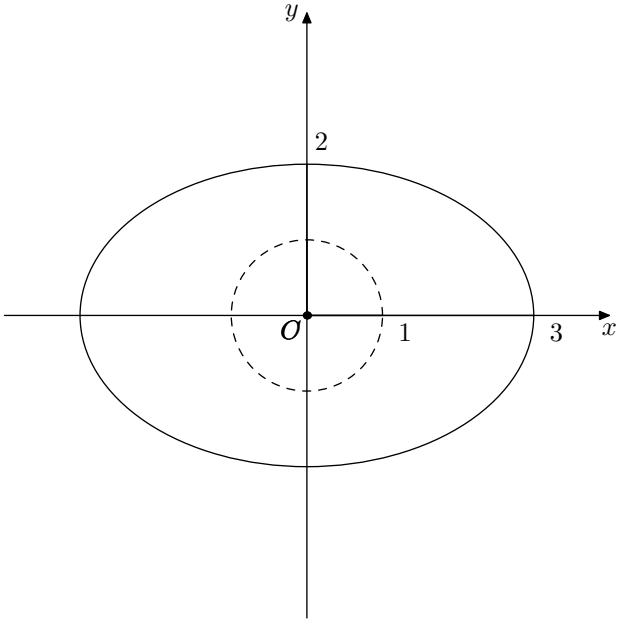


Esempio 8.4.2. *Applicare alla circonferenza di equazione* $x^2 + y^2 = 1$ *un'cambio di scala di rapporto 3 su x e 2 su y.*

La matrice M' *della circonferenza trasformata viene calcolata nel seguente modo:*

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & \frac{1}{3} & 0 \\ 0 & 0 & \frac{1}{2} \end{pmatrix} \cdot \begin{pmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 1 & \frac{1}{3} & 0 \\ 0 & 0 & \frac{1}{2} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \frac{1}{9} & 0 \\ 0 & 0 & \frac{1}{4} \end{pmatrix}$$

La matrice risultante è associata all'equazione $\frac{x^2}{9} + \frac{y^2}{4} - 1 = 0$*, cioè l'equazione della seguente ellisse avente semiasse orizzontale pari a 3 e semiasse verticale pari a 2:*



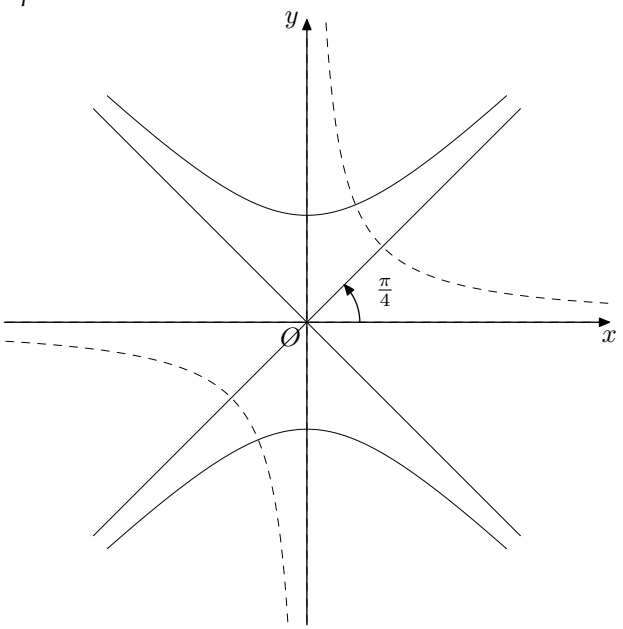
Esempio 8.4.3. *Applicare all'iperbole equilatera riferita ai propri asintoti di equazione* $x * y = 1$, *o anche* $x * y - 1 = 0$, *una rotazione di angolo* $\alpha = \frac{\pi}{4}$.

La matrice M' *dell'iperbole trasformata viene calcolata nel seguente modo:*

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\frac{\pi}{4} & -\text{sen}\frac{\pi}{4} \\ 0 & \text{sen}\frac{\pi}{4} & \cos\frac{\pi}{4} \end{pmatrix} \cdot \begin{pmatrix} -1 & 0 & 0 \\ 0 & 0 & \frac{1}{2} \\ 0 & \frac{1}{2} & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\frac{\pi}{4} & \text{sen}\frac{\pi}{4} \\ 0 & -\text{sen}\frac{\pi}{4} & \cos\frac{\pi}{4} \end{pmatrix} =$$

$$\begin{pmatrix} -1 & 0 & 0 \\ 0 & -\frac{1}{2} & 0 \\ 0 & 0 & \frac{1}{2} \end{pmatrix}$$

La matrice risultante è associata all'equazione $-x^2 + y^2 - 1 = 0$, *o meglio* $x^2 - y^2 = -1$, *cioè l'equazione della seguente iperbole equilatera avente le rette* $y = \pm x$ *come asintoti obliqui:*



8.5 LE CONICHE MATRICIALI IN LABORATORIO

Per eseguire esercitazioni di laboratorio mediante l'applicativo Maxima in applicazione delle trasformazioni geometriche alle coniche in forma matriciale si suggerisce di costruire un archivio contenente le funzioni di base di seguito descritte; esse verranno poi applicate per realizzare particolari esercizi.

