Importazione ed esportazione

- **Introduzione**

Abbiamo visto le potenzialità di *Mathematica* per quanto riguarda il calcolo simbolico e numerico. Tuttavia, nonostante abbiamo imparato a fare un sacco di cose, ci siamo limitati a lavorare sempre all'interno del programma. Molte volte, invece (poche, a dire il vero, se siamo studenti), capita di dover effettuare delle elaborazioni su dei dati acquisiti esternamente al programma, oppure vogliamo esportare i dati in modo che possano essere utilizzati da altri programmi. Lo scambio dei dati fra programmi è un concetto importante da imparare, specie se bisogna lavorare molto al calcolatore per elaborare e sviluppare le opportune ricerche. Per esempio, si usa un programma in C ideato da un'altra persona, per effettuare una simulazione, salvando i risultati in un file, e dopo si vogliono importare questi dati per vedere se effettivamente sono come ce li aspetiamo. oppure, magari, potremmo usare i dati derivati da un oscilloscopio campionatore per effettuare analisi che l'oscilloscopio non è in grado di eseguire, per esempio se non è in grado di effettuare la trasformata di Fourier dei campioni (anche se probabilmente riesce a farlo se è anche solo sufficiente come strumento, ma parlo solo di esempi). Le applicazioni sono tutte importanti, e *Mathematica* le gestisce praticamente tutte, da entrambi i sensi. Vediamo, quindi, cosa possiamo fare per elaborare con *Mathematica* i nostri dati.

- **Importazione**

Mathematica è in grado di importare una quantità di file di diverso formato: file di testo, immagini e quant'altro. Esiste una variabile di sistema che elenca tutti i possibili formati di file che si possono importare:

```
$ImportFormats
```

```
{AIFF, APS, AU, Bit, BMP, Byte, Character16, Character8, Complex128,
 Complex256, Complex64, CSV, DICOM, DIF, Dump, DXF, EPSTIFF,
 Expression, ExpressionML, FITS, GIF, HarwellBoeing, HDF, HDF5,
 Integer128, Integer16, Integer32, Integer64, Integer8, JPEG, Lines,
 List, MAT, MathML, MGF, MPS, MTX, NB, NotebookML, PBM, PCX, PGM,
 PNG, PNM, PPM, Real128, Real32, Real64, SDTS, SND, STL, String,
 SymbolicXML, Table, TerminatedString, Text, TIFF, TSV, UnicodeText,
 UnsignedInteger128, UnsignedInteger16, UnsignedInteger32,
 UnsignedInteger64, UnsignedInteger8, WAV, Words, XBitmap, XLS, XML}
```

Fra i molti che non conosco neanch'io, potete notare i formati più conosciuti, come BMP, JPEG, WAV, Text, e, importante soprattutto per la compatibilità con i programmi più moderni, XML. Questa lista ovviamente dipende dalla versione del programma posseduta (la mia è la 5.1), ma la maggior parte dei file più comuni rimane invariata.

Il comando usato per importare dei dati è il seguente:

`Import["`*file*`", "Table"]`	importa una tavola di dati da un file

La seconda opzione dice in particolar modo al programma in quale modo sono organizzati i file nel file. Per i file di testo le opzioni più comuni sono:

- **CSV**: valori tabulari separati da una virgola.
- **Lines**: linee di testo.
- **List**: linee che sono formate più valori.
- **Table**: matrice bidimensionale di numeri oppure stringhe.
- **Text**: strnga di caratteri ordinari.
- **TSV**: tabella di testo con valori separati da tabulazione.
- **UnicodeText**: stringhe di caratteri Unicode di 16 bit.
- **Words**: parole separate da spazi o da ritorni a capo.
- **XLS**: formato di foglio di calcolo di Excel.

Supponiamo di voler importare una lista di dati:

Come potete vedere, l'importazione di un'immagine è cosa alquanto semplice e rapida

```
dati = Import["D:\\Documenti\\dati.txt", "List"]
```