Tale archivio potrà essere costruito utilizzando un qualsiasi text editor, ma con l'accortezza di salvare il file col formato txt (testo), e sarà costituito dall'elenco delle definizioni delle funzioni, ciascuna scritta in una riga e chiusa dal simbolo $ (per maggiori dettagli sull'uso di tale simbolo si veda *Parte IV - Programmazione - L'ambiente di programmazione funzionale Maxima*, senza nessun altro simbolo o commento.

Sarà poi sufficiente aprire il suddetto file e copiare tutte le funzioni in esso contenute nella finestra di input delle espressioni di una sessione Maxima per poter disporre di queste funzioni come fossero predefinite.

Le funzioni fondamentali sono:

- `matrice(a,b,c,d,e,f):=matrix([f,d/2,e/2],[d/2,a,c/2],[e/2,c/2,b])$`

  : dati i coefficienti a, b, c, d, e, f dell'equazione della conica in forma normale viene prodotta la matrice associata;

- `equazione(m):=m[2,2]*x^2+m[3,3]*y^2+2*m[2,3]*x*y+2*m[1,2]*x+2*m[1,3]*y+m[1,1]=0$`

  : data la matrice m della conica viene prodotta l'equazione associata in forma normale;

- `tipo_conica(m):= ...`

  funzione che data la matrice m della conica individua il suo tipo;

- `trasforma(m,t):= transpose(t).m.t`

  : data la matrice m della conica e la matrice t di trasformazione viene restituita la matrice della conica trasformata;

- `trasla(a,b):=matrix([1,0,0],[-a,1,0],[-b,0,1])$`

  : date le componenti a e b del vettore di traslazione, rispettivamente orizzontale e verticale, viene restituita la matrice di traslazione;

- `cambioscala(h,k):=matrix([1,0,0],[0,1/h,0],[0,0,1/k])$`

  : dati i due rapporti di cambio di scala, da applicare rispettivamente alle ascisse e alle ordinate, viene restituita la matrice di cambio di scala;

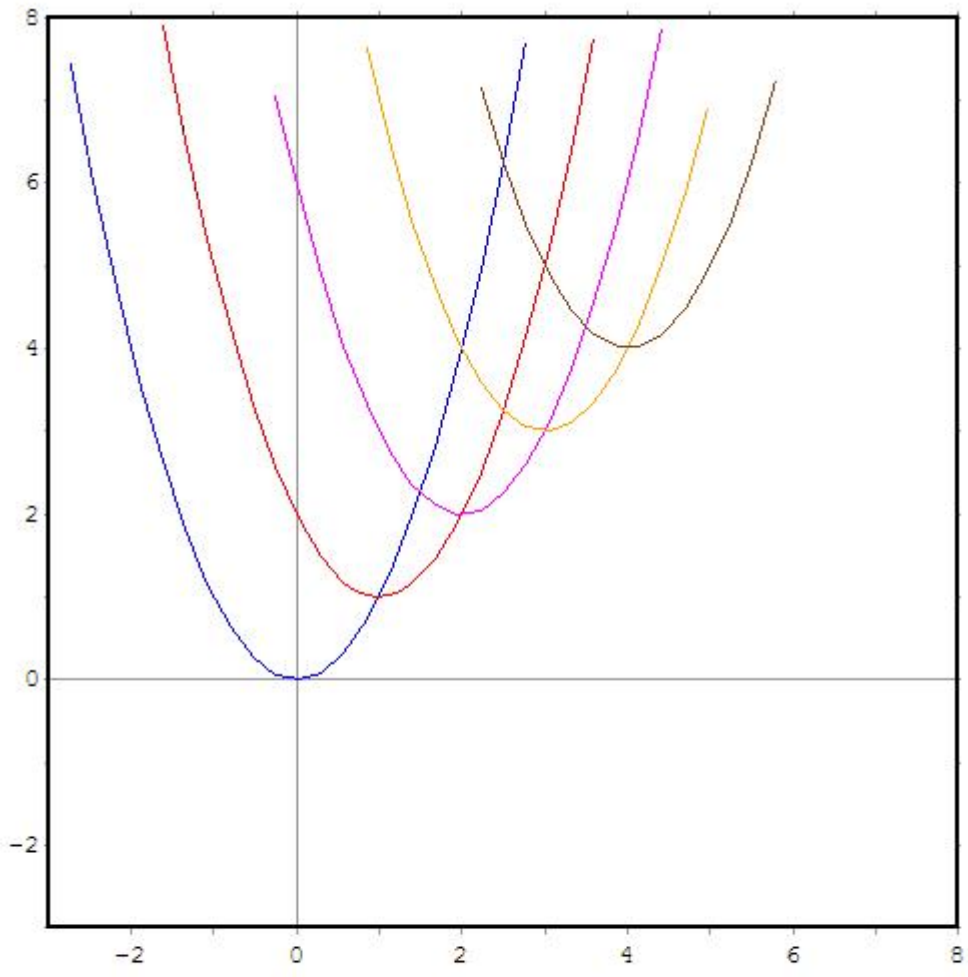
- `rotazione(\alpha):=matrix([1,0,0],[0,cos(alfa),sin(alfa)],[0,-sin(alfa),cos(alfa)])$`