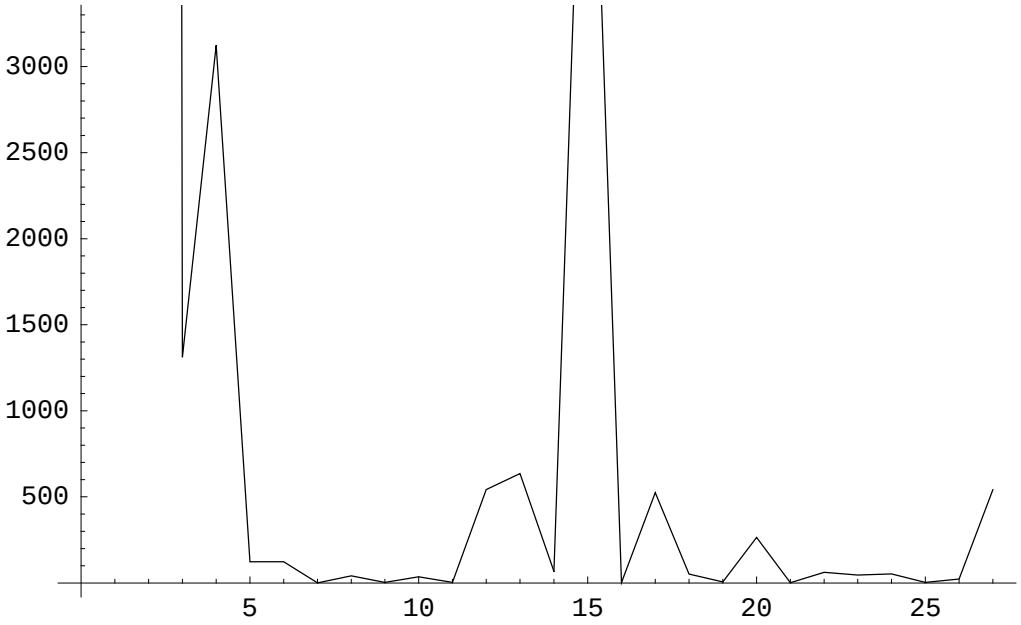
```
{12312, 124314, 1312, 3123, 123, 123, 1, 41, 4, 35, 4, 543,
 635, 67, 5735, 3, 524, 52, 6, 264, 2, 62, 46, 53, 4, 23, 542}
```

Prima di continuare, notate come sono definiti i percorsi di file in *Mathematica*: infatti, usa una notazione propria, che è indipendente dalla macchina e dal sistema operativo che si sta usando. Per poter importare un file, è necessario il percorso assoluto, a meno che non si trovi nella cartella del programma, cosa che fra l'altro vi sconsiglio di fare. Meglio tenere sempre separati programmi e dati, come chiunque in vita sua ha provato almeno una volta a fare un backup sa bene. Per semplificarvi la vita, potete usare la voce di menù del programma Input->Get File Path, che vi scrive

direttamente il percorso quando scegliete il file da una finestra di Esplora Risorse. Detto questo, possiamo andare avanti...

Come possiamo vedere, abbiamo appena importato il nostro file, con i valori contenuti in esso raggruppati nella lista:

```
ListPlot[dati, PlotJoined → True]
```




```
- Graphics -
```

Avendo la lista, adesso possiamo trattarla esattamente come abbiamo imparato fino ad adesso possiamo elaborarla, estrarre valori, farne la trasformata di Fourier... Tutto quello che ci serve, insomma. Possiamo anche importare formati grafici:

```
cane = Import["D:\\Documenti\\Immagini\\dog02.jpg"]
```

```
- Graphics -
```

E possiamo visualizzarlo esattamente come se fosse una primitiva grafica di tipo Raster:

```
Show[cane]
```

```
- Graphics -
```

Tuttavia, sebbene sia abbastanza semplice importare le immagini, estrarne i dati per le elaborazioni richede un passaggio in più.

```
evanescence = Import[
   "D:\\Documenti\\Immagini\\EvanescenceFallenGroupWallpaper.jpg"];
```

```
Show[evanescence]
```


```
- Graphics -
```

Possiamo vedere come sia stata importata l'immagine nel suo complesso. Tuttavia, possiamo notare che, nel caso di immagini, importiamo oggetti grafici, che possono essere usati solo come tali. Invece, può capitare che un'immagine ci serva come matrice, nel senso che ogni pixel contiene valori numerici che ci interessano per elaborazioni numeriche. In questo caso, ci serve la lista che *Mathematica* usa per rappresentare l'immagine raster.

```
Shallow[InputForm[evanescence]]
```

```
Graphics[Raster[<< 4 >>], Rule[<< 2 >>], Rule[<< 2 >>], Rule[<< 2 >>]]
```

Il comando Shallow permette di rappresentare in forma abbreviata il contenuto di un comando o di una variabile. Vedremo in una delle sezioni successive più in dettaglio questo comando. Per ora ci basti sapere che serve per visualizzare brevemente il contenuto di qualsiasi cosa esista in *Mathematica*.

Possiamo vedere che la variabile evanescence che rappresenta il formato grafico è formato dal comando Raster, più tre regole. sappiamo che il comando Raster richiede una lista di dati da visualizzare, per cui la lista che ci interessa si trova proprio lì dentro. Possiamo estrarre direttamente la lista dal formato grafico, nel seguente modo:

```
dati = evanescence[[1, 1]];
```

```
Short[dati, 4]
```

```
{ <<1>> }
```

Con il comando Short vediamo una rappresentazione riassunta del contenuto di qualcosa. L'ho dovuta usare perchè i dati di un'immagine sono numerosi, ed avrebbero riempito decine di pagine. Si può vedere che, per un'immagine, i dati sono rappresentati da una matrice, i cui elementi sono a loro volta delle liste di tre numeri, rappresentanti rispettivamente i valori Rosso, Verde e Blu del pixel. Una volta ottenuta la lista, possiamo elaborarla come più ci piace.