  : data l'ampiezza in radianti dell'angolo di rotazione viene restituita la matrice di rotazione.

Verranno proposti di seguito alcuni esercizi nei quali la difficoltà di esecuzione è via via crescente.

La fantasia del lettore sarà essere comunque la miglior consigliera.

Esempio 8.5.1. *Rappresentare graficamente n parabole con vertice lungo la prima bisettrice a partire dalla parabola base; ad esempio, per $n = 5$:*



La funzione Maxima che permette di realizzare questo grafico è la seguente:

```
parab(n):=
makelist(rhs(solve(equazione(trasforma(matrice(1,0,0,0,-1,0),trasla(i,i))),y)[1]),i,
       0,
       n-1
       );
```

Proviamo ad analizzarla partendo dal cuore della funzione e cioè:

```
equazione(trasforma(matrice(-1,0,0,0,1,0),trasla(i,i)))
```

In questa frase viene richiamata la funzione matrice che, come dice il nome stesso, produce la matrice della parabola base di equazione $y - x^2 = 0$ *(si ricorda che i sei parametri sono i coefficienti dell'equazione della conica in forma normale e quindi laddove un termine non compare nell'equazione significa che il suo coefficiente è nullo). Inoltre viene richiamata la funzione che produce la matrice di traslazione da applicare alla parabola base, nel nostro caso trasla(i,i) sta ad indicare che il vettore di traslazione da applicare ha le componenti orizzontale e verticale uguali e quindi la parabola traslata avrà il vertice certamente sulla prima bisettrice. La funzione trasforma restituisce la matrice della parabola trasformata secondo la traslazione indicata e la funzione equazione restituisce l'equazione della stessa parabola.*

Le chiamate alle funzioni predefinite di Maxima solve ed rhs sono legate al successivo uso della funzione predefinita plot2d (si ricorda brevemente che per la rappresentazione grafica è necessario passare alla funzione plot2d solo il secondo membro dell'equazione in forma esplicita della funzione da disegnare; per maggiori dettagli vedere Parte IV - Programmazione - L'ambiente di programmazione funzionale Maxima).