Naturalmente, possiamo effettuare tutte le operazioni di elaborazione delle immagini che vogliamo. Soltanto che non è che conosca bene cose come il filtro di Kalman oppure algoritmi di convoluzione... accontentatevi di questo esempio stupido ed anzi, se sapete suggerirmi qualche algoritmo di elaborazione dei segnali particolarmente simpatico, sarò bel lieto di usarlo...

In questo caso, andiamo ad usare una matrice di convoluzione per trasformare la nostra immagine da matrice di tre valori, a matrice di singoli valori, applicando un certo effetto, quindi trasformata in scala di grigi. Andiamo a definire il nostro tensore:

```
kernel = {
   {{-1, -1, 0}, {1, 0, 1}, {1, -1, 1}},
   {{1, 1, 1}, {-2, -5, -1}, {1, 1, 1}},
   {{1, 1, 1}, {1, 1, 1}, {-1, 0, -1}}
  };
```

Una volta creata la nostra matrice di convouzione, andiamo ad applicarla alla nostra immagine, o meglio, alla lista di valori ad essa correlata (dato che la sola importazione comporta la creazione non della lista, ma di un elemento grafico Raster): dobbiamo elaborare ogni singolo canale di colore

```
datielaborati = ListConvolve[kernel, dati];
```

```
datielaborati = datielaborati / Max[datielaborati];
```

Quindi adesso, dopo aver elaborato e normalizzato la nostra lista di elementi dell'immagine, possiamo andare a mostrare direttamente il risultato:

```
Show[
 Graphics[
  Raster[
   datielaborati
  ]
 ],
 AspectRatio → Automatic
]
```


```
- Graphics -
```

Ho effettuato una scrittura annidata per farvi capire meglio dove stanno i vari elementi: prima di tutto ho usato il comando Show per rappresentare l'oggetto grafico rappresentato da Raster. Quest'ultimo, come dovreste sapere, deve contenere la matrice necessaria per la visualizzazione; inoltre, ho anche specificato la funzione colore da utilizzare, che stavolta è GrayLevel, in quanto il risultato della convoluzione è data da una matrice i cui elementi sono singoli valori, invece che lista di tre come per il caso dell'immagine originale... A questo punto, ho impostato per Show il rapporto originale dell'immagine, e il risultato è bell'e visualizzato.

Anche se vuole essere soltanto un mero esempio, mi scuso ancora per non aver trovato un filtro più originale ed utile ma, come vi dicevo poco fa, non ho mai avuto a che fare con l'elaborazione delle immagini, ed anzi, penso che come prima volta non sia andata poi tanto male... Tanto dovete essere voi a definire le vostre operazioni da eseguire sui dati, non io, ed inventarsi di sana pianta qualcosa (in dieci minuti, fra l'altro... mica ho tutta la vita da spendere!!!) non è cosa facile.

A proposito, come avrete capito io adoro quest'album!!! Se qualcuno che legge queste righe, in segno di riconoscimento volesse presentarmi la cantante...

- **Esportazione**

Dati

E adesso guardiamo l'altra faccia della medaglia, ovvero l'esportazione dei dati. Come sicuramente sapete, avere un programma che fa cose bellissime e fighissime, e poi non possiamo farle vedere a nessuno, è cosa alquanto inutile. Basti pensare a quanto può essere utile scrivere al computer una tesina, se poi il programma non è in grado di stamparla... Esportare i dati è altrettanto importante che importarli, perchè ci permette di andarli ad usare dove più ci serve. Può capitare infatti che, per esempio, preferiate redigere le vostre relazioni in Word, e vi piacerebbe avere i risultati dei dati oppure delle immagini grafiche all'interno delle sue pagine pacioccose. Ma perchè non usate Open-Office, dico io...

Comunque, vediamo che, effettivamente, esportare i dati è altrettanto semplice:

`Export["`*file*`", `*list*`, "Table"]`	esporta i dati contenuti in *list* in un file come tavola di valori
`Export["`*name*`.`*ext*`", `*graphics*`]`	esporta *l'* immagine in graphics con il formato dedotto dall' estensione del file
`Export["`*file*`", `*graphics*`, "`*format*`"]`	esporta la grafica nel formato specificato

Supponiamo di avere la seguente lista:

```
lista = {{1, 2, 4, 5}, {2, 3, 1, 24}, {2, 2, 4, 4}, {5, 6, 3, 2}};
```

Per esportarla nel file di testo esempio.txt, basta scrivere:

```
Export["esempio.txt", lista]
```

```
esempio.txt
```

Se vogliamo vedere il contenuto del file, basta scrivere:

```
!! esempio.txt
```

```
{{1, 2, 4, 5}, {2, 3, 1, 24}, {2, 2, 4, 4}, {5, 6, 3, 2}}
```

Come possiamo vedere, in questo caso la lista è stata esportata proprio come la scrive *Mathematica*, cioè mediante le parentesi graffe. Un formato che evita questo è il seguente:

```
Export["esempio.dat", lista]
```

```
esempio.dat
```

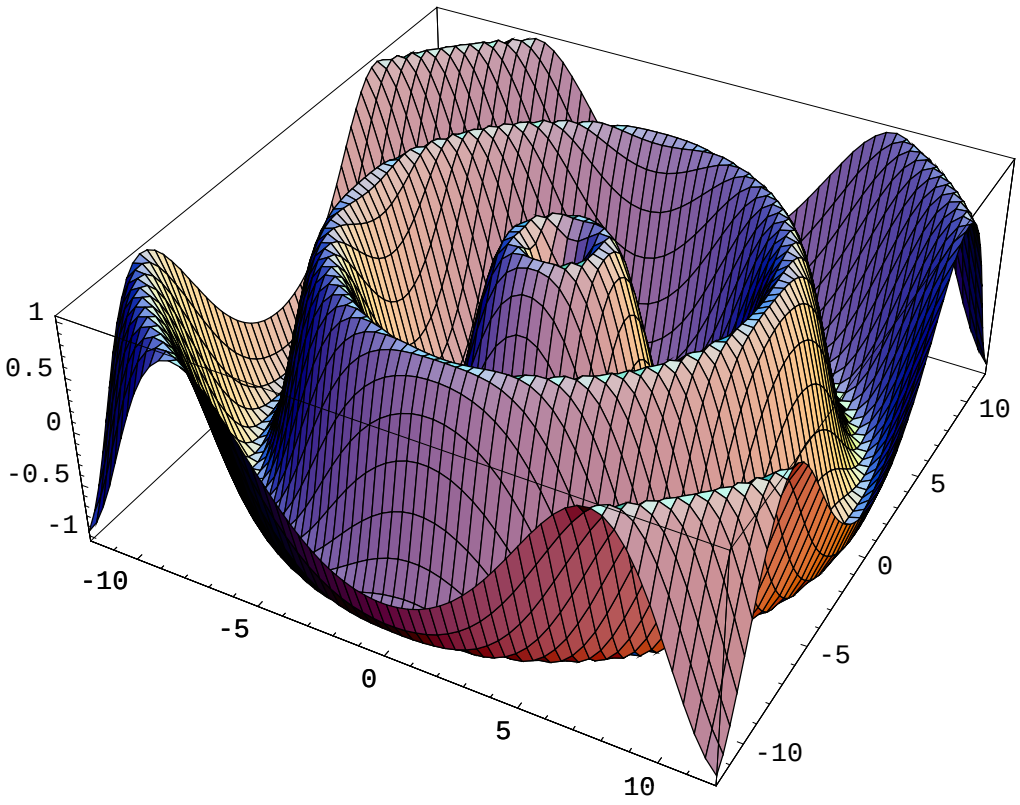
```
!! esempio.dat
```

```
1   2   4   5
2   3   1   24
2   2   4   4
5   6   3   2
```

Come possiamo vedere, stavolta, cambiando l'estensione del file, i dati vengono esportati nella maniera corretta e leggibile da altri programmi come Excel, cioè come matrice bidimensionale separati da tabulazioni.

In maniera analoga possiamo esportare le immagini che creiamo con il programma:

```
grafico = Plot3D[Sin[Sqrt[x^2 + y^2]],
  {x, -12, 12}, {y, -12, 12}, PlotPoints → 60]
```




```
- SurfaceGraphics -
```

```
Export["grafico.bmp", grafico]
```

```
grafico.bmp
```

In questo caso abbiamo esportato l'immagine in formato bitmap. Tuttavia, se avete intenzione di stampare i vostri lavori, e se il programma che usate ve lo consente, io vi consiglio di usare il formato Encapsulated PostScript .eps, perchè è gestito meglio da *Mathematica*, e soprattutto perchè è un formato vettoriale, e quindi indipendente dalla risoluzione. Inoltre, è nativamente supportato da Adobe Acrobat, che vi permette quindi di mantenere le immagini al massimo della qualità con un considerevole risparmio di dimensioni rispetto ai formati raster quali BMP, JPEG anche se compresso e TIFF, tutti comunque supportati dal programma. Le eccezioni sono i grafici a curve di livello e quelli di densità, cioè quelli che capite è difficile da rappresentare vettorialmente, in quanto le dimensioni del file crescerebbero alquanto, a vantaggio dei formati raster. Per farvi capire la differenza, ingrandite le ultime due immagini, quella del grafico, disegnata in eps da Acrobat, e quella degli Evanescence, in formato compresso, e vedere come i pixel si vedano in questo secondo caso, mentre nel grafico vettoriale anche ingrandendo la definizione rimane invariata.