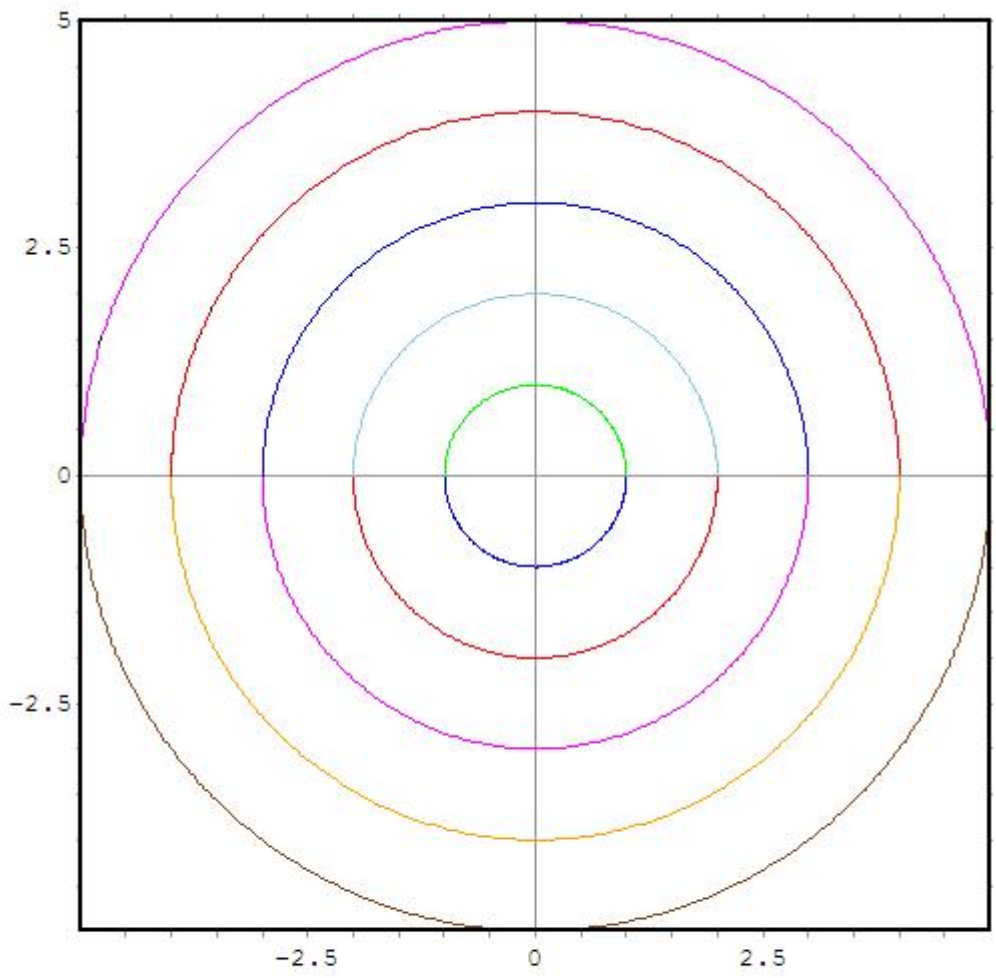
Ovviamente la chiamata alla funzione predefinita makelist permette di costruire la lista contente il numero desiderato di parabole traslate.

Il grafico mostrato si potrà quindi ottenere conla seguente chiamata:

```
plot2d(parab(5),[x,-3,8],[y,-3,8],[plot_format,openmath]);
```

dove range x e range y sono stati scelti per una rappresentazione ottimale delle parabole e l'opzione del formato grafico openmath è necessaria per ottenere il grafico in una finestra separata.

Esempio 8.5.2. *Rappresentare graficamente n circonferenze concentriche a partire dalla circonferenza base fino alla circonferenza di raggio n; ad esempio, per $n = 5$:*



N.B. Si osservi che per rappresentare graficamente con Maxima una circonferenza, è necessario scomporla in due semicirconferenze che, ovviamente, verranno rappresentate graficamente con due diversi colori.

Esempio 8.5.3.

Esempio 8.5.4.

Esempio 8.5.5.

Parte VI

CONTRIBUTI

COLOPHON

Questo lavoro è stato realizzato con LaTeX 2ε usando una rielaborazione dello stile ClassicThesis, di André Miede, ispirato al lavoro di Robert Bringhurst *Gli Elementi dello Stile Tipografico* [1992]. Lo stile è disponibile su CTAN.

Il lavoro è composto con la famiglia di font Palatino, di Hermann Zapf. Le formule matematiche sono state composte con i font AMS Euler, di Hermann Zapf e Donald Knuth. Il font a larghezza fissa è il Bera Mono, originariamente sviluppato da Bitstream Inc. come *Bitstream Vera*. I font senza grazie sono gli Iwona, di Janusz M. Nowacki.

Versione [09/2008.1][S-All]

CONTRIBUTI E LICENZA

Erica Boatto	Algebra - Insiemi
Beniamino Bortelli	Grafici
Roberto Carrer	Numeri - Funzioni - Coordinatore progetto
Morena De Poli	Laboratorio matematica
Piero Fantuzzi	Algebra - Insiemi
Carmen Granzotto	Funzioni
Franca Gressini	Funzioni
Beatrice Hittahler	Funzioni trascendenti - Geometria analitica
Lucia Perissinotto	Funzioni trascendenti - Geometria analitica
Pietro Sinico	Geometria I

La presente opera è distribuita secondo le attribuzioni della Creative Commons.
La versione corrente è la  .
In particolare chi vuole redistribuire in qualsiasi modo l'opera, deve garantire la presenza della prima di copertina e della intera Parte IV Contributi composta dai paragrafi: Colophon e Contributi e licenza.

Settembre 2008

Dipartimento di Matematica
ITIS V.Volterra
San Donà di Piave