Formule

Mathematica è anche in grado, ovviamente, di esportare le formule create nei suoi notebook. Supponiamo di avere la seguente espressione:

```
formula = Integrate[1 / Sqrt[x^4 - 3], x]
```

$$\frac{\sqrt{3 - x^4}\ \text{EllipticF}\left[\text{ArcSin}\left[\frac{x}{3^{1/4}}\right], -1\right]}{3^{1/4}\ \sqrt{-3 + x^4}}$$

Possiamo esportarla come file eps:

```
Export["formula.eps", ToBoxes[formula]]
```

```
formula.eps
```

Possiamo anche esportare l'espressione in notazione tradizionale, se vogliamo:

```
tradizionale = TraditionalForm[formula]
```

$$\frac{\sqrt{3 - x^4}\ F\left(\sin^{-1}\left(\frac{x}{\sqrt[4]{3}}\right)\middle|-1\right)}{\sqrt[4]{3}\ \sqrt{x^4 - 3}}$$

```
Export["formula.eps", ToBoxes[tradizionale]]
```

```
formula.eps
```

Notate come si debba usare la funzione ToBoxes, mi raccomando.

Questo permette di esportare le formule in formato grafico. Tuttavia, esiste anche un modo mogliore, e cioè usare il formato TEX. Come molti di voi fedeli seguaci saprete sicuramente, TEX e LATEX sono due importantissimi strumenti per la documentazione scientifica, che permettono con degli opportuni comandi in file di testo ASCII di poter creare formule e documentazione di alta qualità e dall'aspetto decisamente professionale.

Mica pensavate che i libri di matematica si facessero col ridicolo Equation Editor di Word, vero????

Per convertire la formula in formato TEX, basta usare l'opportuno comando:

`TeXForm[`*expr*`]`	scrive *expr* in formato TeX

Possiamo fare l'esempio con la formula di sopra:

```
TeXForm[formula]
```

```
\frac{\sqrt{3-x^4} F\left(\left.\sin
   ^{-1}\left(\frac{x}{\sqrt[4]{3}}\right)\r
   ight|-1\right)}{\sqrt[4]{3} \sqrt{x^4-3}}
```

Questo testo è pronto per essere copiato ed incollato nel vostro editor LATEX (che preferisco rispetto a TEX), ed una volta compilato darà il risultato desiderato. Appena vete un poco di tempo, se non lo conoscete, dateci uno sguardo, anche perchè se farete la tesi con questo programma invece che con Word, un punticino in più per la presentazione è assicurato!!!

Naturalmente, è anche possibile fare l'operazione opposta, ovvero convertire in espressioni tipiche di *Mathematica* quelle scritte con LATEX:

```
ToExpression["\\frac{(x+y)^2}{\sqrt{x y}}", TeXForm]
```

$$\frac{(x+y)^2}{\sqrt{x\,y}}$$

Inoltre, è anche possibile convertire unintero Notebook usando il comando di menù File->Save As Special, generando così sia il file tex che tutte le immagini eps necessarie.

Un'altra interessantissimissima (a mio avviso) caratteristica è il poter convertire ed esportare le espressioni anche in formato C (oppure Fortran):

```
CForm[formula]
```

```
(Sqrt(3 - Power(x,4))*
     EllipticF(ArcSin(x/Power(3,0.25)),-1))
    /(Power(3,0.25)*Sqrt(-3 + Power(x,4)))
```

Come potete vedere, l'espressione adesso è pronta per essere copiata ed incollata nel vostro editor C, ammesso che abbiate definito prima le opportune funzioni. Si potrebbe anche usare *Mathematica* stessa, usando dal vostro programma C il suo kernel, ma è una cosa che non ho mai fatto, e che non ho intenzione di affrontare, anche perchè non saprei da dove cominciare. Comunque, vivrete tranquillamente anche senza, fidatevi.

Invece, può essere estremamente utile, in questo caso, il seguente comando:

`Splice["`*file*`.m`*x*`"]`	elabora l'input di *Mathematica* contenuto nel file *file*`.m`*x*, e scrive il risultato in *file*`.`*x*
`Splice["`*infile*`", "`*outfile*`"]`	specifica file di input e di output

Ipotizziamo di dover scrivere un programma di simulazione fisica in C, e che non ci ricordiamo il risultato di un'espressione. In questo caso, potremmo scrivere all'interno dello stesso file un input in formato *Mathematica*, opportunamente definito da degli opportuni indicatori di inizio e fine espressione:

```
......
double x;
double y;
....
y= <* Integrate[1/Sqrt[1+x^3],x] *>;
......
```

Notate come la definizione del comando *Mathematica* sia racchiuso fra <* *>, per permettere al programma un parsing più veloce ed efficiente. Adesso, diamolo in pasto al comando:

```
Splice["C:\\Documents and Settings\\Daniele\\Desktop\\prova.mc"]
```

Il risultato è un file contenente il seguente testo:

```
......
double x;
double y;
....
y= (2*Power(-1,0.16666666666666666)*
```

```
    Sqrt(-(Power(-1,0.16666666666666666)*
       (Power(-1,0.6666666666666666) + x)))*
    Sqrt(1 + Power(-1,0.3333333333333333)*x +
      Power(-1,0.6666666666666666)*Power(x,2))*
    EllipticF(ArcSin(Sqrt(-(Power(-1,0.8333333333333334)*(1 + x)))/
      Power(3,0.25)),Power(-1,0.3333333333333333)))/
  (Power(3,0.25)*Sqrt(1 + Power(x,3)));
.....
```

Come potete vedere, questo comando è una figata pazzesca, e sono convinto che vi aiuterà in più di un'occasione, fidatevi!!! Sempre se avrete a che fare con la programmazione, però, ovvio!!!

- **Packages**

Introduzione

I packages sono, in poche parole, dei file che contengono definizioni di formule, costanti ed altro. Vengono utilizzati quando avete delle funzioni che utilizzate spesso, specialmente in lavori diversi fra di loro (per esempio, preferite usare un vostro set di funzioni personalizzate per eseguire delle operazioni di routine, come importazione da file sempre uguali, oppure funzioni personalizzate che vi servono, come delle equazioni differenziali che state studiando in quel periodo).

Memorizzarle una volta sola ha il vantaggio di non doverle riscrivere ogni volta che ne avete bisogno, risparmiando tempo, ed evitando tra l'altro la possibilità di poter sbagliare le equazioni che scrivete, facendolo una volta sola, ed, ovviamente, assicurandoci che in quell'unica volta siao giuste!!!!!

Diciamo che i packages sono un poco come i file di header del C, contenenti funzioni e quant'altro, già belli pronti all'uso.

Mathematica include già, in fase di installazione, un bel po' di packages, pronti per l'uso, che contengono parecchie funzioni specialistiche. Quando volete utilizzare una particolare funzione, basta caricare il package che la contiene. Se poi utilizzate spesso funzioni contenute in vari packages (per esempio, se usate spesso avanzate funzioni di statistica), potete caricare questi file all'avvio di *Mathematica*, che quindi avrà fin dall'inizio, nella sua pancia, quello che vi serve. Occhio, però, che se utilizzate lo stesso file in un altro computer, dovete ricordarvi che probabilmente i file non saranno pre-caricati, e dovrete caricarli di nuovo a mano.

Quindi, vedremo prima come realizzare i propri packages personali, come salvarli, come caricarli, e come fare in modo che *Mathematica* li carici automaticamente. Dopo faremo una panoramica su vari packages di *Mathematica*. Non avrò nè il tempo nè lo spazio di descriverli tutti, anche perchè di solito variano in numero da varsione a versione di *Mathematica* (io sto usando la 5.1). Se non

trovate qua quello che vi serve, probabilmente lo troverete nell'help di *Mathematica* nella sezione Add-ons.

Salvataggio e caricamento

Caricare un package in *Mathematica* è abbastanza facile, a parte una cosa che, purtroppo, è rognosetta per chi ha la tastiera italiana: un package predefinito si richiama con la seguente sintassi:

```
<< directorypackage`nome package`
```

I simboli che circondano il nome del package non sono i singoli apici della nostra tastiera itailana; sono degli accenti gravi, che possiamo scrivere utilizzando il loro codice unicode 96: per poterlo scrivere dobbiamo quindi battere da tastiera ALT+096, con i numeri del tastierino numerico, non quelli della tastiera, per cui chi possiede un portatile (come me....), deve attivare la simulazione del tastierino numerico. Tutavia basta farci l'abitudine, che si fa dopo qualche volta che si richiama un package. In alternativa, è possiblile copiarlo dalla Mappa Caratteri di Windows, oppure dall'help di *Mathematica*, ma imparare a scriverlo direttamente dalla tastiera sarà meglio... Inoltre, potete al limite scrivere voi stessi un package personalizzato, dove andate a scrivere tutti i package predefiniti che vi servono, senza bisogno di chiamarli, e chiamando solamente il file che avete creato, che sarà senza caratteri strani, e soprattutto breve!!!

Comunque, vediamo dapprima di creare i nostri packages. Supponiamo di volerci definire una funzione personalizzata; aprendo un nuovo file possiamo allora scrivere all'interno la nostra funzione

```
f[x_, y_] := x^4 + x^Cos[3 y x]
```

Una volta creato la funzione, o più probabilente le funzioni che vogliamo caricare nel package, non abbiamo finito: prima di salvare il file, dobbiamo andare a convertire le celle in cui sono contenute le funzioni, trasformandole in celle di inizializzazione. Per fare questo, selezioniamo una cella, cliccando sulla corrispondente linea sulla destra del notebook, e andiamo su Cell->Cell Properties->-Inialization Cell. Vediamo che il sibolo che contraddistingue la cella è cambiato. A questo punto andiamo su File->Save As Special-> Package Format. In questa maniera andremo a creare in un file con estensione .m il package desiderato. Adesso, se volete, cancellate con Clear la definizione della funzione, e caricate il file in questa maniera:

```
<< file
```

Se salvate il file in una posizione precisa, per trovarlo potete utilizzare il comando di menù Input->-Get File Path, permettendovi di selezionare dalla finestra di Explorer il vostro file. Nel mio caso ho:

```
<< "C:\\Documents and Settings\\Daniele\\Desktop\\prova.m"
```

Una volta caricato, notiamo come la f sia direttamente disponibile:

```
f[5, 7]
```

$$625 + 5^{\text{Cos}[105]}$$

Ovviamente, raramente si utilizzeranno packages per poter memorizzare funzioni semplici. Di solito si utilizzano per memorizzare e precaricare funzioni abbastanza complicate, programmi di parecchie pagine, e tutto il resto. Ovviamente anche definizioni semplici, magari se sono parecchie e se dovete caricarle tutte ogni volta che iniziate un lavoro. Il mio consiglio è di non abusarne, andando a memorizzare ogni cosa vi capiti, perchè in questo caso avrete un numero esagerato di packages e non saprete più cosa contengono. Inoltre, createli organizzati, senza andare a mettere funzioni e programmi a caso in un unico package. In questa maniera, ed utilizzando nomi appropriati, sarete in grado di poter caricare ogni volta il package che vi serve.

Se andate ad aprire adesso il file .m, noterete che contiene la definizione della funzione:

```
f[x_, y_] := x^4 + x^Cos[3y x]
```

Quando avrete molte funzioni, potrebbe però essere difficile andarsi a ricordare quello che fanno, anche se si guarda la definizione, se non opportunamente commentata. Tuttavia, possiamo anche inserire nel package, assieme ad una funzione, anche una spiegazione riguardante la sua funzione, che appare quando andiamo ad usare il comando ? sulla funzione, esattamente come per le funzioni predefinite:

```
?Cos
```

```
Cos[z] gives the cosine of z. More…
```

Per questo è utile andare ad utilizzare parecchi commenti quando si vanno a creare questi files, esattamente come le buone regole di programmazione.

Esiste uno standard de facto per il commento dei packages: di solito si inizia così:

```
(*:Mathematica Version:x.x*)
(*:Package Version:y.y*)
(*:Name:Nome`def`*)
(*:Author:Nome Autore*)
(*:URL:sito web del package*)
(*:Summary:*)
(*:History:v1.00 (2003-02-20):
    Written v1.01 (2003-10-18):
     Modified now that Norm is
      built in to v5.Updated context.(c) 2004 Autore*)
(*:Keywords:*)
(*:Requirements:None.*)
(*:Discussion:*)
(*:References:*)
(*:Limitations:None known.*)
```

Ogni riga rappresenta un commento, di conseguenza, tutto quello che avete visto finora è completamente facoltativo... Mettete comunque qualche informazione, specialmente se fate un package serio, e magari volete metterlo su Internet per poter condividerlo assieme agli altri...

Dopo aver scritto 'la presentazione', occorre il comando BeginPackage per cominciare il package vero e proprio:

```
BeginPackage["PersonalPack`Esempio`"]
```

"Esempio" rappresenta il context del package, e rappresenta il modo in cui si evita di sovrascrivere altre definizioni di altri packages. Per intenderci, possiamo utilizzare due files, uno con context "Esempio" e l'altro con context "Mare". Inoltre, può capitare che entrambi questi packages abbiano definito una funzione con lo stesso nome, diciamo f. Allora, per distinguere la f di un package rispetto all'altro, bisogna accodargli il context, per specificare a quale funzione ci riferiamo; se, quindi, vogliamo utilizzare la funzione di Esempio, dobbiamo scrivere:

```
f`Esempio
```

Dove bisogna sempre utilizzare l'accento grave. Potete anche farne a meno, ed utilizzare soltanto il nome principale, utilizzando però l'ultimo accento grave.

Dopo, seguono le descrizioni delle funzioni, che sono di questo tipo:

```
funzione::usage = "f[x,y] Spiegazione della funzione"
```

Quando si creano riche di questo tipo, la stringa viene utilizzata per poter spiegare, tramite il comando ?, il funzionamento della funzione (scusate il gioco di parole...), esattamente come per i comandi predefiniti: per esempio, scritto questo nel nostro package, e ricaricato (dopo aver scritto anche il resto, però...)

```
?f
```

```
f[x,y] Spiegazione della funzione
```

Come potete vedere, abbiamo inserito la spiegazione della funzione.

Dopo averle scritte, possiamo andare a definire le nostre funzioni, esattamente come faremmo all'interno di *Mathematica*. Ricordatevi solamente che tutte le celle devono essere inizializzate, come abbiamo visto poco fa. A questo punto, scriviamo le ultime due righe che finiscono il package:

```
End[]
```

```
EndPackage[]
```

A questo punto, salviamo il file come file .m ed il package è fatto, pronto per essere utilizzato quando vogliamo.

Un'altra caratteristica dei package, sarà familiare a chi conosce il C++: sto parlando delle funzioni pubbliche e private. Quando andiamo a creare le funzioni in questa maniera, tutte quante sono disponibili da *Mathematica* quando si richiama il package. Tuttavia, ci possono essere funzioni e variabili che non servono a noi direttamente, ma servono per valutare le funzioni all'interno del package. Per evitare, allora, che nascondano altri nomi, oppure per evitare che le modifichiamo acccidentalmente, occorre definire privati queste variabili, e funzioni, e questo si può ottenere facilmente inserendo le definizioni di queste funzioni in questo blocco:

```
Begin["`Private`"]
```

```
... ..........
```

```
End[]
```

A questo punto, nessuno più vi toccherà queste funzioni, che saranno invisibili per voi, e non ve ne dovrete preoccupare.

Vediamo di fare un esempio pratico: vogliamo creare un piccolo package che contenga alcune funzioni per i numeri complessi. Vediamo quali sono le funzioni che vogliamo:

- **exptoalg[x]** per convertire un numero complesso dalla forma esponenziale a quella algebrica.
- **algtoexp[x]** converte il numero complesso dalla forma algebrica a quella esponenziale
- **retta[x,y]** calcola l'equazione della retta che, nel piano, passa per i due numeri complessi specificati

La prima funzione sarà la seguente:

```
exptoalg[x_] := ComplexExpand[Re[x]] + I ComplexExpand[Im[x]]
```

Notate come abbia utilizzato ComplexExpand, per farlo funzionare anche con valori letterali.

La seconda funzione sarà definita nel seguente modo:

```
algtoexp[x_] := ComplexExpand[Abs[x]] Exp[I ComplexExpand[Arg[x]]]
```

La retta, invece, sarà costruita dalla seguente funzione:

```
retta[xx_, yy_] :=
 Solve[ComplexExpand[Im[xx - yy]] / y == ComplexExpand[Re[xx - yy]] / x, y]
```

Andiamo a scrivere il nostro file:

```
(*:Mathematica Version:5.1*)

(*:Package Version:1.0*)

(*:Name:Complessi`Conversione`*)

(*:Author:Daniele Lupo*)

BeginPackage["Complessi`"]

algtoexp::usage =
 "algtoexp[x] scrive il numero complesso x in forma algebrica"

exptoalg::usage =
 "exptoalg[x] scrive il numero complesso x in forma esponenziale"

retta::usage =
 "retta[x,y] restituisce l'equazione
   della retta passante per i due numeri complessi"

exptoalg[x_] := ComplexExpand[Re[x]] + I ComplexExpand[Im[x]]
```

```
algtoexp[x_] := ComplexExpand[Abs[x]] Exp[I ComplexExpand[Arg[x]]]

retta[xx_, yy_] := Solve[
  ComplexExpand[(Im[xx] - y) / (Im[xx - yy])] ==
   ComplexExpand[Re[xx] - x / Re[xx - yy]],
  y
 ]

EndPackage[]
```

Una volta creato il nostro package, andiamo a richiamarlo nel notebook

```
<< "C:\\Documents and Settings\\Daniele\\Desktop\\complessi.m"
```

Adesso vediamo che, senza bisogno di riscrivere le funzioni, queste sono effettivamente già definite (potete fare la prova richiudendo ed aprendo *Mathematica*, e caricare direttamente il package):

```
algtoexp[3 + 9 I]
```

$$3\sqrt{10}\ e^{i\,\text{ArcTan}[3]}$$

```
exptoalg[%]
```

$$3 + 9\,i$$

```
retta[9 + 9 I, 7 + 6 I]
```

$$\left\{\left\{y \to \frac{3}{2}\,(-12 + x)\right\}\right\}$$

```
?retta
```

```
retta[x,y] restituisce l'equazione
  della retta passante per i due numeri complessi
```

Come potete vedere, tutto funziona esattamente come volevamo...

Potete quindi notare come possiamo personalizzare in questa maniera *Mathematica*, andando a crearci da soli quello che ci serve, se casomai non ci dovesse essere, ma mi sembra difficile... I primi tempi probabilmente troverete più utili i packages nel creare alias molto brevi di funzioni che utilizzate spesso e che sono lunghi, per esempio un comando Plot con diverse righe di opzioni di formattazione...

Se poi vogliamo caricare automaticamente un package, basta usare la seguente funzione:

`DeclarePackage["`*context*`", {"`*name*$_1$`", "`*name*$_2$`", … }]`	dichiara che il package deve essere automaticamente caricato quando un simbolo con uno qualsiasi dei nomi *name*$_i$ viene utilizzato

In questo modo, specifichiamo le funzioni che vogliamo siano automaticamente caricate, e da quale file vogliamo che vengano lette.

Ricordate soltanto (e lo ripeto per la seconda volta), di mantenere quanto più organizzati e specifici possibili i vostri packages. Dovrebbero essere progettati inmodo da poter risolvere dei compiti specifici, non per essere dei contenitori casuali di funzioni...

Nota

Abbiamo appena visto come poter creare i nostri packages. Tuttavia, *Mathematica* in fase di installazione copia nel computer parecchi packages predefiniti, utilissimi in molti casi, semplificando di molto la risoluzione di parecchi problemi. Nell'appendice A di questi appunti vi descriverò alcuni dei tanti packages, quelli che ritengo più utili. Vedete se non trovate quello che fa al caso vostro!!!